A REST Model for High Throughput Scheduling in Computational Grids

Ian James Stokes-Rees
Linacre College, Oxford

Thesis submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy at the University of Oxford

Michaelmas Term, 2006

Abstract

Current grid computing architectures have been based on cluster management and batch queuing systems, extended to a distributed, federated domain. These have shown shortcomings in terms of scalability, stability, and modularity. To address these problems, this dissertation applies architectural styles from the Internet and Web to the domain of generic computational grids. Using the REST style, a flexible model for grid resource interaction is developed which removes the need for any centralised services or specific protocols, thereby allowing a range of implementations and layering of further functionality. The context for resource interaction is a generalisation and formalisation of the Condor ClassAd match-making mechanism. This set theoretic model is described in depth, including the advantages and features which it realises. This RESTful style is also motivated by operational experience with existing grid infrastructures, and the design, operation, and performance of a proto-RESTful grid middleware package named DIRAC. This package was designed to provide for the LHCb particle physics experiment's "off-line" computational infrastructure, and was first exercised during a 6 month data challenge which utilised over 670 years of CPU time and produced 98 TB of data through 300,000 tasks executed at computing centres around the world. The design of DIRAC and performance measures from the data challenge are reported. The main contribution of this work is the development of a REST model for grid resource interaction. In particular, it allows resource templating for scheduling queues which provide a novel distributed and scalable approach to resource scheduling on the grid.

I dedicate this work to Emily and Maggie,
who have made it possible, and made it purposeful.

9 What has been will be again,
what has been done will be done again;
there is nothing new under the sun.

10 Is there anything of which one can say,
Look! This is something new?
It was here already, long ago;
it was here before our time.

11 There is no remembrance of men of old,
and even those who are yet to come
will not be remembered by those who follow.

Ecclesiastes 1:9-11

Acknowledgements

There are numerous groups of people who have, each in their own way, contributed to the four years of work you are now (in part) reading about. My supervisors Ian McArthur and Steve McKeever I thank for their constant encouragement and direction. Thank you for letting me pick the direction three years ago and then standing each side of me to keep it all on course. Within the rest of the Oxford Physics Grid group, this work has been improved by the camaraderie of Alexander Soroko, Carmine Cioffi, Matt Leslie, Stefan Stonjek, Chris Dennis, and Rhys Newman. Jeff Tseng's time and experience with physics computing has been invaluable, and I thank him for corralling us software-types. For an incredible and life changing year in Marseille, I have to thank Frank Harris for befriending me and introducing me to Andrei Tsaregorodtsev and the LHCb computing group based at CPPM. Great thanks to my partner in crime Vincent Garonne for all the time spent developing DIRAC and watching Sopranos at CERN. As well, thanks to everyone on the LHCb Computing Team. It was great to work so well together making "the grid" a reality for LHCb and particle physics.

These have been several exciting years for grid computing. The time spent with the various Oxford e-Science projects, our monthly lunches at St. Hugh's (not fish again!), and the Software Engineering seminar series provided an informal environment that deeply affected my thinking on grid computing. The results were additional insight into the broader challenges of the field, inspiration for new ideas, and much hilarity as we repeatedly discovered everyone was finding it pretty hard to make grid computing really "work".

Much of this work was developed on experimental kit rigged together and connected to the department network with the kind permission and even support of Pete Gronbech, Chris Hunter, and John Harris. Thank you for your patience and allowing me to keep turtle-pace Pentium's with Red Hat 7 running for so long.

I must also thank the patient readers of the various mailing lists I spent endless hours writing to, in search of solutions to the problems of grid computing. In

particular, thanks to tb-support, lcg-rollout, lhcb-paste, and globus-discuss.

Contributing in less concrete ways I also must thank the MER team at the FSA for the good times over the final 8 months (and for paying my rent), the time during which the bulk of this was actually written, for the many early morning walks and talks with Robin Mayfield along the canal, and for the Oxford Vineyard Church staff team, who have been incredible to work with and grow along side. In particular, my thanks to Andrew Myatt for patience, leadership, high expectations, and believing in my ability to do more and be more than I ever thought possible.

I have come to realise the importance of family in the last few years and must acknowledge the incredible influence my parents, Mary and Jim Stokes-Rees have had on my life. Thank you for putting in me the values and priorities which I have today. Thank you to Andrew, my brother, for constantly challenging me to remember how big and exciting the world is, the need to find balance, and the importance of life long learning.

Finally, thanks to the most important people in my life. Emily, for encouraging me and supporting me in every possible way over the past four years, Maggie, for giving me the world's best thesis break wrestling matches, and of course to God for the blessing of this opportunity and having a plan for my life. For everyone, I pray I will be able to make the most of it.

Statement of Authorship

The bulk of the work presented in this dissertation is solely my own. Conceptually, the idea of Condor-style prioritised task queues and dynamic queue re-ordering is attributed to Andrei Tsaregorodtsev, my supervisor while at the Centre de Physique des Particules de Marseille and one of the lead scientists for the LHCb grid software. These are core architectural aspect of the LHCb DIRAC grid infrastructure described in Chapter 3. This chapter is largely taken from the journal paper [1], where I was the primary author. Other sources of material in this chapter came from these conference papers which I co-authored: [2–4].

Within this chapter, the LHCb computing requirements (Section 3.5.1) are, as referenced, a summary of the salient points from the LHCb Computing Model [5]. The overall DIRAC Architecture was conceived by Andrei Tsaregorodtsev (Section 3.5.2). He also had full responsibility for the Data Management Service and the DIRAC Agent aspects (Sections 3.5.4 and 3.5.7, respectively). The Job Management Service described in Section 3.5.3 was implemented by Vincent Garonne, who also created Figure 3.5. The Monitoring and Accounting Services (Section 3.5.6) were conceived and implemented by Ricardo Graciani, Manuel Sanchez, and Vincent Garonne.

My involvement with the work described in Chapter 3 was implementing the DIRAC Agent (Section 3.5.7), the XMPP Instant Messaging infrastructure (Section 3.6.3), the service fault tolerance strategies (Section 3.6.4), the Configuration Service (Section 3.5.5), and prototyping the OGSI implementation of DIRAC (Section 3.5.8). The actual LHCb Data Challenge 2004 was, as described in Section 3.7, conducted by a team of 6-8 people, of which I was one. Vincent Garonne, Andrei Tsaregorodtsev, and myself had primary responsibility for the continuous operation of the DIRAC Services over the 6 month period of DC04. I had primary responsibility for the interface between DIRAC and the LHC Computing Grid (LCG), both in terms of the software interface within DIRAC, and operationally monitoring and managing these jobs. Towards the end of DC04 this task management responsibility was

shifted to Ricardo Graciani. All of the discussion, metrics, plots, and analysis found in Sections 3.7 and 3.8 are entirely mine, based on work I did to merge task-level logs, DIRAC logs, and LCG logs over 6 months and 330,000 tasks (this work is briefly described in Section 3.8.2).

The work in the remaining chapters is exclusively my own.

Contents

Chapter 1

Scalable Computational Grids

One of the main goals of computational grids is to facilitate the interorganisational sharing of data and computing resources in a ubiquitous manner, similar to the way in which the World Wide Web has facilitated global information sharing, and the Internet global networking. While the inherent complexity of remote task execution and a multitude of strategies for work-flow management and data replication may preclude a single "Global Grid" from emerging, it is still of great value to consider combining the strategies which have contributed to the success of the Internet and the Web into a grid architecture which could operate on an Internet-scale.

The primary motivation for this work is the present lack of a stable, scalable computational grid infrastructure. Existing systems are largely based on extending distributed computing paradigms and single-site/single-organisation batch computing systems. These approaches have failed to demonstrate scalability, usability, and manageability, therefore it is necessary to consider if fundamentally different approaches are required. The underlying usage scenario for this work is drawn from simulation and analysis tasks typical of a particle physics experimental environment. In particular, this research has been funded through the United Kingdom Particle Physics and Astronomy Research Council (PPARC)[1] and the author has been affiliated with the LHCb experiment[2] based at CERN[3], in Geneva, Switzerland.

This dissertation considers architectural aspects from the Internet and Web and proposes a model for distributed, scalable grid computing. Five major areas are covered: a report on the DIRAC grid infrastructure developed by the author for the CERN LHCb project; analysis of a 6 month particle physics grid computing run; a

[1]See http://www.pparc.ac.uk/
[2]See http://lhcb.cern.ch/
[3]See http://www.cern.ch/

survey of grid task and resource description languages; a RESTful (Representational State Transfer) model for grid resource interaction; and scheduling and resource management properties derived from the RESTful grid model. REST is the name given to the architectural style used to guide the development of the Web, and was coined by Roy Fielding, one of the original architects of the World Wide Web, and who wrote the `httpd` server and co-authored the HTTP[6] and URI[7] IETF specifications.

This is one part design report and operational experience, and one part theoretical model. Chapter 3 covers the DIRAC design and operational experience, while Chapters 5, 6, and 7 introduce a set-theory based grid resource description model which follows the REST architectural style. Chapter 8 then explores, among other things, the scheduling properties of this model, showing how it enables an effective scalable distributed scheduling strategy while avoiding attempts at finding an optimal schedule, which is an NP-complete problem (and therefore intractable in a grid environment)[8]. Chapter 4 provides the bridge between the DIRAC operational experience sections and the theoretical sections by detailing the characteristics of existing grid resource description strategies prior to presenting a new model which is partially novel, and partially a synthesis of the best of the existing models. The overall vision, which is only just begun in the work presented here, is a RESTful grid. For that reason, Chapter 2 is found at the start of the work in order to set the scene by describing the components of the RESTful grid, and defining the terminology used in the remainder of the dissertation. As the theoretical portion of this work is limited to the initial representation, composition, and scheduling of grid resources, Chapter 9 provides a summary of conclusions and discussion of the future work required to more fully realise a RESTful grid.

1.1 The RESTful Grid Vision

This dissertation presents the first steps towards an architecture for large computational grids based on the REST[9] principles which guided the development of the Web and HTTP. As Fielding puts it, the Web consists of "communicating large-grain data objects across high-latency networks and multiple trust boundaries"[9] – just one of many properties also shared by a grid. Computational grids draw closer similarity to the Web than they do to distributed applications or batch systems, the latter two being the more common influences for grid architectures. Consider the following which are common to grids and the Web, but typically not found in tra-

ditional distributed systems or batch systems: heterogeneity, dynamism, federated or untrusted interconnected systems, high fault rate, and a multiplicity of security domains. The central contribution of this work is to re-phrase a computational grid using a REST architecture, focusing on describing all elements of the system as "resources" using a common description language, and restating interaction of these resources as "compositions", using set theory and set comprehensions, while leaving the architecture sufficiently open (under or unspecified) to allow independent client "rendering" (interpretation) of resource representations and the realisation of resource compositions formed within the framework presented here.

The REST architectural style consists of a stateless, layered, cacheable, client/server model, where the "visible" components of the system consist of resource representations, accessed via a uniform resource naming scheme. REST emphasises a simple scalable resource-centred distributed architecture rather than common software interfaces (*e.g.* libraries, protocols, APIs) as the basis of a large scale federated computing environment. This is sometimes described as the REST style emphasising a plethora of "nouns" (named object classes, or "resources") within the system, and a limited or constrained set of "verbs" (actions which can be carried out on the "nouns"), where these "verbs" can be applied universally to the "nouns" (*i.e.* providing a limited but universal interface to the unconstrained set of object classes within the system)[10]. This, and other aspects of the REST architectural style, are clearly in stark contrast to the more common Service Oriented Architecture (SOA) or Object Oriented Design which emphasise service or object interfaces and data hiding (*i.e.* the importance and richness of many well defined "verbs", specific to each particular "noun").

The RESTful grid design described here is inspired by successful Internet standards – HTTP[6], DNS[11], and the Web[12] – and an approach to"cycle-scavenging" championed by the highly successful Condor Project[13], namely High Throughput Computing (HTC)[14]. The RESTful approach is conducive to simplicity and replication which are seen as essential properties for robustness and scalability. A REST approach naturally avoids the monolithic and homogeneous nature of existing grid systems by decoupling descriptors (resource representations) from services which act on those descriptors. Federations of heterogeneous computing resources on an Internet scale require different approaches from those currently being developed or adopted[15]. This argument in favour of a different design paradigm and fundamentally different software architecture for Internet-scale computing has been presented by Rosenblum [16] and Fielding [9], and is suggested in "Architectural Principles of

the Internet"[17].

Preliminary verification of this strategy can be found in the LHCb particle physics experiment which has developed a proto-RESTful grid infrastructure. Further application of a RESTful approach to grid computing shown in the grid model presented later in this dissertation also reveal attractive properties not found in existing systems. It is argued that priority must be given to descriptions of computational tasks and resources which succinctly capture their salient features, yet also allow for extensibility. An emphasis of this proposed architecture is to recognise the duality which exists between resources and tasks, to abstract that commonality, and allow task/resource matching to be driven equivalently by the task owner, the resource owner, a third party, or some combination of all three. This builds on the concept of symmetric matchmaking developed by the Condor Project[13], where computing resources can constrain the tasks they will accept, and tasks can constrain the set of computing resources they are prepared to run on. By developing a model for this based in set theory it is then possible to derive valuable properties which greatly facilitate scheduling in a grid environment. This is also abstracted to general pair-wise and n-way resource composition.

1.2 Key Goals

The key underlying goals for a generic computational grid are enumerated below and will be referred to repeatedly throughout this work:

KG1 Scalability The grid architecture must be able to operate on a scale the size of the Internet in terms of hosts, users, files, and domains.

KG2 Reliability The grid architecture must be *fault tolerant*. Failure of any one component or area of the grid must have little or no effect on other areas.

KG3 Usability Users must be able to interact easily with the grid.

KG4 Extensibility It must be possible for developers to extend, replace, and interface with the grid infrastructure. This is associated with *modularity* and *replaceability* of grid components.

KG5 Manageability Administrators must be able to easily deploy, monitor, and maintain aspects of the grid under their authority.

KG6 Security Users, hosts, and resources must have a flexible but secure mechanism for controlling access. Delegation of authority is also critical.

From an engineering perspective, both DIRAC and the REST model which extends DIRAC, are of particular value because they directly address Key Goals 1 to 5 in a realisable way. Key Goal 6, Security, has not fully been incorporated into DIRAC, however the REST model provides a number of avenues for identity management, access control policies, and delegation to be developed on top of the RESTful grid architecture presented here.

1.3 Single Task Particle Physics Use Case

While consideration has been made not to preclude advanced features of computational grids such as work flows, parallel processing, resource coordination, accounting, multiway resource composition and advanced reservation, this dissertation only discusses the simple use case of self-contained remote task execution. As will be described, this base case has yet to be satisfactorily addressed in what will be defined, for the purposes of this dissertation, as a *computational grid* (see Section 2.9).

The particle physics community, in particular, has an impending demand for a reliable computational grid that will handle exactly this simple use case. By the end of 2007 the four Large Hadron Collider (LHC) experiments based at CERN will require a global computational grid of 140,000 of today's fastest CPUs operating 24 hours a day, 365 days of the year, and accessing 47 PB of data[5, 18–20]. These CPUs will be spread across hundreds of sites around the world, and also often be shared with other non-physics users local to those sites. Furthermore, the data access will, in general, be random and high volume, meaning the transfer of data from its storage location to an executing process could require significant time and bandwidth. This consideration must be factored in when task scheduling occurs.

The combined LHC computing load will be a significant test of the most basic objective of grid computing: large-scale sharing of federated computing resources and distributed data over a long time period across a diverse collection of *ad hoc* user groups. Addressing this successfully will provide a platform for many other applications and a stepping stone to the more advanced functionality mentioned above.

DIRAC, implemented for use by the LHCb experiment in 2004, was utilised for the execution of 300,000 tasks consuming 670 CPU years and producing 98 TB of data at dozens of sites located around the world. The performance of this system

and operational experience led to the refinement of the RESTful grid architecture presented herein through a better understanding of failure modes, bottlenecks, and functional requirements. Furthermore, DIRAC operated in parallel with the LHC Computing Grid, providing numerous points of comparison with another grid infrastructure.

1.4 Contributions

This dissertation makes the following contributions to the field of computational grid research:

1. Presentation of the DIRAC grid infrastructure, developed for the CERN LHCb experiment. This service oriented architecture has been developed to meet the Key Goals described in Section 1.2 and address the needs of particle physics computing, which it has done successfully.

2. A study of a large grid computing run. The results from over 300,000 long-running jobs executed over 200 days on a global network of sites provided invaluable insight into real grid computing issues[1]. Operational issues, architectural issues, and overall performance were all examined. This grid computing infrastructure incorporated early implementations of the architecture proposed by this dissertation[3], thus providing an opportunity to validate the model, and resulted in an improved understanding of the system requirements.

3. A survey and critique of existing batch system and grid description languages. This covers nine different task and computing resource descriptors, drawing out those properties which facilitate the Key Goals, and those which obstruct it. An analogy of a grid task to an operating system process is also made, in order to highlight key features and complexities of describing and managing grid task state throughout its lifetime.

4. The observations from the survey form the basis for the proposed RESTful Grid Resource Description Language. This focuses on the aspects for initial description leading towards matching and scheduling of resources, and support for grid process description throughout the full task life-cycle. Well defined properties of this model are developed by utilising a set theory approach. This is a refinement and extension of the Condor ClassAd language, and avoids the need for tri-state logic (*true, false* and *undefined* states, and the consequent

3×3 truth tables), thereby significantly simplifying comprehensibility and implementation.

5. A distributed grid scheduling architecture. This extends and generalises the concept of "matchmaking", originating from the Condor project[21]. Push, pull, and third party scheduling mechanisms for grid tasks are proposed. Support for hierarchical matchmaking via templates, gang scheduling, and priority pools are all discussed. It is shown how this strategy avoids the NP-complete problem of finding an optimal schedule while still efficiently providing an effective schedule.

The emphasis of this work is on laying a foundation for a practical architecture for task management in scalable, generic, computational grids. Issues around security, data management and networking are only touched on briefly, as this work has focused on the REST representation of grid resources, rather than interactions (beyond composition and scheduling) between those resources.

Chapter 2

Terminology

This chapter defines important terms used throughout this dissertation. It lays a conceptual foundation for the idea of a RESTful computational grid by describing the constituent aspects and their interrelationships.

2.1 Elements, Form, Rationale

Specific terms regarding the proposed architecture are taken primarily from Perry and Wolf[22] and Fielding[9].

Perry and Wolf propose that software architecture consists of *elements*, *form*, and *rationale*[22]. The *elements* are further broken down into three different classes: *processing elements*, *data elements*, and *connecting elements*. The *form* refers to how the *elements* are structured, while the *rationale* presents the motivation.

Fielding chooses to use the terms *components*, *data*, and *connectors* for Perry and Wolf's three classes of *elements* and *configuration* in place of *form*[9]. This dissertation will use Fielding's terms. For completeness we reproduce their definitions here:

Component An abstract unit of software instructions and internal state that provides a transformation of data via its interface.

Datum An element of information that is transferred from a component, or received by a component, via a connector.

Connector An abstract mechanism that mediates communication, coordination, or cooperation among components.

Configuration The structure of architectural relationships among components, connectors, and data during a period of system run-time.

Figure 2.1 illustrates the relationships of this terminology. The *system architecture* is made up of a *rationale*, which describes the motivation and objectives of the overall architecture, the *elements* which are the components which make up the architecture, and a *configuration* which describes how the components relate to each other. The elements can further be broken down into *components*, containing the processing rules or intelligence of the system (*e.g.* the software), *data*, which contain the information on which the components operate, and the *connectors*, which provide the interfaces, connections, filters, and buffers over which the components interact and the data items are transmitted.

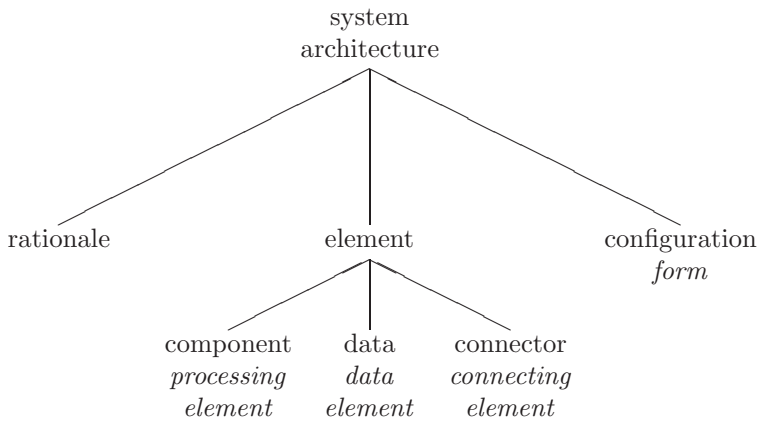

Figure 2.1: *Hierarchy of system architecture terminology (italics indicate original terms used by Perry and Wolf[22])*

2.2 Resources and Representations

In the REST architecture *components* and *data* are *resources* with corresponding *identifiers*. Resources are hidden entities which are only observed through their *representations*, and operations on resources are done by sending the *representation* and *representation meta-data* (that is, what to do with that representation) to a particular resource[9]. For example, a web server may provide English or German language representations of a particular web-page resource dependent upon the request or request meta-data. One or more identifiers are associated with a single resource, and access to a resource is achieved via one of these identifiers. The information which is available concerning a particular resource is contained within the

resource representation, which may be only a sub-set of the entire resource's state and may be interpreted (or rendered) in any way by the receiver.

A key aspect of a RESTful style is to phrase the entities within the system as resources, emphasising a common representation of those resources' state, and de-emphasising an architecturally asserted behaviour, protocol, or interface on those resources. Figure 2.2 illustrates the ontological decomposition of resource classes within the RESTful grid model. Behavioural inter-relationships (*e.g.* user resource creates and submits task resource which runs on executor resource), and compositional relationships (*e.g.* data resources associated with a particular storage resource) are explicitly unspecified in a RESTful architecture and are left to user communities or applications to assert on top of the underlying RESTful architecture. The first division of resource classes is into *host, user, task* and *data* resources, where hosts represent physical hardware, users are the people who interact with the system, tasks which represent actions or work-flows initiated by users, and data which is created or used by user tasks. Hosts are further divided into *executors* and *storage*, the former representing systems focussed on data processing, and the latter data storage. Service and agent resource classes are discussed in a later section.

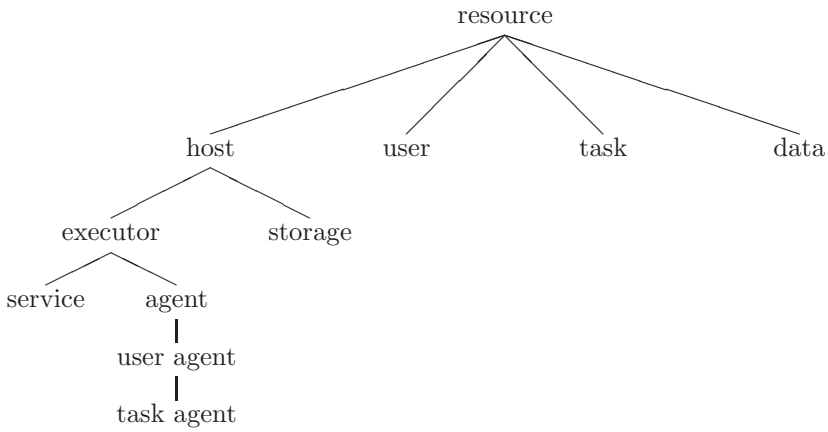

Figure 2.2: *Resource class relationships*

It is important to note that in a REST context the term "resource" takes on a much broader meaning than is commonly used within the grid computing domain. A grid resource is usually considered only to be a computing resource: either a single CPU, a node, a task queue, or a computing cluster. This work will refer to such a resource as an "executor" or a "computing resource". The more general definition of "resource" was deemed appropriate as a key position of this dissertation is the similarity between tasks and executors from the perspective of symmetric scheduling,

and the desire to model a computational grid in a RESTful way, where the primary entities are all "resources".

2.3 Representational State Transfer

The Representational State Transfer (REST) architectural style presented by Fielding forms the foundation for the architecture described in this dissertation. REST is a retrospective description of the principles which led to the design of the Web and HTTP, which in turn were built on the "end-to-end argument"[23] which formed the basis for the early Internet protocols and architecture evolutionary process[17, 24].

Fielding writes:

> REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self descriptive messages; and hypermedia as the engine of applications state.[9]

REST emphasises representation of system state over operational interfaces, leaving interpretation and manipulation of that state under (or un-) specified. In REST, units of system state are representations of resources and first class objects. This is in stark contrast to an Object Oriented paradigm which emphasises exactly the opposite, with state encapsulated and hidden inside objects with strict interface definitions. They key REST properties relevant to a RESTful grid are:

1. Stateless services and interaction, allowing for service replication/parallelism and caching.

2. Communication and interaction conducted (primarily) through the transfer of resource representations.

3. "Hypermedia as the engine of application state"[9], meaning system state and state transition alternatives are contained within persistent resources, rather than executing applications or services.

4. Dynamic, unspecified, and "hidden" resource and service component topology, thus alleviating the problem of brittleness, as the topology can continuously change. This is facilitated by a consistent, uniform, and loosely bound resource identification mechanism.

5. Uniform resource naming (*e.g.* URIs for the Web).

6. Hidden resource state, exposed resource representation.
7. Consumer-driven interpretation of resource state (*i.e.* client-side rendering of representation).
8. Decoupling and flexibility due to loose binding of references to resources (*i.e.* references to non-existent resources are permitted).
9. Dynamism through late binding of references to representations (*i.e.* representation of a particular resource is realised as late as possible "at run-time").
10. Content negotiation, returning a resource representation customised to a particular resource request or the preferences of the client.
11. Client-side and server-side cacheable representations.

The main observation is that a large, federated, dynamic, and generic distributed system requires flexibility in utilisation to allow different communities to develop interaction patterns suitable to their needs, while still having design constraints which allow an underlying common infrastructure to be put in place. REST constrains a system by asserting stateless, connectionless, cacheable client/server interactions with a uniform entity naming system, while providing design freedom (for higher layers) in terms of interpretation of resource representations and operations (actions) on those representations. The RESTful grid model, therefore, does not assert any behavioural constraints on grid resources and instead constrains resources in terms of how they are described and that the first "stage" of resource interaction is understood through resource composition, based on a strong set theoretic model describing resource properties.

2.4 Identities and Delegation

In this work, the term *identity* will use a broad definition that may contain verifiable properties regarding a resource. In this sense, an *identity* is also a resource.

Resources may *delegate* part of their identity to another resource, providing that second resource with a mechanism to act on behalf of the first resource. Instantiation and identity management of a particular resource or its delegate is the responsibility of the instantiator, within an identification namespace delegated from another resource. This means there may legally be multiple instances of the same resource within the system, each with multiple identifiers. Four examples of this are files,

tasks, users, and executors. For each one of these it is possible to imagine a single "abstract" or "true" instance of the resource, with multiple "actual" instantiations within a grid system, and each one of those instances having multiple identifiers associated with it. Of course in practice such replication, if permitted at all by a particular implementation, must be carefully managed in order to manage consistency and equivalence of resources.

REST describes *identifiers* simply as pointers to a resource representation. In the Web context, these identifiers are URIs. Here, there is no restriction given to the mechanism for resource identifiers, although URIs would be suitable in many cases. Identifiers and identities are quite independent of each other.

This dissertation does not attempt to specify any particular mechanism of identity exchange, delegation, verification, or use. Suffice it to say that various mechanisms exist to do this to different degrees of functionality, secrecy, and security (*e.g.* PERMIS[25], X.509[26–28], Globus proxy certificates[29], and LCG Virtual Organisation Management System[30, 31]). Furthermore, analogous to the Web, a large computational grid infrastructure could reasonably be expected to support a range of different approaches to security. Although the RESTful grid architecture presented here does not presume to use URIs and HTTP, one of the experimental systems developed does utilise URIs to reference resources accessed over HTTP and representations realised in XML. In this scenario, all interactions could be subject to any one of: X.509-based certificate authentication, HTTPS-encrypted username/password, plain username/password, or client-side cookies. Expanding the range of protocols beyond HTTP, a single work-flow could be envisioned to include data access via the Secure Shell protocol (ssh)[32] using DSA keys[33], OpenPGP[34] to sign an SMTP email message, one X.509 certificate to submit a sub-task to one cluster, and another X.509 proxy certificate to make a data base query to a remote service. The point of this illustration is to emphasise the importance of decoupling a particular security strategy from the underlying computational grid infrastructure, thereby allowing a range of different security strategies to be layered on top. In this domain, the objective is to avoid precluding any particular security strategy of a higher architectural layer.

2.5 Hosts, Executors and Storage

This architecture describes four autonomous types of resources: *executors, storage, services* and *agents*. In Unified Modelling Language (UML) parlance, these would be

considered *actors* which initiate and respond to change within the system. A meta-class of resource is the *host* which provides the underlying system for an *executor* or *storage* resource.

Hosts represent abstractions of hardware – for example, a particular CPU or operating system instance – or a system acting as a gateway to a collection of hardware resources. A host will typically have full control (e.g. "root" access) over the resource it represents and all its subordinate resources, and may encapsulate the various "core" processing components which make up that resource (*e.g.* storage, operating system, and associated utilities).

Executor resources, or *executors* for brevity, represent the computational resources within the system. An executor may contain and manage a set of subordinate services and agents.

Storage resources provide the infrastructure which manage data resources. This is included for completeness, however a system for data management is not an emphasis of this research, therefore is not covered in any depth. In the particle physics use case, efficient data management is a critical aspect of the computational infrastructure, given the volume of data which is produced, processed, and queried. As such, data catalogues, meta-data, access control, staging and replication are all areas where extensive work has been done. In a fashion similar to the discussion earlier on security, data management services must be layered on top of an underlying grid architecture. The RESTful architecture presented here allows a range of different data management techniques to be used without prejudice for one or another.

2.6 Services and Agents

Services represent processing components which are "externally" accessible (meaning accessible beyond the context of the immediate component which started them). These may operate on behalf of a host or user and hold delegated identities, or have their own identities. Service entities typically are reactive (rather than proactive), long lived, stateless, and instantiated by a host, although none of these are essential properties. Services may instantiate agents in order to monitor or carry out some request.

Agents are similar to services, but instead are typically proactive (goal oriented), have a limited life time (until the goal is achieved), are stateful, and are instantiated by users, services, or other agents. They accept and delegate identities and may have their own identities. They are not necessarily externally accessible.

2.7 Users

Users are human beings with a set of identities available to them. Users are also considered a type of resource. They utilise "interactive" processing components to interact with the system. A user may create a *delegate* which will operate autonomously (without user interaction) on behalf of the user. The delegate may appear as a service or agent. A typical form of *user agent* is a *task agent* which operates on a *task resource* (via its representation) to execute the task.

2.8 Grid Task

A self-contained piece of work to take place "on the grid" will be termed a *grid task* or simply *task*. In the general sense a task can have dependencies with other tasks, or can itself consist of a sequence of sub-tasks. In many ways it closely parallels the concept of a "process" in an operating system, with the key difference being that a grid task may not be actively executing. This allows for a task to be a template prior to execution, and to continue to exist after the execution of the task has completed. A grid task is synonymous with a "grid job" (although it should be noted that the GGF Grid Scheduling Dictionary[35] makes a subtle distinction between the two).

The overhead and latencies involved in cross-site communication is such that a grid task is taken to be fairly coarse grained, typically requiring on the order of minutes to hours to execute, and an overall lifetime of months or even years. Finer grained tasks will suffer from a high overhead to computation ratio (OCR), and would more suitably be batched together as a coarser grained single grid task. Furthermore, task-external communication performance in a grid environment is expected to be relatively slow, unreliable, and unpredictable, meaning that a low communication to computation ratio (CCR) is expected. This does not preclude a single grid task from encapsulating a parallel execution involving a large number of co-located processors.

A task consists of some combination of: processing operations, data operations, constraints, characteristics, configuration, security policy, accounts, and an identity set.

2.9 Computational Grids

The definition for a *computational grid* is taken from the context of particle physics computing. It consists of 100 or more computing sites each consisting of 1-10,000 independent nodes. A *computing site* is an autonomous computing centre, typically located at a single physical location (for example, a university, a research centre, or a company). Each site is assumed to have complete authority over their own resources. A *node* is the smallest managed computing resource, which is typically a dual or quad processor rack mounted computer. The total number of nodes provided by all sites is 100,000 or more, and the sites are interconnected via the Internet. The user base consists of 1000-1,000,000 users, arranged in dynamic autonomous virtual organisations which can self-create, and utilising minimally or un-coordinated user identity and accounting systems. The per-centre storage will typically be 1-1,000 TB, with 10-10,000 Mb/s bandwidth between nodes and between sites. The grid will typically be loaded with computing tasks at 0.01 (low task contention) to 100 times (high task contention) the available computing resources, with some "hot spots" possibly having a TCR (Task Computing Power Ratio) of up to 1000 (that is, 1000 tasks queued per available CPU).

There is no assumed coordination between any two centres, and failures, network topology changes, and dramatically varying resource demands are regular but unpredictable. Furthermore network partitions, both intentional (nodes disconnecting from the network) and accidental (power outages, network or equipment failure, etc.) are a regular feature. The grid is also considered to be a hostile environment, in that some nodes and users may attempt to corrupt data, interfere with execution, and hijack identities or resources. The hardware and software which makes up the grid is entirely heterogeneous, and the overall system state is hidden. That is to say, there can be no expectation of discovering the complete and consistent system state. From any point within the grid, only portions of the system can be expected to be visible (*i.e.* have state information available), however it is likely that this view will provide out of date and possibly erroneous information.

Table 2.1 lists typical ranges for some of these properties, and includes a column with reference values for a representative computational grid. The reference values are rounded to the nearest order of magnitude typical for the combined LHC experiments' computational needs.

Use of the term *grid* refers to computational grids, unless the context suggests otherwise.

Characteristic	Range	Reference Value
Sites	100+	100
CPUs/Site	1-10,000	1,000
Storage/Site	1-1,000 TB	100 TB
Total CPUs	$>$ 10,000	100,000
Users	1000-1,000,000	10,000
Task Load	0.01-100	10
Task Duration	$>$ 10s	10 hr
Intra-Site Bandwidth	100-1000 Mb/s	1,000 Mb/s
Inter-Site Bandwidth	10-10,000 Mb/s	100 Mb/s

Table 2.1: *General quantitative characteristics of a computational grid.*

Chapter 3

Grid Computing for Particle Physics

Having introduced and defined the key aspects of a RESTful grid architecture in the previous chapter, this chapter summarises experience from within the LHCb experiment of designing, operating, and using a large computational grid. It is composed of material from a number of reports and papers which presented the work of the 2004 LHCb Data Challenge[1], the design of the DIRAC grid software system[3], and a study of the performance of the computational grid used for the Data Challenge[36]. It provides an overview of the computational needs of current particle physics research and considers experience with existing grid architectures to meet those requirements. The DIRAC architecture applied proto-RESTful grid principles in terms of decoupled, stateless services, generic resource composition, and client-driven resource access and resource interpretation, thereby serving as the basis for the work developed in later chapters.

3.1 Introduction

The particle physics community is one of the strongest drivers for the development of computational grids. Experimental particle physics is breaking new ground in our understanding of the most basic laws of the universe and therefore receives significant research funding around the world. The nature of experimental particle physics is such that individual experiments will typically have a lifetime of fifteen to twenty years, cost hundreds of millions of pounds, and involve thousands of people. The collaborators for each experiment are distributed across institutions around the world. In the past it was possible for the majority of experiment related computing to take place at the experiment site with "local" storage of all data, however this will not be possible with the newest generation of experiments. For example, the four new CERN experiments estimate they will each require thousands of dedicated CPUs

(by today's computing power) on a continuous basis just for the reconstruction stage which converts the raw data produced by the detectors into physics "events"[5, 18–20]. This load must be distributed to national and institutional computing centres as CERN is not in a position to provide this quantity of dedicated processors. In fact, this initial reconstruction phase is only the tip of the iceberg, making up less than 20% of the total computational demand of the four experiments. Other large physics centres such as the Stanford Linear Accelerator (SLAC) and Fermi National Laboratory (FNAL) in the United States have similar requirements to distribute significant amounts of data and computing to remote sites.

The new generation of particle physics experiments have orders of magnitude greater demands for data storage and data processing. While it would be conceivable to centralise all data and processing at CERN, using a large but otherwise traditional computing cluster providing a "batch farm", this is undesirable from a number of perspectives. As an organisation, CERN attempts to distribute as much responsibility to member states and their respective physics institutes as possible. This principle reduces administration and management at CERN, and provides a better mechanism for responsibility, accountability, and dissemination of knowledge by having CERN's work completed by members within their home institutes. Furthermore, a distributed model allows member states or individual institutes to commit computing resources relative to their own priorities and capabilities. This issue dovetails with the desire to have the large volume of data replicated and available "close" to the computing centres which will process it, thus providing increased robustness, increased effective network bandwidth, and reduced latencies. Many of these issues are common to data and compute intensive applications and scientific research, especially given the increase in inter-organisation collaborative teams, where each sub-team may be participating in numerous collaborations. In this environment of *Virtual Organisations*, the importance of a robust framework for distribution of data and dynamic sharing of computing resources towards a common goal becomes clear. This scenario provides the motivation for Key Goals 1, 2 and 6 (Scalability, Reliability, and Security).

This chapter will look at experience with existing grid computing infrastructures for handling the requirements of particle physics computing. Based on these requirements and experience from deploying and utilising early grid systems, a new grid computing architecture was developed in 2003 and 2004 for the LHCb experiment. It followed many REST principles of atomicity and simplicity of services. In particular DIRAC featured stateless, connectionless services, a client-server model, and a

RESTful resource model for tasks and agents. The operation of this system led to the largest known utilisation of a global generic computational grid, consuming 670 CPU years over 6 months and producing 98 TB of data from 300,000 tasks. This experience led to key insights which have further refined the requirements definition for particle physics computational grids. This system, the results of its use in 2004, and the observations drawn from that experience are presented in later sections.

3.2 The CERN Large Hadron Collider

The four new CERN-based particle physics experiments – ATLAS, CMS, Alice, and LHCb – will all utilise the Large Hadron Collider (LHC) once the collider and detector construction is complete with a projected start date in 2007. As a proton/proton collider it will have a centre of mass energy of 14 TeV, many times greater than any existing collider, allowing it to examine high energy particles better than ever before, and some, such as the anticipated Higgs Boson, hopefully for the first time. The higher energy collisions will necessarily produce a larger number of particles, and the short life time of the highest energy particles requires very precise spatial, temporal, and energy measurements. This is now possible due to improved electronics and the decrease in computing and storage cost but will result in the production of unprecedented amounts of data which will then require analysis.

The detectors, while in operation, have particle collisions occurring at rates of up to 1 MHz. A series of "triggers" are used to down-sample this to only select collisions containing events of interest. This produces an output stream at a rate of 100-2000 Hz. Each "event" contains a picture of the particle collision as seen by the entire detector assembly. It contains information taken directly from the online data acquisition system. This includes for example, time stamps, signal pulse information and channel information from which spatial and other information can be deduced. Real-time calibration data concerning the configuration of the detector may also be included. The detectors are in full operational mode for a 7 month period each year, and during each operational day 14 hours are spent "filling" the particle ring, and then 10 hours "draining". The draining period is when the collisions occur and real data is gathered.

Figure 3.1 shows the historical and estimated annual storage and computing requirements for different particle physics experiments. It is compiled from [5, 18–20, 37, 38]. The LHC experiments anticipate 140 million SPEC Int 2000[39] years of processing power in 2008, which will be the first full year of normal operation of the

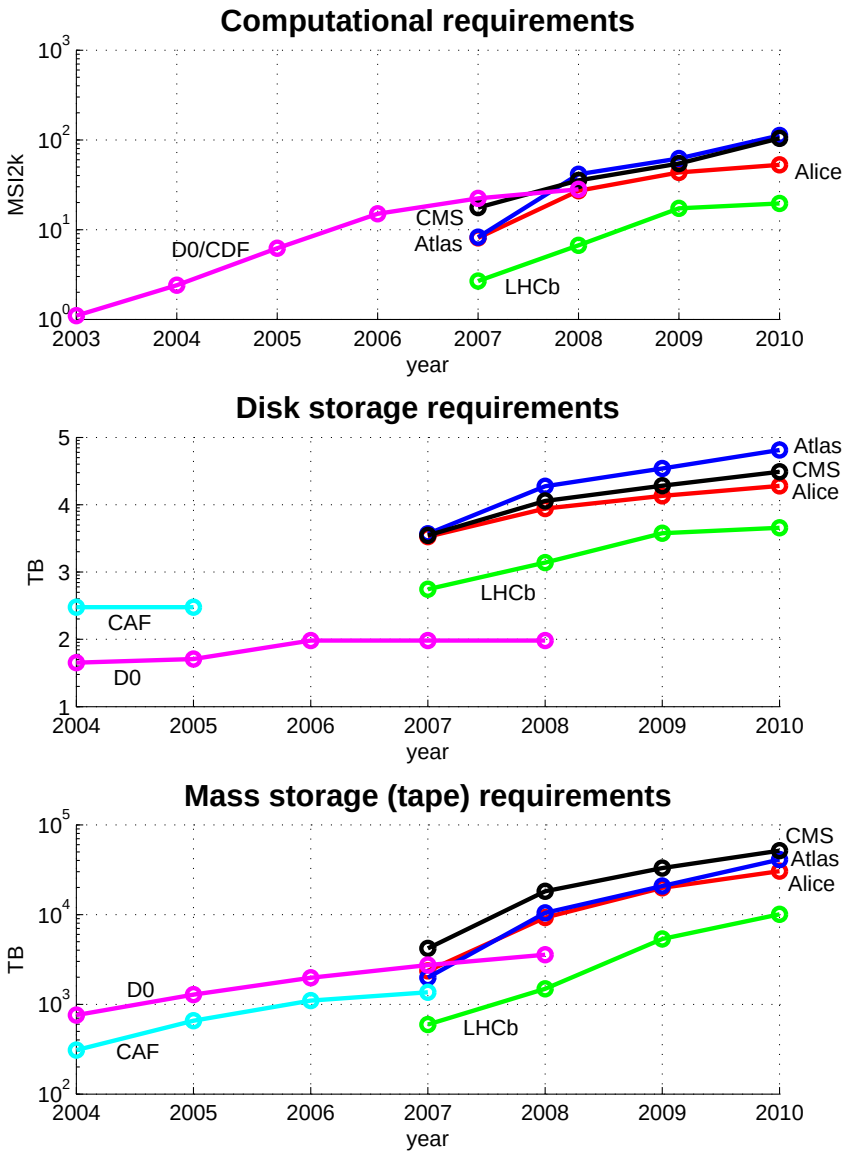

Figure 3.1: *Particle physics computing requirements. Processing, tape, and disk storage requirements for a selection of current large experiments. 2007 and beyond are estimates.*

LHC. This is approximately equivalent to 140,000 3GHz Intel Xeon CPUs operating continuously for a year. At the same time 47 PB of tape storage and 64 PB of disk storage are required[5, 18–20].

3.3 Typical Particle Physics Computing Model

Modern particle physics experiments have come to adopt similar strategies for organising and executing their simulations and data analysis. It is this common approach to mass data handling and processing which motivates the model described in later chapters. Furthermore, the field of grid computing largely grew out of physics computing requirements which were not met by existing cluster or supercomputer systems. For this reason, many of the existing grid computing strategies have been developed with physics computing requirements in mind. This section provides a summary of the computing model used by the CERN-based experiments.

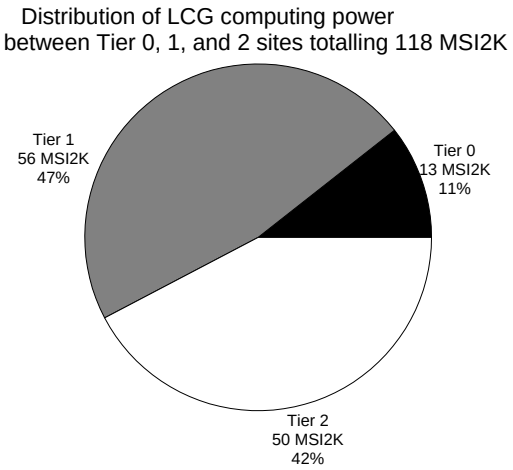

Figure 3.2: *Computing resource distribution for Tier 0, 1, and 2 sites committed to LCG in 2008.*

The CERN MONARC project[40] proposed a four tier system for distributing the processing load and data across a global network of physics sites[41]. The system is centred at CERN, called the Tier 0 site, where the data from the detector originates. Subsequent tiers consist of increasing numbers of sites, but the computing and storage capacity at each site is successively smaller, and the reliability of the site decreases. Tier 1 sites are typically large national computing centres with high reliability systems, dedicated support staff, high bandwidth networks, and large storage capacity. They are also termed "Regional Centres" and act as hubs for the Tier 2 to 4 sites under their umbrella. Tier 2 sites represent institution or university computing centres which are predominately dedicated to physics computing. Tier 3 sites are small clusters typically belonging to individual working groups, and Tier 4 represents individual computers, typically desktops or laptops belonging to

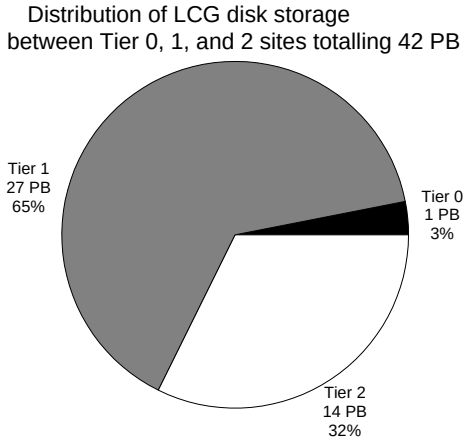

Figure 3.3: *Disk storage distribution for Tier 0, 1, and 2 sites committed to LCG in 2008.*

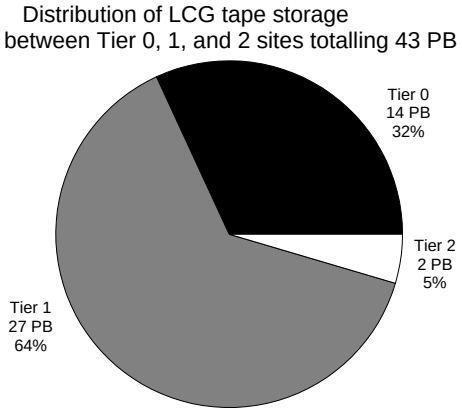

Figure 3.4: *Tape storage distribution for Tier 0, 1, and 2 sites committed to LCG in 2008.*

experimental collaborators. The current LCG Memorandum of Understanding details commitments of Tier 0, 1, and 2 sites and describes Service Level Agreements, expected operational environment, qualitative requirements, and quantitative characteristics for computing power, disk storage, tape storage, and network bandwidth committed by institutions involved with LCG[42]. Figures 3.2, 3.3, and 3.4 illustrate the distribution of some of these quantities between the Tier 0, 1 and 2 sites. It should be noted that there is a shortfall for all three of these, as the MoU commits 118 MSI2K computing resources, 42 PB of disk storage, and 43 PB of tape storage,

while the experiments' computing models require 140 MSI2K computing resources, 47 PB of disk storage, and 67 PB of tape storage. As the MoU is the most recent document, it is possible this is due to the latest official experiment computing models being out of date with the current computing requirements.

Within particle physics experiments grid computing is taken to mean the transmission, processing, and storage of data once it is on commodity components. Every experiment will contain custom hardware, electronics, sensors, and processing which is "close" to the detector and part of the real-time "on-line" system which is not considered part of the grid computing infrastructure. Grid computing provides facilities for what is known within the particle physics domain as "off-line" computing. There are three primary operating modes for particle physics grids: reconstruction, simulation, and analysis.

Reconstruction is very predictable, as it must be completed as the raw detector data is produced, during the annual 7 month operational period of the detector. This utilises stable software with parameters describing the detector configuration and calibration values. Reconstruction must take place in near-real time, meaning the data produced by the detector each day must be reconstructed into physics events within a day. The throughput of reconstruction must match the rate of data generation, as the detector operates continually during the 7 month period and anything less would lead to a permanent and growing backlog until either the operational period came to an end or the disk storage buffers were saturated and it became necessary to flush the data and later retrieve it from tape storage. It also allows for data quality monitoring which feeds back into controlling the run conditions. Beyond the annual operational window, the detector data may be reprocessed (that is, reconstructed a second or third time) based on improved software, calibration data, or detector models. This reconstruction is planned in advance and administered at the experiment level.

Simulation can be done by individuals, small working groups, or directed by the experiment. The experiment-level simulations are well defined in advance and typically consist of thousands of days of computing doing wide ranging simulation of detector response. Simulations conducted by individuals or working groups are much more chaotic, in terms of the data requirements, software requirements, and computing load.

Analysis is the most chaotic and consists of individual physicists or small working groups making cross-cutting selections of reconstructed data in a search for interesting physics. This will typically use locally customised software to analyse the data,

and will be even more unpredictable than the simulation work. These present some of the greatest challenges to grid computing as the requirements of these jobs vary widely.

3.4 LHC Computing Grid

In preparation for the computational demands of the LHC experiments, and in acknowledgement of a new paradigm for computation in scientific research, the European Data Grid (EDG) project was initiated in 2001 with the objective of developing a set of grid tools and an overall grid framework addressing security, data management, task management, and system monitoring[43]. The specific goals and use cases of this work were largely driven by the CERN LHC experiments, and described in two reports HEPCAL I and HEPCAL II (High Energy Physics Computing Application Layer)[44, 45], although consideration for other fields such as bio-medical research, economic analysis, and climate modelling were also included.

The EDG project looked to build on and stabilise the Globus Toolkit, which had been utilised successfully by many distributed computing applications, thereby providing a generic grid computing framework suitable for the full range of computing applications required by particle physicists. The product of this work would ultimately be delivered to the CERN-based computing group for the LHC which would coordinate the distribution, deployment, and operation of the LHC Computing Grid (LCG)[46]. Concerns with EDG in 2002 and 2003 led to the ARDA Report (Architectural Road-map for Distributed Analysis)[47], prepared jointly by members of the CERN experiments, which proposed a more decoupled and modular approach to LCG than EDG was providing. This report motivated, in part, the direction of the successor project to EDG, the Enabling Grids for e-Science in Europe (EGEE) project[48]. This project operated in parallel with LCG and looked to re-engineer the EDG software into a system named “gLite”.

It was at the point of transition when the ARDA Report was published and LCG and EGEE were commencing their work with the inherited EDG software that the work described in this dissertation was initiated. The LHCb experiment sought to implement aspects of the services proposed by ARDA, given the difficulties of working with EDG, in order to reduce the risk of relying completely on LCG for the experiment’s computational infrastructure.

3.5 DIRAC Grid Infrastructure

During 2003 and 2004 the CERN LHCb experiment developed the DIRAC computing infrastructure to support the transition from traditional batch farm computing at large computing centres to distributed grid computing. In the Spring of 2004 this system was simultaneously deployed at 12 European physics computing centres affiliated with the LHCb experiment and integrated with the LHC Computing Grid (LCG). There were four objectives for the 2004 Data Challenge:

1. Test the LHCb software chain for physics simulation and analysis;
2. Simulate physics events for the LHCb detector using Monte-Carlo methods;
3. Validate the LHCb computing model;
4. Test the LCG (grid) computing infrastructure (software, hardware, and processes).

The following sections describe the LHCb grid software which was developed to support this initiative, and examines the results from the Data Challenge. Many of the strategies and approaches espoused by this dissertation are motivated by the LHCb computing requirements, the prior experience and philosophical approach which led to the development of DIRAC, and the observations made and conclusions drawn from the utilisation of a real grid infrastructure during the 2004 LHCb Data Challenge.

The author's primary contributions to the DIRAC infrastructure were around the integration of OGSA [49] principles, exploring security options for DIRAC, interfacing DIRAC with LCG, designing and implementing a novel instant-messaging based monitoring system, designing and implementing a dynamic, distributed configuration service, and putting in place fault tolerance mechanisms for Services, Agents, and Tasks. In depth discussion is limited to these areas, however the other aspects of DIRAC are presented for completeness.

3.5.1 LHCb Computing Requirements

The LHCb Computing Model [5] describes the approach taken by the experiment for handling and processing data, outlining the storage and processing requirements for the next several years. There are four main types of offline processing: initial reconstruction of physics events from detector signals, stripping of detector data

to select particular channels (groups of "interesting" events), data analysis, and Monte Carlo simulation. There are also five main types of data produced by the experiment: RAW data, which represents the response of the detector to collision events after calibration adjustments have been applied; DST data, which are the reconstructed events from the RAW data; rDST data, which are reduced versions of the DST sufficient only for efficient pre-selection of full DST data sets; TAG data, for quick reference to DST data "of interest", which summarises event characteristics and references the relevant DST data; and N-tuples which are specially selected sets of events from DST data which contain comparable or groupable events.

Online processing takes place either within the detector assembly itself, or utilising computing resources co-located with the detector and are not relevant to this discussion except in that the general purpose computing hardware which is part of the online system can contribute to the grid environment when the detector is not operational.

The remaining processing is done in the offline system. DIRAC is the grid computing environment developed by LHCb to do this offline processing. It incorporates LCG computing resources and functionality while also, critically, allowing the integration of non-LCG resources (see Figure 3.5). This section reports on the experience of developing DIRAC, integrating it into the LCG grid environment, making use of existing middleware services and libraries, and advances from incorporating new technology such as instant messaging into the architecture.

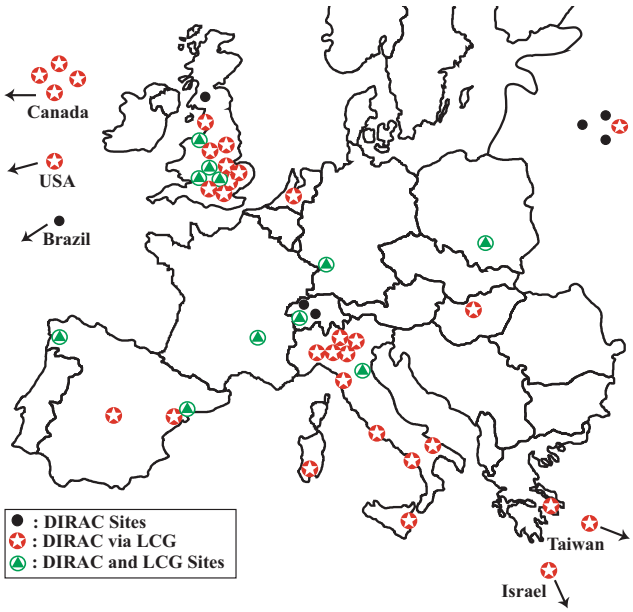

Figure 3.5: *Sites running DIRAC. This includes a mixture of grid-enabled sites and conventional computing centres.*

3.5.2 Architecture

DIRAC is designed following a lightweight Agent/Service model, which emphasises a *service oriented architecture* (SOA). It provides a scalable high throughput generic grid computing environment for uncoupled or loosely coupled long running computational tasks, requiring significant input data and producing large volumes of output data. The basic design objectives are to support 100,000 queued jobs, 10,000 running jobs, and 100 sites.

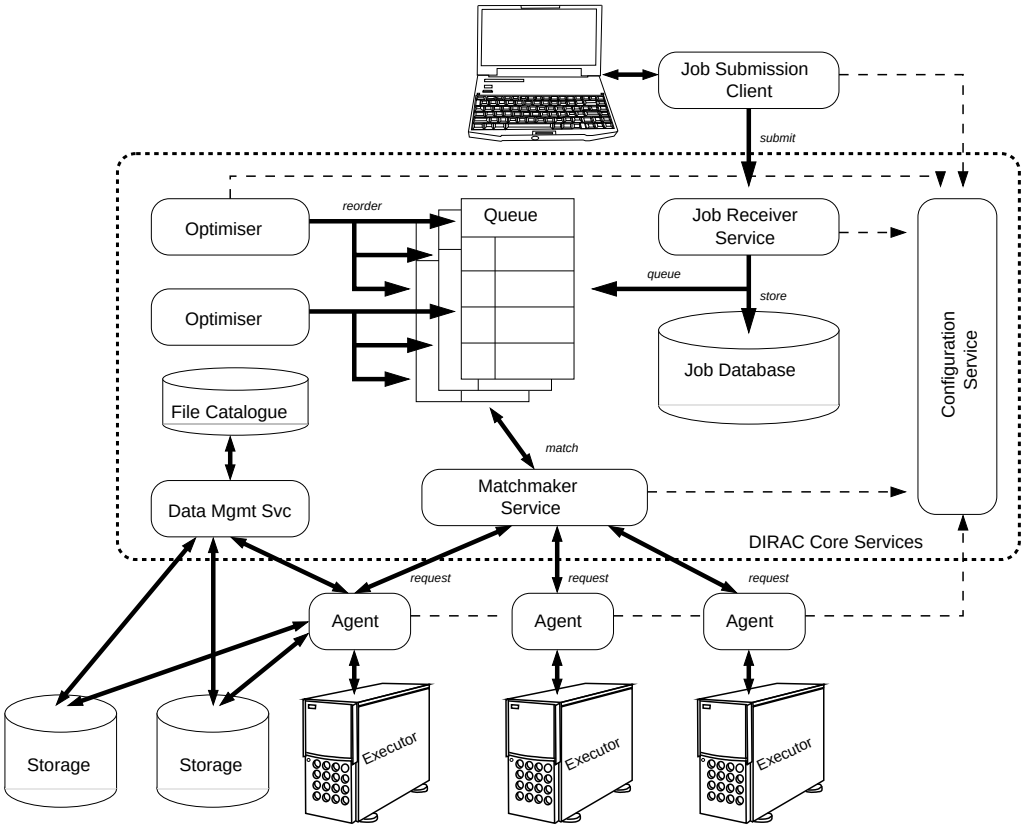

Figure 3.6: *DIRAC Core Services.*

The architecture is divided into five areas: Services, Agents, Executors, Storage, and User Interface. The core of the system is a set of independent, stateless, distributed *Services*. The services are meant to be administered centrally and deployed on a set of high availability machines. *Executors* and *Storage* are resources available at remote sites, beyond the control of any central administration. *Agents* run on each executor to monitor the resource availability, requesting jobs when possible from the DIRAC services. Figure 3.6 illustrates these components and their relationships.

The *User Interface* API provides access to the Services, for control, retrieval,

and monitoring of jobs and files. It has been incorporated into command line tools, GUIs, and web sites. A complete GUI interface for managing LHCb jobs has been produced by the Ganga project[50].

The general separation between Services and Agents is that Services are *stateless* and *reactive*, whereas Agents are *stateful* and *proactive*. The Services can be distributed across several machines, or run from a single server. This allows easy replication for redundancy and load-balancing.

3.5.3 Job Management Services

Jobs are described using the text based ClassAd Job Description Language (JDL) designed by the Condor project for use with the Condor Matchmaking scheduling system[21]. A JDL file is submitted to the Job Receiver Service which registers the job in the Job Database and notifies the Optimiser Service. The Optimiser Service sorts jobs into different job queues and dynamically re-prioritises queue ordering. Agents monitor availability of remote executors. When they detect "free slots", they submit a job request to the Matchmaker Service, which interrogates the various Job Queues and returns a suitable job, based on the resource's profile.

3.5.4 Data Management Services

The DIRAC Data Management Services provide fault tolerant transfers, replication, registration, and meta-data access for files both at DIRAC computing centres and long term mass storage sites.

A *Storage Element* (SE) is an abstracted interface to Internet-accessible storage. It is defined entirely by a host, a protocol, and a path. This definition is stored in the Configuration Service (see Section 3.5.5), and can be used by any Agent, Job, Service or User, either for retrieving or uploading files. Protocols currently supported by the SE include: gridftp, bbftp, sftp, ftp, http, rfio or local disk access.

The *File Catalogue Service* provides a simple interface for locating physical files from aliases and universal file identifiers. This has made it possible to utilise two independent File Catalogues, one from the already existing LHCb Bookkeeping Database, and another using the AliEn File Catalogue from the Alice experiment[51]. In the recent LHCb Data Challenge they were both filled with replica information in order to provide redundancy to this vital component of the data management system, and to allow performance comparisons to be made.

Within a running job, all outgoing data transfers are registered as *Transfer*

Requests in a transfer database local to each Agent. The requests contain all the necessary instructions to move a set of files in between the local storage and any of the SEs defined in the DIRAC system. Different replication, retry, and fail-over mechanisms exist to maximise the possibility of successfully transferring the data (see Section 3.6.4). This system decouples the data transfer from the job execution in a manner similar to that done by Condor Stork[52]. This also allows pipelining of execution and data transfer. It recognises that data placement is a significant, but under appreciated, part of a computational grid infrastructure.

3.5.5 Configuration Service

When working with large numbers of dynamic collaborating components, possibly with replication either on the same host or across a set of hosts, coordinating the configuration and information access for each of these components and between components is a difficult task. This is closely related to the classic Name Service problem, addressed in other contexts by DNS (Domain Name Service)[11], LDAP (Lightweight Directory Access Protocol)[53], UDDI (Universal Description, Discovery, and Integration)[54, 55], MDS (Monitoring and Discovering System)[56], and R-GMA (Relational Grid Monitoring Architecture) [57, 58], to name a few[2]. Of these alternatives, DNS came the closest to providing a simple, de-centralised system that did not require the installation and configuration of a central information server, or have particular preconceptions concerning the information contained by the service or the method of service access. DNS, however, still presented a sufficient level of complexity from both the implementation and end user perspective to warrant the development of a new service. The other approaches were all powerful, yet complex, and required significant infrastructure to utilise. Many design ideas for the DIRAC Configuration Service are taken from the DNS architecture, such as iterative navigation, hierarchical information, replication, and caching. The DIRAC Configuration Service featured a simple interface, with a conceptually simple design, replicating the concept of a configuration file.

The specific requirements were that every DIRAC Service, Agent, and Client required a uniform API which would provide:

- Local configuration and information;
- Global (system wide) configuration and information;
- Remote component configuration and information;
- Configuration and information sharing;

```
[ServiceA]
ServiceName = DIRAC Job Matcher

[AgentConfig]
Modules     = JobAgent TransferAgent

[JobAgent]
CEUniqueIds = in2p3.fr/pbs-short
AgentName   = TestModularAgent

[InfoService]
List = /etc/site-config.ini     \
       http://lbnts2.cern.ch    \
       http://marsanne.in2p3.fr
```

Listing 3.1: *Example configuration file.*

```
void    set     (section, option, value, [source])
value = get     (section, option, [source])
list  = options (section, [source])
list  = sections([source])
```

Listing 3.2: *Configuration Service API.*

- Overriding of global settings;
- Ease of deployment;
- Ease of updates;
- Robustness;
- Simplicity.

As such, and in keeping with the principles of simplicity and lightweight implementation, a network-enabled categorised name/value pair system was implemented, which overloads the Python *ConfigParser* API and utilises the Microsoft Windows "INI" file format. An example is shown in Listing 3.1. Components which use the Configuration Service do so via a *Local Configuration Service* (LCS). This retrieves information from a local file, from a remote service, or via a combination of the two.

The simplicity of this format means non-expert users can easily modify configuration files. As well, it presents an information data model which is conceptually easy to grasp. The goal was to present information to a software component as if it had come from a single local INI file. The basic interface is borrowed directly from the *ConfigParser* module and is shown in Listing 3.2.

To clarify the distinction between the local object which exposes this API and a remotely accessible service, two distinct classes were created: `LocalConfiguration-`

`Service` and `ConfigurationService`, respectively. The intention was that on a semantic level the APIs to these two objects will be identical, although syntactically, and due to particularities of the RPC mechanism, it is possible the API may vary.

A `LocalConfigurationService` object can be passed a number of information sources when it is created. This ordered list represents the hierarchy of sources which will be queried in order, either until a requested item is found or an exception returned once all sources have been attempted, thus indicating the item does not exist. This list of sources can be composed of a mixture of local files and remote sources. Local files are read directly into memory and not referenced again, while remote sources are queried only when necessary. This approach implies file based information is static and a snapshot is taken at object creation time, while remote information can be dynamic and subsequent requests may return different results. A mechanism exists to copy results from remote sources, placing a snapshot of those requested items in memory in the local object, avoiding subsequent calls to the remote service, but sacrificing the ability to catch changes to remote information.

There is also the option to create a `LocalConfigurationService` without any information sources, and simply add information to the object during program execution, or add a list of sources at a later point. Similarly it is possible to change the list of remote sources, meaning a single `LocalConfigurationService` object can act as an interface to request information dynamically from any remote source. This functionality explains the use of the optional `source` parameter in the API, which makes the object a stateless adapter for interfacing to a remote information source exposing the Configuration Service API.

The following lists the primary strategies for utilising the Configuration Service.

- A list of service access points which expose their local configuration via the Configuration Service API can be registered with a LocalConfigurationService object, which will iterate through them to acquire required information from the first source which can provide it. Copy-on-read can be specified to avoid future requests for the same information being made to remote sources.

- A service exposing the Configuration Service API may provide recursive fetching of information such that when it receives a request for information, rather than simply checking the local in-memory information, it may forward the request to its own set of remote or alternate sources. In this way the service acts as an intermediate proxy for queries.

- An object which requires information from multiple sources can either be

passed the location of the sources "directly" (via local initialisation files, start-up parameters, or hard-coding), or can refer to a Configuration Service which acts as a directory, and returns the location of a particular service given, for example, its canonical name. With this list of sources, the LocalConfigurationService object within the component in question can then fetch necessary information from the relevant remote components directly.

For DIRAC, the central Configuration Service is implemented using an Internet accessible multi-threaded XML-RPC interface, over HTTP. The actual information is contained in a database for persistency, making use of *(section, option, value)* tuples where *(section, option)* form a joint key. This service is supported by a multi-threaded database connection pool which allows a pre-defined number of simultaneous queries to be supported.

During DC04, it was found that using a single central Configuration Service and a set of local INI files for each component allowed the efficient distribution and management of system information. In this way, the central Configuration Service provided system-wide "global" information, such as the location of other services, grid-enabled storage nodes, or configuration parameters. At each site a site-wide configuration file was used, coupled with a component specific configuration file. These three sources (component, site, and global) allowed Clients, Agents, and Services to self-configure and inter-operate, and provided the flexibility to override information if necessary (*e.g.* by utilising a local value in preference to a global value).

Due to the importance of this information-sharing network, it was found that network outages, system failures, and overloading of the central Configuration Service could result in failure of active jobs, therefore to address this several complementary techniques were used to minimise the chance of failed information queries:

- Replication of the central Configuration Service at multiple sites;
- Timeout, retry, and fail-over settings;
- Caching of information to eliminate multiple queries;
- Block queries to return values for an entire section at once;
- "last-ditch" information source from a local INI file.

These combined techniques resulted in a robust grid information infrastructure which could handle the load of thousands of jobs executing simultaneously. Figure

3.7 illustrates the load placed on the Configuration Service during DC04, showing peaks of 300 requests per minute. Load testing showed that the system lost responsiveness when more than 40 requests per second were made.

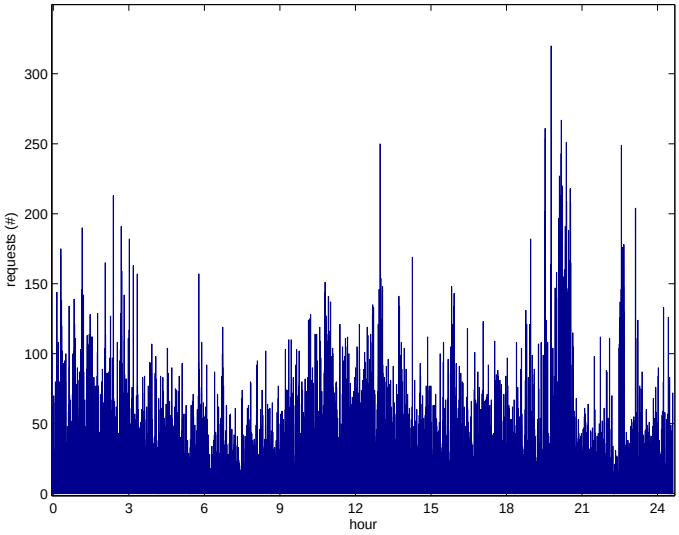

Figure 3.7: *Configuration Service queries per minute over 24 hour period.*

3.5.6 Monitoring and Accounting Services

The *Job Monitoring Service* provides an interface for Agents and Jobs to update job state and for other Services or Users to query job state. This only retains information for jobs which are active in the system. Jobs which have completed or failed are eventually cleared to the *Job Accounting Service*. There are three access modes to the Monitoring Services: an API, a web-interface, and command line tools.

3.5.7 Agent

The Agent handles DIRAC jobs at a computing site by submitting them to the Local Resource Management System (LRMS). The current system supports PBS, LSF, BQS, Sun Grid Engine, Condor, Globus, EDG/LCG, Fork, and In-Process resource types. The job JDL is inspected by the Agent to handle staging and registering of input or output files. The Agent monitors the progress of the job and sends status updates to the Monitoring Service.

The design consists of a set of pluggable Agent Modules. The modules are executed in sequence in a continuous loop. Typically a site runs several agents

each having its own set of modules, for example job management modules or data management modules. This feature makes the DIRAC Agent very flexible, since new functionality can be added easily, and sites can choose which modules they wish to have running. Further details are discussed in Section 3.6.2.

Agents can operate in a *cycle-scavenging* mode at the cluster level, where they only request and execute jobs when the local resources are under-utilised. This idea comes from global computing models, such as SETI@Home, BOINC, and distributed.net [59–61], which perform cycle-scavenging on home PCs.

3.5.8 OGSA and OGSI

The framework proposed by OGSA[49] and specified in detail by OGSI[62] was seen as an opportunity to move toward increased interoperability between grid software components. Several months of intensive work were invested in developing DIRAC Services as OGSI Java components utilising Globus Toolkit 3, however this was abandoned shortly before Globus and IBM jointly announced their intent to discontinue OGSI and in its place proposed WSRF (Web Services Resource Framework)[63].

In principal, the concept of dynamic, stateful, transient Grid Services, as compared to Web Services which are static and stateless, is sound. A common standard for security, lifecycle, service data, and publish/subscribe event notifications are all required, however the realisation of those in OGSI was, in practice, unworkable for the following reasons:

Heavyweight and complex Impossible to develop lightweight clients, difficult to run as a regular user, and significant infrastructure required for deploying Grid Service containers;

Not standards compatible Unable to leverage existing Web Services tools;

Poor documentation For installation, maintenance, debugging, and development;

Poor implementation Many bugs in GT3 and a constant flood of exceptions.

Together these made it difficult to develop, debug, deploy, and maintain OGSI Grid Services. Similar experiences were recorded by others[64–68]. LHCb considered using a pure Python implementation of OGSI, pyGridWare, implemented by Lawrence Berkley Laboratory (USA), but this was not sufficiently complete to be practical. While it is understood that the more recent versions of GT3 and GT4 (for WSRF) have corrected many of the early technical problems, the combined

facts that OGSI had no future, and the complexity of development under GT3 forced development of DIRAC to return to Python and XML-RPC.

While OGSA provided rich mechanisms for service description, interaction, and management, the stability and simplicity of Python and XML-RPC were of greater utility, given the complexity of available OGSA implementations and difficulty in developing and deploying stable Grid Services.

3.6 DIRAC: Key Features and Advances

This section discusses four aspects which have been key to the success of DIRAC: the pull scheduling paradigm, lightweight modular agents, the use of instant messaging, and mechanisms to provide fault tolerance.

3.6.1 Pull Scheduling

DIRAC emphasises *high throughput* rather than *high performance*. This idea is championed by the Condor project[13], from which DIRAC borrows heavily in terms of philosophy for designing generic distributed computational systems[14]. It advocates immediately using computing resources as they become available, rather than attempting global optimisations of all jobs over all executors. In the Condor approach, which we will call the *pull* paradigm, executors request computing tasks by announcing their availability. In contrast a *push* paradigm has a scheduler which monitors the state of all queues and assigns jobs to queues as it wishes.

For *push* scheduling to work, all the information concerning the system needs to be made available at one place and at one time. In a large, federated, grid environment this is often impractical as information may be unavailable, incorrect, or out of date. Even if it is available, job allocation complexity grows quadratically with the number of jobs and resources, where every possible allocation combination must be evaluated to select an optimal schedule. This is a classic NP-complete problem. While there are efficient heuristic approaches that in practise come near to an optimal solution, such algorithms generally require complete and up to date information regarding system state, and are typically designed to operate on homogeneous computing resources with $10^2 - 10^3$ queued jobs. Later in Section 3.8, Figure 3.16 illustrates the poor performance of centralised push scheduling in the LCG environment.

As a result of this, *push* scheduling in a grid environment has proven to be problematic. By contrast, the DIRAC Central Services simply maintain queues

of prioritised jobs (see Section 3.5.3) and allocate the highest priority job which matches an executor's job request. The Condor Matchmaking libraries facilitate dynamic resource definitions, as opposed to the traditional batch system which contain queues consisting of static characteristics[21]. This is the only aspect of Condor which is used directly by DIRAC. While Condor can operate across sites and in a federated fashion, these are not its strengths. It has trouble coping with several thousand active jobs, hundreds or thousands of users, not to mention that the same version of Condor must be installed on all systems by a super user and a single security domain must be maintained[69–73]. This is highlighted by accounts of an eight user and one thousand CPU system being termed a "very large Condor pool" which resulted in a "melt-down" when the single scheduler had 3000 to 4000 queued jobs[69]. Condor has grown out of a distributed batch system with a small set of high availability central services, and has an architecture catered for this environment[73]. LHCb required a decoupled system which could handle tens of thousands of active jobs across hundreds of sites and be very fault tolerant. The DIRAC design attempts to address these requirements.

With a pull paradigm, the previously difficult task of determining where free computing resources exist is now distributed to the remote Agents (see Section 3.5.7) which have an up-to-date view of their sites' state. Since jobs are grouped into queues based on common requirements, the worst case is that each job request from an Agent will be compared against each queue once, where typically the number of queues is much less than the total number of queued jobs.

Both long matching time and the risk of job starvation can be avoided through the use of an appropriate Optimiser to move "best fit", "starving", or "high-priority" jobs to the front of the appropriate queue. As reported elsewhere[74], this allows a mixture of standard and custom scheduling algorithms.

Figure 3.8 shows the match times for jobs during LHCb DC04. 85% of the time this operation takes less than two seconds even with tens of thousands of queued jobs, thousands of running jobs, and dozens of sites requesting jobs concurrently. This is 30 times faster than the mean scheduling time of the LCG Resource Broker in performing essentially the same task. Performance of LCG will be discussed at length in Section 3.8.2, including a comparative histogram of LCG matching time in Figure 3.16.

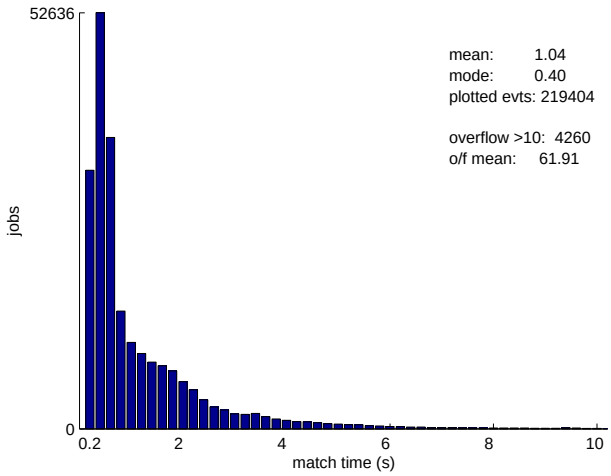

Figure 3.8: *DIRAC match time distribution for 223,000 jobs during DC04*

3.6.2 Lightweight Modular Agents

By providing simple abstractions of *Computing Elements* (CE or *executor*) and *Storage Elements* (SE), and exposing simple APIs to the Core Services, it was possible to implement lightweight Agents (see Section 3.5.7) which can be installed and run entirely in unprivileged user space on any executor. This allows the rapid utilisation of heterogeneous systems in a federated manner — the most general objective of computational grids. Collaborators with normal access to the remote systems simply install a one megabyte self-contained package with all the custom software for the DIRAC Agent.

The configuration allows local policies on queue usage to be applied, and selection of which Agent modules to run. This modularity gives local Agent administrators great flexibility and control, and makes it easy to write custom modules.

The only pre-requisites are a recent version of the Python interpreter and outbound Internet connectivity, in order to contact the DIRAC Services. This allows the Agent to run under virtually any computing and network environment, including behind firewalls and private networks utilising Network Address Translation (NAT) to reach the Internet. Installation entirely in regular user space mitigates the security risks present in software which requires "root" access and system wide installation, and accommodates LHCb members who administer a DIRAC Agent at their local computing centre but do not have privileged access.

3.6.3 Instant Messaging for Grid Services

Instant messaging can provide a mechanism to connect grid components and users in a peer-to-peer fashion, transcending firewall and NAT issues, and providing a level of indirection to the physical site and node the data or processes reside on via a portable instant messaging address. Certain instant messaging infrastructures also provide message buffering, thereby protecting communications from network outages, overloads, and service restarts. These features drew the LHCb grid software group to investigate the potential of using instant messaging in DIRAC.

Instant messaging has since been incorporated into all the DIRAC components: Services, Agents, Jobs and User Interface, providing reliable, asynchronous, light-weight and high speed messaging between components. Public demand for instant messaging has led to highly optimised packages which utilise well defined standards, and are proven to support thousands to tens-of-thousands of simultaneous connections. While these have primarily been for person-to-person communication, it is clear that machine-to-machine and person-to-machine applications are possible, and it is in these areas DIRAC has demonstrated a novel application of the technology.

While XML-RPC was appropriate for DIRAC Services to use to expose their APIs, this protocol is not as well suited for the Agents and Jobs which also must be reachable by the Services, or by Users. In this environment, instant messaging provided a useful access channel. No *a priori* information is available about where or when an Agent or Job will run, and for security reasons local networks often will not allow Agents or Jobs to start an XML-RPC server that is widely accessible. Firewalls are typically configured to block inbound network connections to worker nodes. This suggests a client-initiated dynamic and asynchronous communications framework is required.

The Extensible Messaging and Presence Protocol (XMPP), now an IETF Internet Draft[75], is currently used in DIRAC. This has grown out of the open-source, non-proprietary, XML based Jabber instant messaging standard[76]. XMPP provides standard instant messaging functionality, such as one-to-one messaging, group messaging ("chat"), and broadcast messaging. An RPC-like mechanism exists called Information/Query, (IQ) which can be used to expose an API to any XMPP entity. The *roster* mechanism facilitates automatic, real-time monitoring of XMPP entities via their *presence*.

The DIRAC Services use XMPP in places where fault tolerant, asynchronous messaging is important. For example, the Job Receiver Service uses XMPP to notify the Optimiser Service when it receives a new job. When the Optimiser gets

this message, it will then sort the new job into the appropriate queues. The IQ functionality has the potential to allow users to retrieve real-time information about running jobs, something which is critical for interactive tasks, or for job steering. It also greatly facilitates debugging and possible recovery of stuck jobs.

XMPP is specifically designed to have extremely lightweight clients, and gracefully handles dynamic availability of entities, buffering all messages until an entity is available to retrieve them. By matching the XMPP IQ functionality to standard XMPP messages, it is possible for users with a standard XMPP client to locate and communicate with Agents, Jobs and Services from anywhere. This has already been put to good use in DC04 for controlling and monitoring the state of Agents.

The basic structure of XMPP consists of Servers, Clients, Users, Entities, and Connections. As mentioned earlier, XMPP utilises "thin-clients", leaving all long-term state information to the server. In this sense it is not truly a peer-to-peer system, but a multi-hub and spoke instant messaging server. The Client manages the connection to a Server, implements the client side protocol handling, and stores any state information regarding the current session. A user (identified by an email-like URI (*e.g.* `xmpp://ijstokes@dirac.cern.ch`) can connect multiple times to a Server, with each Connection creating a separate XMPP Entity, uniquely identified by a Resource Name (*e.g.* `home, office, laptop`). The unique Entity end-point name is formed by concatenating the Resource Name onto the User Name (*e.g.* `ijstokes@dirac.cern.ch/home, ijstokes@dirac.cern.ch/office, ijstokes@dirac.cern.ch/laptop`). In this way a single authentication credential can be used to create multiple simultaneous Entities, each with a unique address, possibly existing at different network locations.

Once the Client negotiates an Entity Connection to a Server, the Server will provide the Client with all user account information stored by the Server. This primarily consists of the user's roster, which is an address list of other XMPP users (messaging end-points). A key difference between the roster and a typical address list, as found in electronic mail systems, is the addition of dynamic state information for each entry. This information describes what state each user is in (*e.g.* "online", "offline", "away", "do-not-disturb", etc.), and a customisable status field. The roster acts as a publish/subscribe mechanism, where any changes in the local user's state is broadcast to all online roster members, and any changes in the state of a remote user who is on the roster list is sent to the local user. XMPP roster management and messages are handled through `<presence>` messages.

The original application of Instant Messaging within DIRAC was to provide asyn-

chronous, buffered messaging between Services. Each Service Class was assigned a single authentication credential (User account), and each Service instance utilised a unique Resource name to provide a single unique address for that instance. This mechanism decoupled Services from each other and allowed them to be stopped, restarted, and even moved to different hosts during live operation. This was critical for robustness of the overall system and maintenance of individual services. Given the number of Services was small (5-20), and the communication between the Services limited, there were no problems with bottlenecks at the central XMPP server seen with this approach.

The next step was to introduce Instant Messaging for state monitoring of Agents. By using the "chat room" functionality of Instant Messaging, an *ad hoc* messaging hub could be created. Agents could connect to an "Agent Chat Room" and publish progress information as chat room messages, and the room roster list acted as an inventory of online Agents. By using custom status fields the roster also provided information regarding where the Agent was running, including host name, directory, and process number. This was critical when misbehaving Agents were discovered. Again, given the number of Agents was initially low (10-100), this operated well and allowed a system administrator to use a standard XMPP GUI Client to connect to the same Chat Room and monitor Agent status.

This naturally led to the question of introducing Instant Messaging to each Job. The intention was to provide job-level live monitoring. Due to the fact that the number of active jobs was two orders of magnitude greater than the number of Agents (1000-10,000 active jobs), it was discovered that thousands of automated Instant Messaging clients connecting to a single XMPP server or chat room resulted in a Distributed Denial of Service (DDoS) attack. The messaging load saturated the network connection, caused the server to consume all available memory, and overloaded the server processor. The XMPP server software was running on the same server as the other DIRAC software services and therefore paralysed the entire system. On the Client side, the XMPP connection was not done in a separate process or thread, so the blocking resulted in stalled processes.

This experience indicated that the XMPP server had to be independent or sandboxed so as to not overwhelm other services on the same host, and that any DIRAC components making use of XMPP client-side connections had to do so in a non-blocking manner – that is, either in a separate thread or fork, and with appropriate timeouts. A memory leak in the JabberD2 server software meant that above 1000 simultaneous connections the process would grow exponentially in size and eventu-

ally crash. This has since been fixed, but at the time the integration of DIRAC with the LHC Computing Grid made it necessary to operate one Agent per Job, meaning that even limiting Instant Messaging usage to Services and Agents would result in excessive connections.

Within the development branch of DIRAC, another application of interest has been implemented. This is to make use of the XMPP `<iq>` messages to provide RPC functionality through to Agents and Jobs, allowing them to be remotely controlled, and to provide access to data local to a Job. The initial implementation provides just basic process control and small file transfer, however in principle a much richer level of RPC interactivity is possible. Utilisation of the standard `<message>` messages would also allow interaction with Services, Agents, and Jobs using standard Jabber/XMPP GUI clients, however this has not yet been investigated in depth.

The two main outstanding issues for the use of Instant Messaging are the security and authentication implications of a "tunnelled" control channel into remote computing sites, and the scalability to tens of thousands of XMPP entities communicating across the same instant messaging network. For the first, a group at Lawrence Berkley Laboratory (USA), have developed an XMPP server which accepts Globus X.509 GSI certificates for authentication. This is a positive step and collaborative work is underway to investigate how this can be used for end-to-end security and authentication, however this has not yet been released for general use. For the second, many high performance commercial XMPP servers are available, however the freely available open source servers still demonstrate some robustness and scalability issues which have created problems when XMPP is used heavily, for example by thousands of jobs broadcasting status updates (see Section 3.8). Existing XMPP clients and servers have limited support for digital certificates, therefore it is necessary to either create a new account for every autonomous client (*i.e.* XMPP clients representing Jobs or Agents), or for insecure accounts to share passwords in plain text. The latter is undesirable but not unthinkable in a purely monitoring environment, where it is conceivable a monitoring mechanism with no authentication could be acceptable. The former is equivalent to trusting the secure placement of proxy certificates on remote nodes, provided the temporary account and insecure shared passwords have equivalent lifetimes to proxy certificates. In any case, a better mechanism is certainly the implementation of end-to-end security, probably utilising some form of PKI. A current Jabber extension proposes exactly this[77], however it has yet to be widely implemented. Some commercial Jabber client/server environments do support the use of X.509 digital certificates for client-server and

client-client authentication, however these were not available for our work.

3.6.4 Fault Tolerance

In a distributed computing environment it is impossible to assume that the network, remote storage, and remote services will constantly be available. The result is that any remote operation may fail in one of three ways: failure to connect to the remote resource; stall during the remote operation; or exit with a failure.

These failures are often not permanent, so a retry at a later time or to an alternate equivalent resource may be successful and allow the parent operation to complete, albeit with a delay incurred due to the retry. In order to cope with these failure modes the following mechanisms were used in DIRAC:

Retry Many commands retry with a time delay in order to overcome any network outages, service request saturation, or service failure and restart.

Replication Numerous services have a duplicate backup service available at all times.

Fail-Over When contacting critical services, after the retry limit is reached, a request to an alternate service is attempted.

Caching In the Local Configuration Service, the remotely fetched data can be cached locally for future retrieval.

Watchdog Monitors components to ensure continuous availability and restart on failures.

All Services and Agents are run under the *runit* watchdog[78]. It ensures that the component will be restarted if it fails, or if the machine reboots. It also has advanced process management features which limit memory consumption and file handles, so one service cannot incapacitate an entire system. Automatic time-stamping and rotation of log files facilitates debugging, and components can be paused, restarted, or temporarily disabled. Furthermore, none of this requires privileged access to setup.

3.7 LHCb 2004 Data Challenge

DIRAC has been developed by a core team of four developers, with extensive input, contributions, testing, and deployment feedback from the LHCb Data Management

Group, and computing centre administrators. It has aimed to bridge the computing requirements of LHCb with the capabilities available at the collaborating computing centres, and to provide a basis for evaluating grid computing approaches, particularly the functionality offered by the LCG environment.

3.7.1 Historical Background

The initial system developed in 2002-2003 was exclusively for performing LHCb physics simulation where Agents pulled simulation parameters from a database[79]. For 2003-2004 the emphasis shifted to providing a generic computational grid system which could incorporate new developments at that time around the *Open Grid Services Architecture* (OGSA)[49], and the Globus Toolkit 3 (GT3) implementation of the *Open Grid Services Infrastructure* (OGSI)[62].

At the same time the EDG project[43] was in the process of delivering the software from its three year development phase to the LCG project[46] which was meant to deploy and stabilise the EDG software. Due to performance issues with the EDG software, a refactoring of the EDG architecture was proposed under the auspices of the ARDA-RTAG (Architecture Road-map for Distributed Analysis — Requirement Technical Assessment Group)[47]. This refactoring was to take the form of a service decomposition with clearly specified interfaces. This was done to improve the conceptual organisation of the architecture, decrease dependencies between components, facilitate the incorporation of new services, and make possible the substitution of customised or alternative services.

The early version of DIRAC had already followed a service oriented architecture, so it was hoped this could then be refactored into a Python based set of OGSI Grid Services, implementing the ARDA-defined interfaces. The ARDA refactoring proposal was then handed over to the EGEE (Enabling Grids for E-science in Europe) project[48], which is the successor to the EDG project, in order to implement a prototype. Rather than specify service interfaces and utilise short release cycles for the new ARDA/EGEE grid services in order to gain rapid feedback, EGEE focused on the implementation of the gLite service oriented architecture. This approach made it impossible for LHCb to collaborate or incrementally integrate gLite into (or in place of) DIRAC. LHCb was thus left to integrate DIRAC with the LCG software and implement a service oriented architecture "in the spirit of ARDA", providing independent, simple services which fit into the core aspects of ARDA required immediately by LHCb for DC04.

3.7.2 Experience

The DIRAC system has been used for the LHCb Data Challenge 2004 (DC04), held from May to October 2004. DC04 had three goals: to validate the LHCb distributed computing model based on the combined use of LCG and conventional computing centres, to verify LHCb physics software, and to generate simulation data for analysis. 300,000 jobs were run, consuming over 670 processor-years of CPU power, and producing 98 terabytes of data. This data was redistributed across the centres for both organised (*i.e.* planned and predictable) and chaotic analysis of the results.

The system operated smoothly with a sustained level of over 3500 running jobs, and 600 gigabytes of data generated and replicated daily. Figure 3.5 shows the participating sites, and Figure 3.10 shows a snapshot of the running job distribution. Once installed, the DIRAC Agents ran autonomously and restarted after failures or reboots. A central team watched the monitoring system and alerted site administrators when problems were detected with the local resources. As shown in Figure 3.9, the jobs were CPU-bound therefore any jobs which did not seem to be executing were almost certainly stalled either due to an internal bug, a node or site problem, or a network problem connecting to central services or network storage elements.

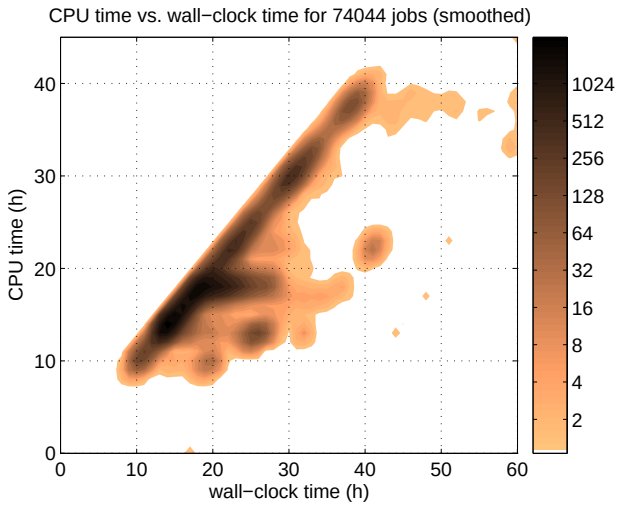

Figure 3.9: *CPU time vs. wall-clock time, showing near-100% CPU Utilisation with only a small number of jobs exhibiting delays due to stalled or blocked transfers.*

The twenty sites with direct involvement in LHCb varied enormously in size, from 20 CPU clusters shared heavily with other users to large 500+ CPU clusters

dedicated to LHCb. A mailing list and weekly phone conferences allowed the DIRAC software developers, site administrators, and data challenge managers to discuss progress and solve problems.

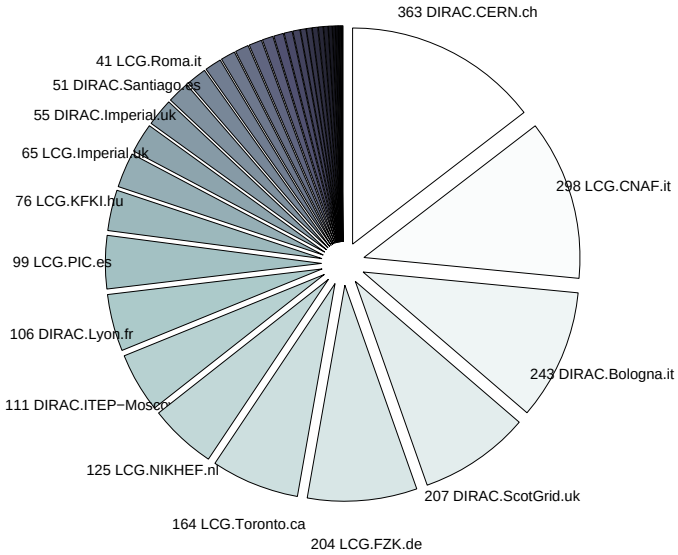

Figure 3.10: *Representative snapshot of running jobs per site during DC04. Notice a mixture of "standard" sites (prefixed DIRAC), and LCG grid sites. Numbers indicate running jobs at that site.*

Another 40 sites were accessed via LCG, and are discussed in Section 3.8. In total, these 40 sites provided almost 3000 worker nodes. For example, more than 40,000 jobs were completed in the month of May with an average duration of 20 hours, running on average at 93% load, the remaining 7% being I/O operations or waiting for external resources to come available. Each job produced on average 400 megabytes, which was replicated to several sites for redundancy and to facilitate later data analysis.

3.7.3 Faults and Major System Failures

There were four major outages in the DIRAC core services availability which resulted in jobs failing, jobs stalling, or sites failing to get new jobs:

Deleted Database One of the high availability core servers, which is monitored 24 hours a day by CERN IT staff, reached 90% full on the local hard drive. This was due to a large and very actively used database on the server. The IT staff intervened by stopping MySQL and deleting part of the database, resulting in the loss of all queued jobs.

Distributed Denial of Service Early efforts to incorporate instant messaging into all aspects of DIRAC resulted in very effective distributed denial of service attack on the server hosting DIRAC and the instant messaging hub. Thousands of jobs were simultaneously sending status information, and in many cases were (unnecessarily) sharing this information with each other, resulting in an extremely high, and unmanageable, message volume which compromised the performance of other services running on the same server.

Network Failure CERN experienced a site wide network failure for approximately one day due to efforts required to isolate an internally compromised machine. All services were unavailable during this time, and it was proposed that a fail-over system be prepared at an external site. This was not completed due to the infrequency of extended total network failure at CERN and the effort required to configure and manage a second DIRAC system.

MySQL Connection Limit MySQL has a default limit of 100 simultaneous database connections. The multi-threaded XML-RPC Services take their DB connection handles from a connection pool local to each Service. If the pool is emptied, the Service creates more connection handles. At a point of high load, due to the ramp up of LCG sites in DC04, one Service repeatedly emptied its connection pool and claimed all 100 available connections, thereby blocking all other Services from communication with the DB. Until the MySQL connection limit was increased and the Services set a pool limit, the DIRAC Services were effectively unavailable. This occurred over the weekend and resulted in a day of lost job matching, although running jobs continued.

Only a few significant bugs were identified in the DIRAC software. These appeared early in the data challenge and were quickly resolved. They generally centred around service scalability and availability, requiring the implementation of operation buffering, timeouts, and fail-over mechanisms on both the client and server sides (see Section 3.6.4). Once these early bugs were resolved, "classic" computing sites (*i.e.* batch queues on computing clusters with local administrators) observed stable performance. Experience with LCG is discussed in Section 3.8.

The small size of DIRAC, buffering of transfer requests, use of a local job database, and independence from the local batch system, all meant that it was possible to stop the Agent, even while jobs were still running on the site, perform a software update, and restart the Agent without loosing existing jobs or transfers. This greatly facilitated rapid resolution of bugs, and was even extended to a prototype Update

Agent module which would perform automatic Agent software updates. These aspects supported Key Goals 2 and 5, Reliability and Manageability. By keeping DIRAC small (clients consisted of a 1 MB archive) it was possible to easily understand how it operated, configure it, and make any changes to correct problems. It also left a small footprint and was quick and easy to install.

Data Management presented the greatest overall challenge. A number of sites experienced significant data transfer delays or failures, resulting in transfer backlogs. Large sites would quickly fill their queues with hundreds of jobs, producing, for example, 40 GB of data and all finishing at approximately the same time, therefore saturating the site's outbound bandwidth or the target Storage Element's inbound bandwidth. Although DIRAC supports a wide range of transfer protocols (see Section 3.5.4), difficulties in using every one of these were encountered at some point during DC04. In particular we note the lack of a simple user-level installation of a grid-ftp client as a major stumbling block towards its adoption. From a global view the system has difficulty in identifying fatally failed transfers (*i.e.* those that will never be retried) and transfers which are outstanding but queued. There are plans to use the XMPP interface to Agents to resolve this. The Transfer Request mechanism performed well and eventually flushed data backlogs.

It has always been identified that managing the volume of data produced by the LHC experiments would be one of the greatest challenges. LHCb made use of multiple File Catalogues to provide a degree of redundancy in registering files which are created and exported to the two primary data stores: CERN's dCache, and the LCG Storage Elements. This proved crucial in identifying errors based on inconsistencies between the two catalogues. It also became clear during DC04 that a decoupling of data staging and task execution was required. Queuing data transfers into worker nodes in advance of task commencement and then allowing tasks to complete before final data transfer/registration from the worker node onto the grid file system would have alleviated many jobs which failed only at the final step due to their inability to export the simulation data they generated before their reservation time expired. The difficulty of the flat and effectively un-searchable file naming URIs used within LCG Storage Elements also made it difficult to locate physical files when doing "post-mortem" analysis of tasks with failed or erroneous File Catalogue entries. These experiences suggest a more developed task management model is required, including "pipelined" data staging and execution, decoupling of data staging and execution, and a structured hierarchical grid file naming strategy.

3.7.4 LCG Integration

LCG is required to make possible the storage and processing of the vast quantities of data produced by the LHC experiments. It brings together dozens of computing centres around the world and eventually will provide an aggregated computing power equivalent to over 140,000 of today's fastest processors. One of the broad objectives for DIRAC is to provide a smooth transition from cluster-based to grid-based computing for the LHCb experiment and to integrate LHCb computing with the LCG resources.

Initially DIRAC considered LCG as "just another batch system", with the LCG Resource Broker (RB) subsequently submitting jobs to grid nodes. This revealed a number of problems due to instabilities in the Resource Broker, missing software or faults in the site configuration. This made it necessary to find a better way to approach LCG. Furthermore, the extremely high level of failures (close to 50%) for jobs submitted to LCG required that special checking be performed to increase the likelihood of successful job completion. Figure 3.11 shows the jobs submitted to LCG and a breakdown of the outcome of those jobs.

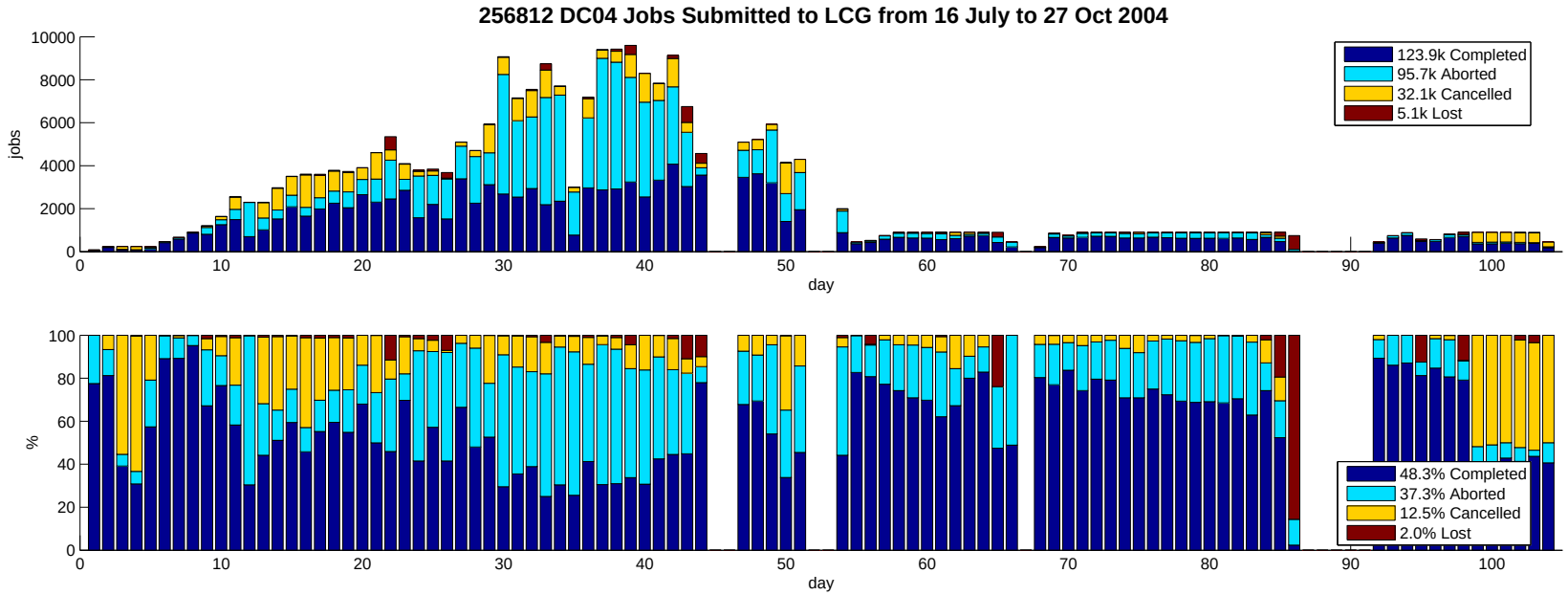

Figure 3.11: *Results of jobs submitted to LCG during DC04*

The next option was to access the LCG Computing Elements directly, bypassing the Resource Broker step. This would require the use of Globus commands. This was impractical both for technical reasons (the Globus API was too low level, so implementation was difficult), and policy reasons (LCG Resource Brokers are the basis of fair share and usage accounting).

The approach that was eventually utilised to good success was to submit DIRAC Agent installer jobs to LCG. These were called pilot jobs, and utilise an approach similar to Condor Glide-In[80]. The pilot jobs would specify generic characteristics of a typical LHCb job in order to be matched appropriately. Once they started to be executed on a grid node they would perform a set of checks to determine available storage space, node performance (via a short benchmark), memory, queue limit, and proxy lifetime. If these tests suggested an LHCb job could be run, then the DIRAC Agent would be installed in a "local run-once" mode via the LCG RB. The Agent would request a DIRAC job, taking into account a combination of the properties specified with the "Agent installer" LCG job and the real-time characteristics of the node on which the Agent was executing. This mode was designed to fetch at most one LHCb job from the DIRAC Services, run it to completion, and then transfer the output data to a DIRAC SE. All the retry benefits of the Agent were in place, and the testing, installation and auto-configuration procedure took less than a minute.

Figure 3.12 shows the transition from traditional batch systems at computing centres to utilisation of the LCG grid-based job execution. This will be discussed further in the following section. Overall, 54% of DC04 jobs have been completed using LCG.

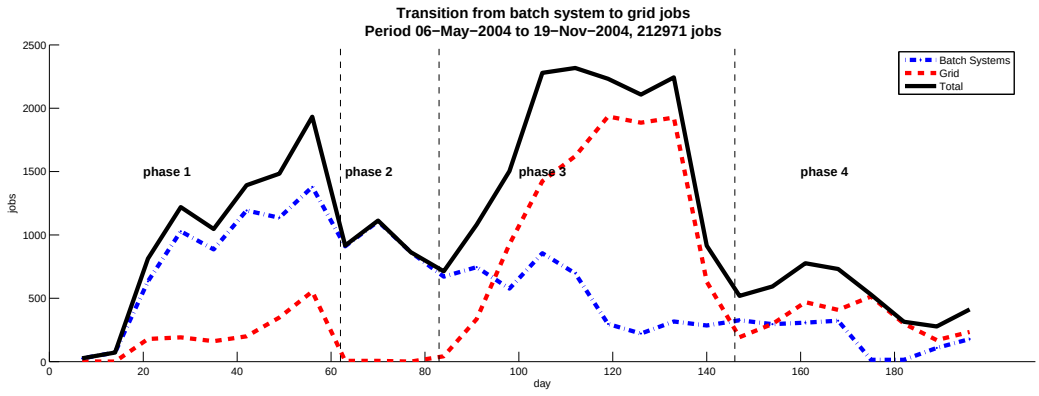

Figure 3.12: *Transition from classic batch systems to grid-based job execution (successful jobs only) (smoothed).*

3.7.5 Development and Deployment Environment

DIRAC development has been supervised by the LHCb Production and Analysis Software Group through weekly phone conferences and mailing list correspondence. This group represents the LHCb stakeholders of DIRAC who require the system to meet the experiment's distributed computing goals. This group advises on development priorities and provides feedback on operational issues. Members of this group also input to the LCG planning, and therefore are able share the LHCb experience of DIRAC and LCG with the LCG management team.

The DIRAC team has utilised CVS and Savannah for software management. Both have proved to be invaluable. Savannah, which is a branch from the popular SourceForge project management environment, is available at CERN and integrated with the CERN CVS repository, and CERN AFS file system. Both users and developers have made extensive use of Savannah for bug reporting, task prioritisation, support requests, software documentation, and software releases. Use of Savannah also allows easy migration of bugs from DIRAC to other software groups, such as LCG or physics software teams.

The Core Services for the initial test deployment were installed on two servers at the Centre de Physique des Particules de Marseille (CPPM). When the production deployment was installed on high availability servers at CERN, the test deployment became an emergency fail-over system and also moved to CERN. At times during development the services were spread across servers in Oxford, Marseille, and CERN, demonstrating the effective distribution of a single DIRAC "installation" with different services installed at different sites. Batch system integration was developed using PBS on the Oxford Physics cluster, Condor and Globus using the Oxford e-Science Centre NGS clusters, BQS at the IN2P3 Computing Centre in Lyon, and LSF at CERN.

3.8 DC04 Performance Results

This Section presents detailed results of the performance of the LHCb applications, DIRAC, LCG grid sites and nodes, and the central LCG services. During DC04 there were four distinct phases to the utilisation of the LCG. These are shown in Figure 3.12 and described below.

Phase 1 Here the DIRAC interface to LCG was exercised for the first time. After a month of operation it was agreed to halt using LCG in order to resolve a set

of significant issues which had been discovered with LCG, surrounding loss of data, stalled jobs, failed jobs, and lost jobs.

Phase 2 During this phase LHCb worked closely with the LCG deployment team to test new developments in the LCG architecture which aimed at resolving the issues identified in Phase 1. No "real" jobs were submitted, but a test system was heavily utilised.

Phase 3 With the major issues resolved, it was then possible to significantly ramp up grid utilisation. During this period LCG was regularly saturated with LHCb jobs. LHCb computing sites which also ran LCG software transitioned their resources to LCG, thus realising one of the key goals of grid computing: allowing the management of computing tasks to be delegated to remote users who are the task owners.

Phase 4 This phase aimed to provide the physics simulations required by LHCb, rather than focus on exercising the grid computing infrastructure. It followed a short break to evaluate the system and then operated at a low but constant level for several months.

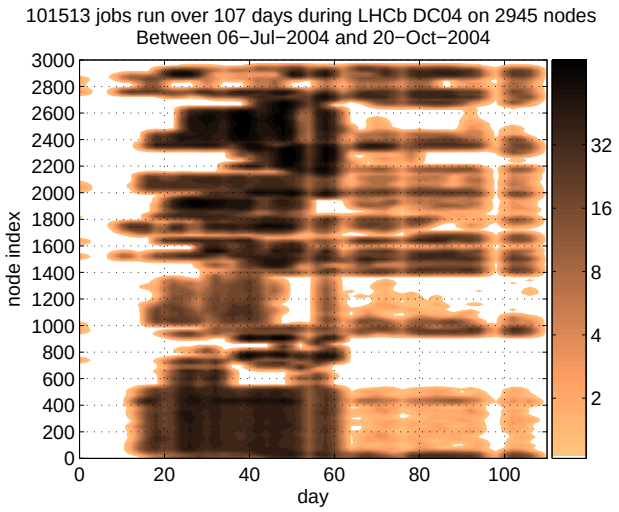

Figure 3.13: *Overview of LCG (grid) job density during DC04. This illustrates where the jobs were running and when. Grouped by site then country. Darkness represents higher job density.*

Figure 3.13 illustrates every node at every site over the duration of DC04, with the shade representing the number of jobs being run at that time. The dark region

at the bottom left represents CERN which, as the centre of LCG, provided the largest and most consistent volume of grid resources. It is possible to see both sites drop out (white squares) and periods when either DIRAC was not providing jobs (the job pools had run dry) or submission to LCG had been stopped, which appear as vertical white bars.

Daily LCG submission rate and outcome is shown in the histogram in Figure 3.11. The largest cause of aborted jobs was due to hitting the retry limit (usually set to 3). Under certain circumstances if a job failed it would be returned to the LCG Resource Broker and resubmitted to a new site. It was found to be the case that if a job failed once it was unlikely to succeed on subsequent automated resubmission. More detailed discussion of LCG performance can be found in Section 3.8.2.

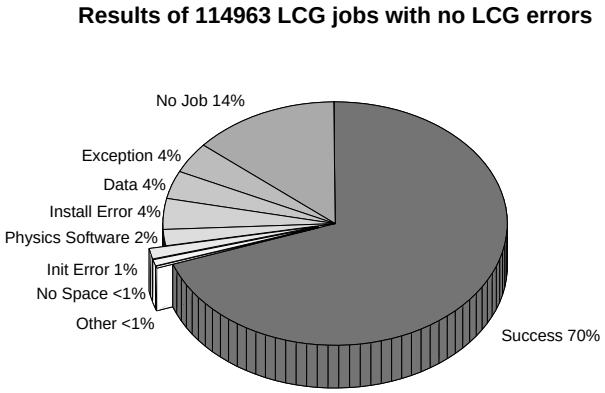

Figure 3.14: *Results of DC04 jobs which had no LCG problems.*

Jobs which executed successfully from an LCG point of view still may have had errors due to either the LHCb grid infrastructure, the physics software, or the data management. The distribution of these results is shown in Figure 3.14. Discounting the "No Job" condition, there was a 16% error rate over all LHCb jobs which began executing on LCG resources. "No Job" occurred when the DIRAC job pools were empty and a DIRAC pilot job running on an LCG node made a DIRAC job request which could not be fulfilled.

No Job (14%): This indicates that when LCG started an LHCb pilot job no DC04 work was available;

Exception (4%): The LHCb grid software or services failed, either due to a bug

or service unavailability;

Data (4%): The LHCb job executed to completion but failed either during data transfer or data registration;

Install Error (4%): The LHCb grid software failed to install the necessary physics software to execute the job. This was often due to lack of available disk space;

Physics Software (2%): The physics software itself reported an error;

Init Error (1%): The LHCb grid software failed to self-configure properly;

No Space (<1%): The worker node ran out of drive space during execution of the job.

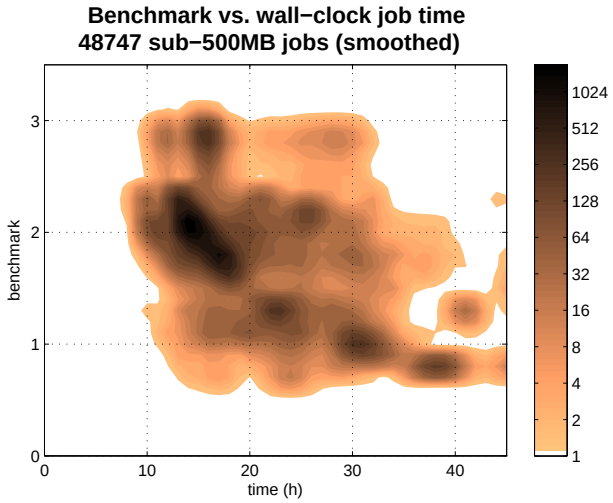

Figure 3.15: *LHCb benchmark vs. wall-clock time to completion for jobs producing 400MB data. This shows a near linear match for the majority of jobs, validating the benchmark prediction.*

As described earlier, LHCb jobs submitted to LCG performed a quick check of node suitability before commencing to execute. Part of that check consisted of a performance benchmark. Figure 3.15 shows the good correlation between high benchmark and low overall execution time.

3.8.1 Challenges presented by LCG

The following are the primary challenges LHCb has faced in trying to use LCG for DC04 and during integration with DIRAC:

Computing power and queue normalisation Job submissions to LCG include a job duration. This only makes sense in some sort of normalised time units (for CPU-bound jobs), however there was no good mechanism to do this. Some sites internally normalised their queue times, resulting in LHCb jobs running with less time than expected. Some sites overload the processors or use Hyper-Threading, similarly throwing off the expected execution time. This resulted in thousands of jobs being terminated by unexpectedly short queue time limits.

Job working space LHCb jobs required 500 MB to 1.5 GB of working space, on top of 2 GB of installed software. NFS mounted job working directories resulted in overwhelming NFS. Other sites did not provide sufficient space for LHCb jobs to complete. Even cases of dual CPU nodes with local job space were problematic, as two jobs could arrive close together, both see sufficient available storage space for job output at the start of execution but data creation by both jobs could exceed this and both jobs subsequently fail for lack of drive space.

Availability of output files after failure LCG jobs run in a "sandbox" which is erased when a job completes, is cancelled, or aborts. Output files are often required to diagnose the cause of failures, but these are not available if the job is cancelled or aborted. LCG only provides limited insight into the cause for an aborted job, with labels such as "Retry Limit", "Count Not Plan", "Proxy Expired", and "No JDL".

Security certificates The X.509 GSI certificates do not allow sharing of job details with others within the VO. This makes collaborative work difficult. There were also bugs found in the handling of proxy certificates which would result in the use of almost-expired certificates in preference over new proxy certificates and the subsequent failure of jobs. Jobs classified as "cancelled" in Figure 3.11 have been stopped manually just before their certificate is due to expire in order to avoid an LCG bug of re-submitting failed jobs with expired certificates.

Operating on large volumes of jobs None of the LCG tools are responsive or easy to use with large numbers of jobs. During DC04 hundreds or thousands of jobs would be submitted to LCG each day. Monitoring and managing these jobs was extremely difficult. Efforts are now underway to address this.

Poor job scheduling It was difficult to evenly spread DC04 jobs across the available LCG resources due to problems with the Ranking algorithm and Esti-

mated Response Time values. At times all jobs submitted to LCG would end up queued at a single overloaded site.

Lack of API documentation The LCG command line tools are largely written in Python, the same language as DIRAC. It would be preferable for DIRAC to use the same APIs as utilised in these tools, however the lack of useful documentation made this difficult.

3.8.2 LCG Performance

The 123,000 jobs submitted to LCG provide a rich set of performance metrics, based on the logging of 46 details for every job, as shown in Table 3.1. The relatively homogeneous nature of the LHCb jobs also facilitate consistent performance analysis. The efficiency of the DIRAC system allowed the saturation of all available LCG computing resources, usually with 90-99% of all executing jobs at a grid site being LHCb DC04 jobs due to under-utilisation by other experiments and virtual organisations. Between days 20 and 60 of DC04 Figure 3.11 is representative of the state of the entire LCG.

Node Details	**Job Details**	**Flags**
Site name	LCG job ID	Install DIRAC
Worker node name	DIRAC job ID	No Work flag
Number of CPUs	Pilot timestamp	Not Allowed flag
CPU type	Execute timestamp	Got Job flag
CPU speed	Transfer timestamp	Rescheduled flag
RAM	Transfer time	
Benchmark	Transfer size	**Resource Broker**
	Data registration time	**Timestamps**
DIRAC Timestamps	Log transfer error flag	Submit
Submit	Data transfer error flag	Wait
Ready	Data registration error flag	Ready
Wait	Load average	Schedule
Match	Memory usage (average)	Run
Schedule	Cache usage (average)	Finished
Queue	CPU Time	Clear
Run	Execution Time	Abort
Done execution		Cancel
Done data transfer		

Table 3.1: *The 46 characteristics recorded for each job in DC04.*

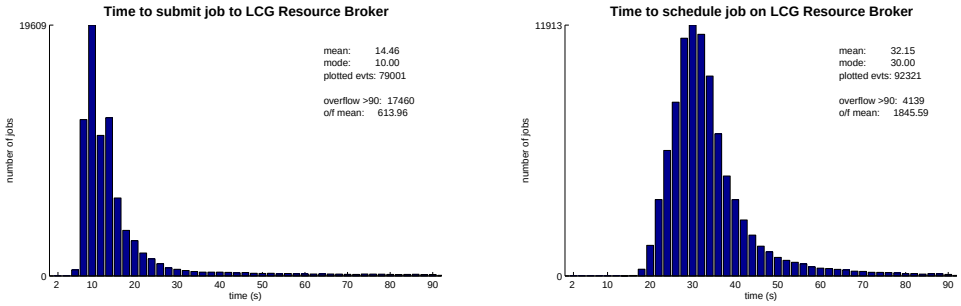

Figure 3.16: *Distribution of job submit and schedule time of LCG Resource Broker for 100k jobs during DC04*

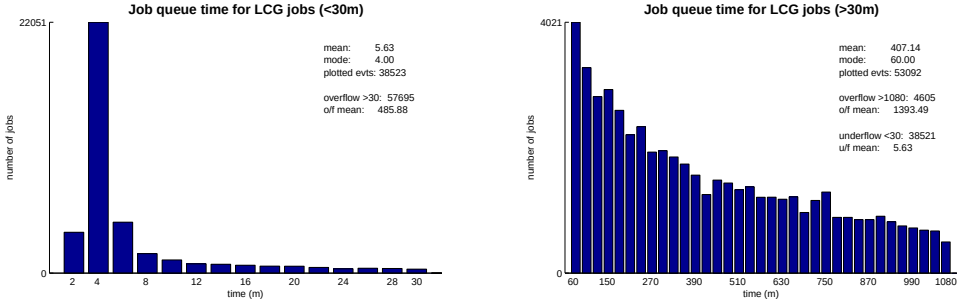

Figure 3.17: *Distribution of job queue time at site once scheduled.*

Scheduling and Queuing

With the rich logging information collected by the DIRAC Services, the Agents, the jobs themselves, and the LCG interfacing software it is possible to reconstruct a good picture of the state of the grid during LHCb DC04. Figure 3.16 shows the performance of the LCG Resource Broker for processing job submissions and doing job matching. Figure 3.17 shows the distribution of job waiting times, which is the time between a job being queued and its arrival at a worker node to commence execution.

Jobs were submitted in batches, usually 500-2000 at a time. The 12 second average submission time shown in Figure 3.16 puts a daily limit of 7200 jobs sent to the Resource Broker. The 31 second scheduling time puts a daily limit of 3000 jobs scheduled by a Resource Broker. LCG addressed these limitations by deploying multiple RBs, however this complicated job management as jobs managed by one

RB were not visible from other RBs.

Job Results

Figure 3.11 shows the daily LCG submitted job results, which are described in more detail below.

Completed (48%): These jobs ran to completion on LCG.

Lost (2%): A small portion of all submitted jobs were lost without record of the result.

Cancelled (12%): Jobs were cancelled by an LHCb operator if they were not scheduled by LCG within 24-36 hours. This was to avoid the job being run with an expired proxy certificate.

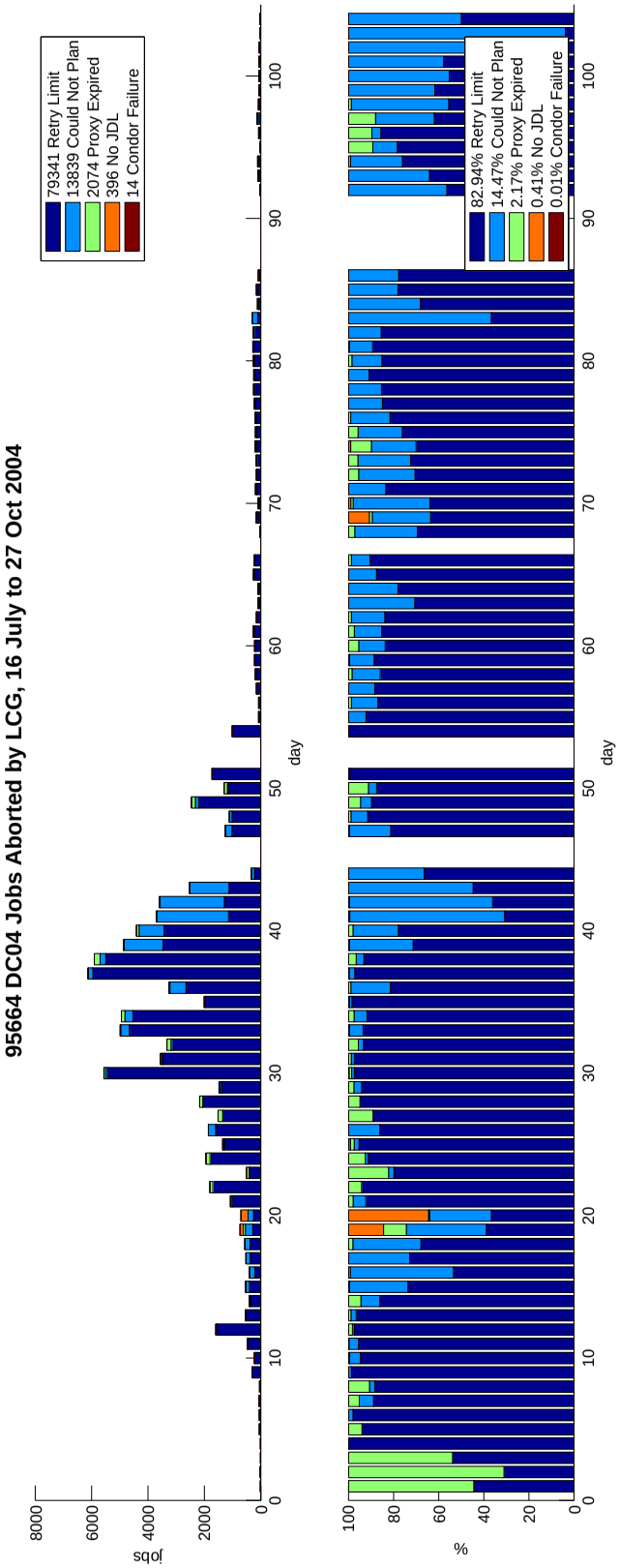

Figure 3.18: LCG distribution of causes of aborted job

Aborted (37%): There were five types of LCG-internal errors which resulted in jobs being aborted. The distribution of these is illustrated in Figure 3.18 and described below:

Retry Limit (83%): Jobs which failed to successfully transit from LCG to a Worker Node to begin execution would be resubmitted. Typically the cause of the fault was within LCG itself, therefore the fault recurred on subsequent retries until the limit was reached.

Could Not Plan (14%): The Resource Broker encountered an error resulting in an inability to schedule the job to any site.

Proxy Expired (2%): By the time the job commenced the proxy certificate used to submit it had expired leaving the executor with no authorization to proceed. This low error rate conceals the fact that 12.5% of all submitted jobs were cancelled in order to avoid this situation. A job whose proxy expires in mid-execution will typically be unable to register or transfer any data or to report any results, therefore this is a critical error.

No JDL (0.41%): This error was concentrated on three days during DC04, and corresponded to a failure in the Resource Broker where it lost access to the original JDL.

Condor Failure (0.01%): This error only occurred 14 times during all of DC04 and was attributed to a failure of the LCG interaction with the Condor scheduler, or a failure of the Condor scheduler itself.

Unfortunately, due to the limited logging information supplied by LCG, the 37% of tasks which were aborted by LCG cannot easily be diagnosed, beyond the five failure categories listed above. In particular, the common "Retry Limit" failure, which occurred in 30% of all submitted jobs, cannot be diagnosed further. From working with the LCG Resource Broker developers it was reported that this would often occur when the Resource Broker contained either corrupted or out of date information regarding available computing resources, resulting in assignment of a task to a computing resource which was either unavailable or unable to accept the task when the Resource Broker attempted to route it there. Due to a quirk in the Resource Broker allocation strategy, when this routing failed it would restart the task allocation process from scratch, usually making the same decision and consequently resulting in the same failure, until the submission retry limit was reached. The other source of "Retry Limit" failures was a situation which would arise where the submitted job description (a JDL file) would be lost on the Resource Broker, leading to a failure at a later stage in the task management process.

Worker Node Characteristics

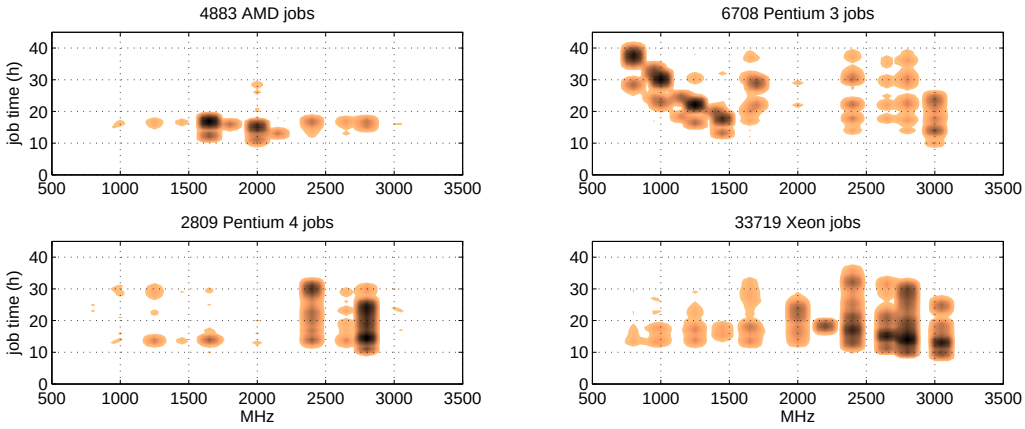

Figure 3.19: *Job CPU time vs. CPU speed sorted by processor type. Only jobs which produced 500MB of output or less are shown. Illustrates variation in CPU processing time for different processor types and speeds.*

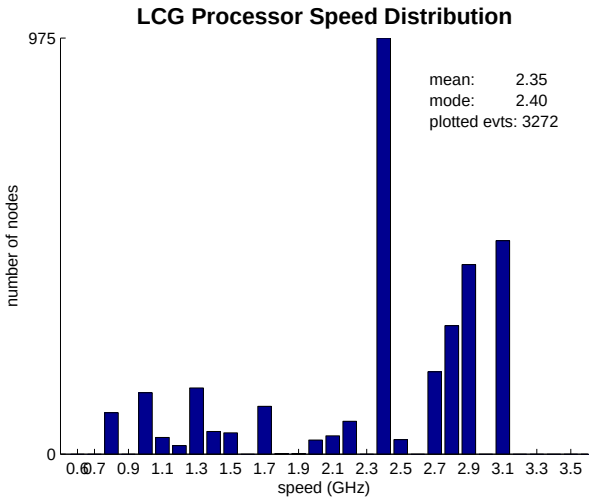

Figure 3.20: *CPU Speed of LCG nodes used by DC04 jobs.*

DC04 provided an opportunity to observe the degree of heterogeneity of computing systems within a grid environment. Figure 3.19 illustrates the execution time vs. processor speed for four different classes of processor (AMD, Pentium 3, Pentium 4, and Xeon). This covers all LCG jobs which completed successfully and is a picture of what the jobs "saw" rather than a description of the grid itself. Hyper Threading, or processor overloading (more than one active job on the processor) was difficult or impossible to identify at run-time, however this figure reveals these

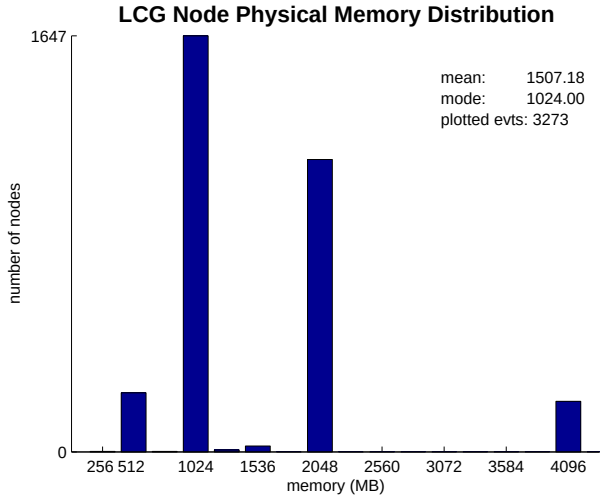

Figure 3.21: *Physical memory of LCG nodes used by DC04 jobs.*

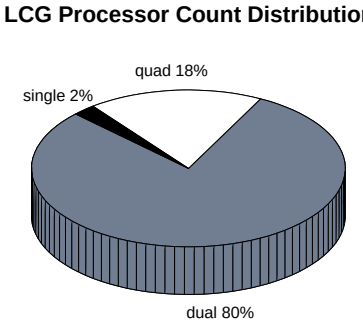

Figure 3.22: *CPU count of LCG nodes used by DC04 jobs.*

conditions with bimodal job densities for a given processor speed. It also reveals the relative performance of the AMD processors is better than the Intel processors based on processor speed. In terms of an overview of the actual nodes within LCG, the DC04 jobs ran on 3272 unique nodes, and their processor speeds and physical memory distributions are shown in Figures 3.20 and 3.21 respectively. Figure 3.22 shows the distribution of processors per node.

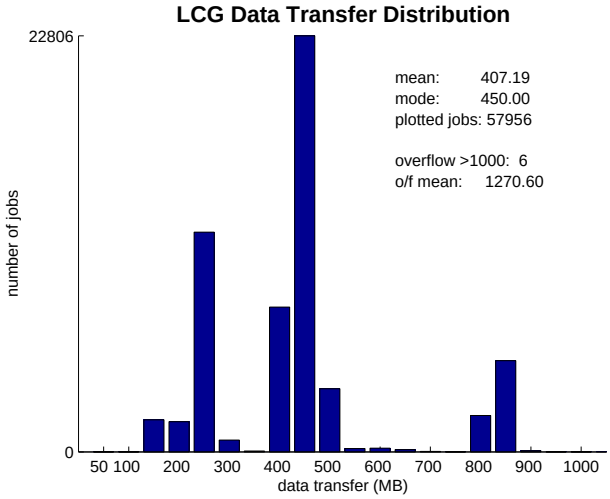

Figure 3.23: *Histogram of data generated by DC04 jobs.*

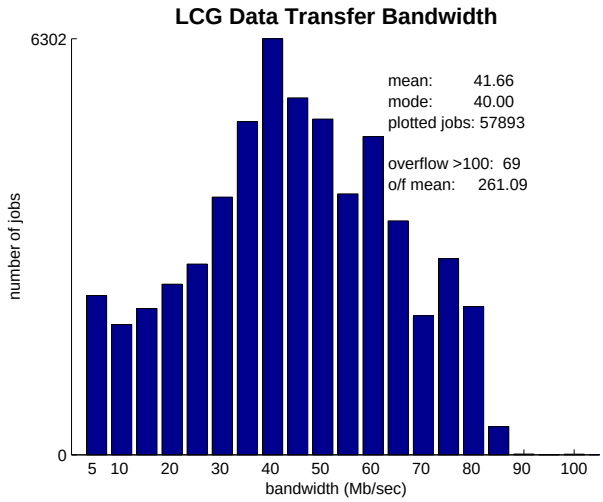

Figure 3.24: *Bandwidth of data transfer stage at end of each DC04 job. Typical transfer volume was 400 MB.*

Data Transfer Bandwidth

As the DC04 jobs produced relatively large amounts of data and transferred these to remote sites on job completion, it was possible to observe the sustained network bandwidth. Figure 3.23 illustrates the distribution of generated data over almost 58 thousands jobs, while Figure 3.24 shows the sustained transfer bandwidth. Although there were problems with the data management mechanisms within DIRAC and provided by LCG (as discussed in Section 3.7.3), LHCb was encouraged to see sustained transfer rates of 40 Mb/s were achieved during DC04.

3.9 Summary of LHCb Computing

The 2004 Data Challenge was extremely successful in proving the viability of grid computing for particle physics. It revealed a number of operational issues with the LCG infrastructure, most of which have since been resolved. It was possible for a team consisting of a handful of people to execute and manage an extremely large volume of computational tasks over several months distributed at over 60 computing centres around the world. It was the first time LCG, or any large grid, had been saturated with computing tasks for an extended period time, measured in months, rather than days or weeks.

The physics software worked exceptionally well, with very low failure rates. This was seen as a significant achievement, given the wide variety of systems the jobs were run on. This can partially be attributed to the homogeneity of LCG nodes in terms of software installation and configuration. In the long term, and for the broader vision of grid computing, this is not sustainable. In many ways LCG reflects a large scale distributed cluster management system, rather than a federated network of heterogeneous computing resources. LCG is federated, but the software stack on each LCG node was essentially identical. In any case, DIRAC managed both "classic" computing centres which had variations in software configuration and LCG-based computing equally well, due to the modular software packaging techniques which were used and the ability of jobs to download and install any missing software to the local job-space using unprivileged accounts. Up to 2 GB of software could be downloaded, installed, and configured, typically in under 15 minutes.

The general success of DIRAC has been attributed to the modular and simple design philosophy. By encapsulating functionality into independent services with basic interfaces it has been possible to focus on correctness of behaviour. Simple decoupled services are also easier to replicate, addressing Key Goals 1 and 2 (Scalability and Reliability). Simple services facilitate the creation of multiple implementations, possibly with extended functionality, thus addressing Key Goal 4 (Extensibility). The stability of the DIRAC infrastructure was achieved through many channels, key among them being service replication, asynchronous decoupled messaging between services, and retry/fail-over mechanisms. The pull scheduling paradigm worked extremely well for the class of simulation tasks which were being executed. Preliminary evaluation of the system for the more chaotic and widely varying analysis jobs suggests the Optimisers and pull paradigm will provide good responsiveness, however this is in the process of being confirmed. Pooling of tasks into job queues and utilising pull scheduling to overcome the tractability problem

of optimal task assignment was shown to be a significant improvement on the LCG push scheduling approach. Later sections will develop the concept used in DIRAC with a more formal model which extends the Condor Match Making system. Instant messaging in this domain certainly could provide significant advantages which need to be explored further. Furthermore, with an XMPP interface to Agents, a peer-to-peer network of directly interacting Agents is envisioned. This would reduce, and possibly even eliminate, the reliance on the Central Services, as Agents could dynamically load-balance by taking extra jobs from overloaded sites.

During DC04 DIRAC operated in a trusted environment, and therefore has had only a minimal emphasis on security. Common access passwords were circulated within the operational team, and task activity was constantly monitored for misbehaving tasks. A more comprehensive strategy is required for managing authentication and authorisation of Agents, Users, Jobs, and Services. It is hoped that a TLS based mechanism can be put in place with encrypted and authenticated XML-RPC calls using a combination of the GridSite project[81], and the Clarens Grid Enabled Web Services Framework, from the CERN CMS project. It is essential that DIRAC comes to properly incorporate X.509 certificates which are the basis for identity management and trust within LCG. In this sense, Key Goal 6, Security, is only minimally addressed within DIRAC.

The operational goals, Useability and Manageability, were less of an emphasis for DC04. It was essential to accurately track task state, and this was done via the Job Monitoring Service, and the logging website. As discussed in Section 3.6.3, XMPP provided powerful tools for live monitoring of agents and services via instant messaging. For Key Goal 5 (Manageability) to be more satisfactorily achieved aspects such as Role Based Access Control (RBAC) and task-level instant messaging clients are necessary. A more integrated and interactive web-based monitoring and control mechanism is also desirable. For Key Goal 3 (Usability), there was not a sufficiently large user base to evaluate this aspect of DIRAC. The GANGA user interface has been developed in conjunction with the ATLAS experiment and provides a graphical interface for job creation, submission, and management. The nature of the work during DC04 was such that it did not warrant the use of GANGA, and instead command line tools were utilised by the DC04 team for job creation, submission, and management of all tasks, with Services deployed manually and Agents deployed either through LCG “pilot” tasks or manually.

DIRAC showed itself to provide a complete grid computing infrastructure. The modular services approach allowed it to integrate with LCG to layer on top of

another computing grid, while simultaneously utilising individual workstations on a cycle scavenging basis, and other traditional computing clusters using the same Agent-based model. LHCb expects further integration of DIRAC with LCG, especially in terms of storage resources which will be critical.

DIRAC will follow with interest the Web Service-Resource Framework (WSRF), the successor to OGSI, with the hopes that the ease of use, language neutrality, tool availability, and complexity level will make it suitable for future versions of DIRAC. It is hoped that computational grids can benefit from the maturing technologies in peer-to-peer file sharing, instant messaging, and global computing which benefit from economies of scale. It is felt that a convergence and integration of these areas will provide solutions to many outstanding issues in grid computing.

3.10 Summary

The main observation which can be drawn from the work described in this chapter is that existing grid computing infrastructures continue to fall short of the early promises of "The Grid". Interestingly, since these early promises were made, many high performance computing users have found they require a computing infrastructure which, once successfully implemented, will provide a computational grid. In this way, "promises for the future" have collided with "requirements for today", and this is especially true for the global network of collaborating particle physicists. While HEPCAL I[44] and HEPCAL II[45] provide a very thorough description of the common computing requirements for the new CERN particle physics experiments, an unavoidable conclusion is that it is first necessary to get the basics working correctly before the rich set of requirements described in the HEPCAL reports can be addressed. The early expectation of the CERN-centred grid computing work was that these issues had already been satisfied, or that they would be trivial extensions to the existing systems. On many fronts this has not been the case. Data management, security, software deployment, and task management have all presented significant challenges which the current middleware is only beginning to overcome. The position of the LHCb computing group that simple, decoupled services are necessary for successful grid computing, rather than a single monolithic system, appear to have been validated by the success of DIRAC in providing a robust computing infrastructure even when faced with an unreliable underlying system (LCG).

This chapter reports on the first comprehensive quantitative description of an operational computational grid utilising dozens of sites over several months and

capturing details of worker nodes, system configuration, data transfer, task performance, and grid middleware performance[1]. This illustrates the degree of dynamism and heterogeneity within a grid, and underlines the need for fault tolerance – faults and failures are a constant in the grid.

This work exposed four avenues for further investigation: security, data management, physics analysis environments, and improved workload management. The first two have been addressed directly by the LHCb computing group. The third has been the topic of other PhD research[82], with many of these developments already incorporated into the most recent version of DIRAC. The final area has been pursued in two different directions. Firstly in [83] which extends, formally describes, and then simulates the DIRAC workload management system illustrating its performance in a dynamic, heterogenous grid environment. Secondly, the DIRAC workload management system was remodelled by the author, focusing on refinements of the Condor ClassAds/Matchmaking system, creating a formal model for task queues and resource symmetry, applying a RESTful design philosophy, and considering the overall lifecycle of a "grid process". That is the work presented in later chapters, following a survey of the main task management approaches which are currently available.

Chapter 4

Cluster and Grid Task Management

This chapter considers the most popular cluster and grid task management mechanisms, specifically from a REST perspective. It starts by discussing the features of a modern preemptive operating system, highlighting those features which must be translated to a grid domain and discussing the difficulties in doing so. A brief summary of traditional process scheduling strategies and grid resource management systems discusses the difficulties of applying these techniques to a large computational grid. This leads to the proposal for RESTful resource management, with a starting point of resource representation. Nine popular grid resource management systems are reviewed. Through a REST lens, the focus of the discussion is on grid resource representations, identifying strengths and weaknesses of the various approaches, and finally concluding with observations which guide the proposed Grid Resource Description Language, presented in the following chapter.

The operation of a computational grid can be broadly divided into three stages: knowledge of present system state, description of desired future state, and a mechanism to transform the current system state into the desired state. These are typically handled by independent systems: the first by monitoring of physical resources (for example executors, storage, and network), the second through task descriptions, and the third via resource management policies and services. This chapter considers existing mechanisms for the first two of these problems: gathering of information regarding physical systems, and task descriptions. The objective is to identify properties for a RESTful grid task management system. This is done in the context of particle physics computing with the objective of providing a large scale generic computational grid.

In the context of the conclusions of the previous chapter, the heterogeneous federated nature of computational grids demands a simple architecture which facilitates

multiple implementations, and a decomposition of the architecture into multiple components to enable a service oriented approach where individual grid sites can choose which services are available, and which implementations are utilised. In this context, while something like WSDL and Web Services can provide a *lingua franca* for the services, a REST approach requires an equivalent to describe the resources within the grid. As described in Section 2.3, the REST approach of common resource descriptions eliminates the necessity of common service APIs, although, of course, any implementation would need to be aware of these to enable service interaction. Furthermore, it is argued that the resource scheduling problem, which is a focus of this work, requires a distributed symmetric matching mechanism, similar to Condor ClassAds. In order to discuss the development of these ideas in later chapters it is first necessary to consider what is provided by existing resource description languages.

This chapter begins by considering modern operating systems and the mechanism by which they provide reliable, consistent process execution and management. This is followed by a summary of process scheduling strategies and their application to the grid domain. It continues with an analysis of existing description languages for tasks and executors and then outlines the requirements deemed necessary for symmetric matchmaking in computational grids.

4.1 Operating Systems and Computational Grids

Ultimately much of the work in clusters, batch systems, and computational grids is an attempt to expand the behaviour of a single computer preemptive operating system. While it is not appropriate to enter into a detailed consideration of modern operating systems, it is valuable to consider the key features which enable reliable program execution, data and process security, process management, and software portability. These systems and approaches have been developed and refined over decades, and therefore offer great insight into features which are also likely to be necessary in computational grids. Detailed descriptions of modern operating system architectures can be found in reference works such as [84], [85], and [86]. Three characteristics of operating systems are considered:

1. An omnipotent and omniscient kernel (or equivalent);
2. Stable and homogeneous system;
3. Security layer for data and processes.

The first characteristic is that modern operating systems utilise a kernel, whether that is a micro-kernel or a monolithic kernel. This kernel has the ability to preempt any "user-level" process and suspend it indefinitely. This omnipotent and omniscient control allows the kernel to provide atomic operations both on data and processes. Atomic operations and the omnipotent kernel provide a high degree of assurance that system-level operations will be completed in a consistent and timely fashion.

At the very least the kernel provides a process management system which keeps track of all "live" processes, typically in a *process table*. This contains aspects such as process ancestry and progeny, memory usage, file access, environment settings (*i.e.* basic global variables), ownership, current state, program counter, and performance or usage metrics. A well defined kernel API provides user-level processes operations for querying or changing their process state, or, if authorised, other processes' state.

The second characteristic is a relatively fixed, stable, and homogeneous base system. Almost any hardware change will require a restart, and *user level processes* (as opposed to long-running and automatically started *services*, which technically may also be user-level processes) rarely expect or manage process state continuity across a restart. Similarly software is expected to utilise the system configuration found at installation time. If available software, services, or libraries change, there is usually no expectation that dependent software will dynamically adjust. Backwards compatible services or libraries may be the exception to this, however the assumption is still that if the software installation process was successful (that is, if the software ran *once*) then it will run successfully at any point in the future. A violation of this through changing system configuration will not automatically be accommodated or even detected and the software in question will likely fail in an unpredictable way.

The third characteristic is a security infrastructure which manages process and data access. The common paradigm is the use of user and group access policies attached to all data and processes, along with a user authentication system which binds a real user to a single user identity and a set of group identities. Users can then interact with data or processes provided they possess the suitable identity given that resource's access policy. Again, kernel level services act as the omnipotent arbiter of data and process access control. Typically users have limited ability to delegate user or group ownership of data and processes, but are able to change the access policy for those resources they own.

These characteristics are contrasted against large-scale computational grids which are federated, dynamic, heterogeneous, have no single information authority, and require rich identity and authorisation management for data and tasks (the grid

analogue of an operating system process). There is no omnipotent "hand of God" which can mediate interactions, nor omniscient "eye of God" which can simultaneously know a complete, accurate, and up-to-date picture of the state of the computational grid. This requires that tasks in a computational grid provide a complete description of their dependencies and configuration such that this can be transferred to remote resources. Conversely, capabilities and restrictions of executors must be sufficiently described to provide reasonable confidence that assigned tasks will successfully execute.

The key characteristics which map from operating system process management to computational grids are the data management, process management, configuration, and ownership properties.

Grid tasks will require data input, data output, and data modification. Operating systems provide mechanisms to control data access and provide file creation and file pinning. The nature of data access on the grid requires that, wherever possible, data requirements are specified *a priori* as dynamic grid data access may be slow, expensive, or impossible.

Grid tasks will have a life cycle, and encapsulating the current process state and providing an interface for operating on that state is essential. Operations analogous to suspend, kill, query, wait, and continue are all required. The distributed nature of a computational grid will make a central *grid process table* difficult to maintain and is unlikely to be scalable. While an operating system kernel tracks process life cycle through a program counter, its environment, file handles, and open memory regions, a grid process cannot expect process management at such a fine grained level. A concept of execution stages and state for each stage provides a coarse grain mechanism for process-internal workflow.

Configuration comes in a number of forms. In an operating system context, a new process inherits the environment settings from its parent and then starts executing a programme with specific arguments and a set of open file handles. Ultimately a grid process will execute in an operating system context, therefore a mechanism is required to specify the environment, arguments, and file set. Parametrisation and the use of variables is also an important aspect of configuration management.

From another perspective, grid process configuration consists of specifying a software set required by the process. In an operating system context, as mentioned above, it is assumed that all necessary libraries and software for process execution are available. In a grid environment it is necessary to specify the software which is required, whether that be implicitly via a tag for a software set, or explicitly via

naming individual packages. Furthermore, specification of specific software versions is an important consideration in a federated computing environment, where there may be considerable version skew between sites.

Finally the concepts of ownership, access, delegation, and identity management must be addressed. Operating systems have an effective mechanism for separating data and process access through the use of user, group, and "global" (within the scope of the system security domain) access policies. This same separation is necessary in a grid context, however a much richer set of access and identity operations are also necessary. It is necessary to specify access control beyond the scope of the operating system security domain. The set of grid data read and write operations required for a single grid process may require the use of a range of identities. Similarly process execution or service access may make use of numerous different identities for access, and require delegation of identities to those services or processes to complete aspects of the grid process. A computational grid environment is characterised by groups of collaborating users working across organisational boundaries and thus security domains. Managing access correctly so the appropriate set of users (and no more) can manage the grid process and its artifacts (*i.e.* data) is essential.

4.2 Traditional Task Scheduling and Resource Management

The classic theory for task scheduling on a set of machines was first presented in 1967 in [87], which defined scheduling problem classes, metrics, and algorithms. While this work was focused on the problem of scheduling people, tasks, and machines in an industrial or operational research context, the same principles apply to compute clusters and grids. In the 1970s and 1980s, extensive work was been done to address the problem of process scheduling on single processor so-called "time-sharing" systems, as the previous section discussed, and for SPMD parallel super computer applications with hundreds or thousands of processors, and [88] provides a thorough survey and taxonomy classification of this work. These same principles were refined and applied through the 1990s to cluster computing, with the EASY (Extensible Argonne Scheduler System) backfill scheduling algorithm gaining prominence[89], amidst other scheduling strategies which aimed to improve on First-Come-First-Serve (FCFS) "naive" task scheduling.

In an online, non-preemptive scheduling environment (*i.e.* where task scheduling occurs dynamically as new tasks arrive, and where running tasks cannot be

preempted), it is well known that optimal scheduling is a difficult, NP-complete problem[90]. As a result, most systems, such as Torque or Maui[91], and Platform LSF[92], use heuristic methods, aiming for a balance between users' desire for fast response time, and system administrators' desire for efficient resource usage and fair-share.

With the advent of grid computing, scheduling in a federated environment is complicated by the addition of multiple administrative domains, in contrast to single site, single scheduler systems which have full authority to assert a schedule on the subordinate resources. [93] shows that the same "single site" strategies are utilised in a grid domain, however [94] discusses the difficulties of applying cluster scheduling to the grid domain. While it may be appropriate for traditional task scheduling and resource management to be applied at the local cluster level (although even this strategy is challenged by Condor[13] and BOINC[60]), it can be seen that the federated nature, heterogeneity, and dynamism of grids make it difficult to extend these strategies to grids. In this circumstance, "grid scheduling" effectively has become a matter of grid system architecture or the result of a particular implementation. Regarding the latter, a particular implementation is not feasible for a large scale grid, as it implies a common, homogeneous software system. Regarding the former, [94] shows that existing computational grid "architectures" are extensions of traditional cluster management systems into a distributed environment. They require coordination between all sites, universal information sources, and often a centralised workload management system for task scheduling and allocation. They also exclusively (again, with the exception of Condor) deal with the allocation of tasks to computing resources, and consider all other entities within the system second class objects.

These architectural paradigms contradict the philosophy presented in [17, 95] and are not seen to be realistic for an Internet-scale computational grid. This strategy can be seen in the LCG Workload Management System[96] and the CREAM interface deployed at sites[97]. It is unclear how many of the shortcomings reported in [98] have been resolved by this revision of the LCG WMS architecture. Instead, a strategy based on the REST principles described in Section 2.3 is presented later in this work as an alternative starting point for grid resource interaction. All entities are abstracted to *grid resources* and numerous interaction patterns are possible, falling within a common RESTful framework. This framework requires a common descriptive approach to enable the *representation* aspect of RESTful grid resources. The rest of this chapter will consider existing mechanisms for describing

entities within a grid, critiquing them and building a list of desirable properties for a RESTful resource description strategy. This approach sidesteps the issue of which scheduling strategy to use, leaving that to either be a site-specific decision (*i.e.* for local task management), or a domain-specific implementation, operating within the boundaries of the RESTful resource interaction constraints. The RESTful model, then, rephrases the task scheduling problem as a resource interaction and composition problem. Subsequent chapters will develop this model, the constraints it applies to resource composition (which very much borrows from the Condor Matchmaking-style interaction), the dimensions of freedom available for implementation design, the foundation of this model in set theory (in order to provide strong mathematical properties to resource compositions), and an introduction to some of the valuable properties which result from this RESTful model.

4.3 Existing Description Mechanisms

The nature of computational grids is such that it is rarely possible to simply name a programme, its arguments, and a set of inputs in order to perform the desired computational task. In a standard time sharing system (*i.e.* on a typical workstation or server) such operations are possible because:

1. they are synchronous and block until the operation completes;
2. they can utilise the underlying locally accessible data store;
3. any user environment settings or configuration is directly available;
4. the capabilities and configuration of the local system are generally known and relatively static;
5. the system is assumed to be stable – any failure or restart will invalidate all active processes.

Batch systems and computing clusters have required a more detailed description of the task in order to prioritise it properly, provide the appropriate configuration and data, and correctly match varying task requirements to computing resource characteristics. Prior to the advent of grid computing, it was possible to leave many requirements as implicit or encapsulated in the properties of a particular job queue. Grid computing has introduced the complexity of externally-controlled heterogeneous computing resources and widely distributed data sets which make it

```
//JOB1      JOB (034D),'RAMESH',CLASS='A',
//  PRTY=6,TYPRUN=HOLD,NOTIFY=ERT54
//stepname      EXEC MINISORT,TIME=60
//SORTIN     DD DSN=input.dataset,DISP=SHR
//SORTOUT    DD DSN=output.dataset,
//  RETPD=____,DISP=(NEW,CATLG,DELETE),
//  SPACE=____,DCB=____
//SYSIN      DD *
```

Listing 4.1: *Sample JCL executing the minisort algorithm.*

necessary to explicitly define all capabilities and requirements. It is also necessary to carefully consider the mixture of security domains requiring different authorisations.

Existing description languages have typically been for specific software packages (RSL, JCL, JDL)[99–101] or so generic as to lack any semantic structure allowing checking and validation (ClassAds)[21]. Efforts to develop generic languages have not addressed the duality of tasks and resources, and have had a high degree of complexity (JSDL, CDDLM)[102, 103].

This section looks at existing approaches and comments on their suitability for large scale computational grids, with the specific goal of identifying features necessary for an abstract computational grid task model that supports symmetric resource matching, as will bepresented in Chapter 5.

4.3.1 Job Control Language

Many batch and grid job description languages trace their roots back to the Job Control Language which was developed by IBM for the OS/3xx series mainframes in the 1970s. This is a suitable starting point, as it pre-dates modern preemptive multi-user time-sharing operating systems while still addressing many of the issues relevant in computational grids. JCL contains aspects which are necessary in all scheduled task description languages: program to execute, conditionals, input and output data requirements, program class, notification mechanisms, accounting details, and time limits and estimates. Listing 4.1 shows an example of JCL.

This language is notorious for its opaqueness and terse syntactical structure, thus emphasizing the importance of an intuitive and simple syntax. While JCL is considered primarily for historical reasons, we can see in Listing 4.1 the following key features:

1. accounting (`034D` identifies the task)
2. class (`CLASS='A'`)
3. data staging (`DD` accesses or creates a file)
4. data access control (`DISP=SHR` states that the data is shared with other jobs)
5. naming (`JOB1` provides a human-readable name for the job)
6. ownership (`RAMESH` is the user who owns the job)
7. priority (`PRTY=6`)
8. timing (`TIME=60` limits the job to 60 minutes)
9. program details (`EXEC MINISORT` is the program to run)

JCL does not address *executor* requirements as these are assumed to be known by the users as JCL is typically intended for execution on a particular system (the system on which the JCL is composed and submitted).

4.3.2 PBS and Torque

The Portable Batch System (PBS)[104] was originally developed by NASA in the 1990s. It is representative of a class of batch system/cluster management packages such as Sun Grid Engine (SGE) [1], OpenPBS and PBSPro by Altair [2], Torque, by Cluster Resources [3], LSF[92], by Platform [4], and LoadLeveler by IBM [5]. Torque is the most recent open source branch of the original PBS software, and both OpenPBS and Torque are widely used within the particle physics community, hence its selection as the representative example.

This class of batch system software typically consists of three types of hardware nodes, and requires three different sorts of configuration. The three hardware nodes are **submit, head**, and **worker**. The submit nodes have user interface tools installed and allow registered users to authenticate to the batch system then create, submit, monitor, and retrieve tasks. These tasks are submitted to the batch system head nodes (also sometimes called the **gatekeeper** nodes) which manage the pool

[1] http://gridengine.sunsource.net/
[2] http://www.altair.com/
[3] http://www.clusterresources.com/
[4] http://www.platform.com/
[5] http://www.redbooks.ibm.com/abstracts/sg246038.html

Property	Description
arch	architecture
cput	maximum CPU time used by all processes in the task
file	maximum disk space consumed by the task
mem	maximum physical memory used by the task
vmem	maximum virtual memory used by the task
nodes	number of nodes required by the task
pcput	maximum CPU time used by any given process in the task
pmem	maximum physical memory used by any given process in the task
pvmem	maximum virtual memory used by any given process in the task
software	any software required by the task
walltime	maxmum wall clock time the task can be in the running state

Table 4.1: *Main task description properties in Torque*

of worker nodes in the cluster. The head nodes maintain a number of different task queues and a combination of user task parameters and administrative policy assign each submitted task to an appropriate queue. The head node monitors the worker nodes and submits tasks from the various queues to available worker nodes based on a combination of the queue policy and a global policy. The three configuration items are **task definitions**, **queue definitions**, and **load balancing policy**. Users control the first, while administrators control the latter two. As an example, Table 4.1 lists the main properties a user may utilise in describing the requirements of their task when submitting it to Torque.

Queue definitions in Torque effectively use the same set of properties for each queue and then match the tasks to the appropriate queues via a matching algorithm and a defined priority on each queue. If the task limits, such as *walltime, cputime, file, mem*, are exceeded then the task is aborted. The load balancing policy in Torque is very basic, and in most cases the Maui scheduler (a separate module, also from Cluster Resources, which interfaces to Torque) is used instead. This manages advanced policies to allocate tasks to queues and select which tasks are run on the worker nodes. Similarly, in other batch systems the scheduler component takes responsibility for prioritisation of task execution and queue management based on defined usage policies.

These systems already provide robust, mature solutions for managing single-site, (relatively) homogeneous and static hardware clusters with shared data storage space. Their shortcomings, however, are that they do not translate into a grid domain. The task queue definitions are fixed. The head nodes typically require complete control over the slave worker nodes, a shared file system is often necessary, and task submission utilises combinations of proprietary arguments, environment

variables, and shell script extensions. None of these are acceptable in a dynamic, federated, heterogenous grid environment. Furthermore, the scheduling strategies utilised by most assume a maximum aggregate queued task size of thousands of tasks on hundreds or perhaps a few thousand worker nodes as the scheduler will attempt to maintain a current, complete, and correct view of the entire sysetm state. An Internet scale grid cannot operate under these conditions. It therefore becomes necessary to develop new approaches which prioritise the features of a batch system and keeps only those which can be realised, possibly in a dramatically different way, in a grid environment.

4.3.3 ClassAds

The Condor Project[13], which has developed a distributed batch scheduling system, developed the concept of Classified Advertisements [21, 105], whereby any entity could express arbitrary characteristics of itself as a *seller* and assert requirements on its *buyer*. This duality does not differentiate between *tasks* and *executors*. It allows a task to specify restrictions on the executor it will accept, and an executor to specify restrictions on the tasks it will accept. In this way a *Matchmaker* is used to build candidate sets of matching task/executor pairs, and via a ranking policy the Matchmaker can chose the best match. The matchmaking can be done at the point of submission (push-scheduling), by the executor (pull-scheduling), or by a third-party independent matchmaker. This system allows a high degree of dynamism in the executor characteristics, rather than simply having static characteristics implied by a queue definition.

The ClassAds Matchmaker does not, itself, have any policy mechanism for asserting ClassAd structure, mandatory attributes, matching requirements, or ranking. It does handle the `Requirements` (or `Constraint`) and `Rank` attributes specially, using them as keys to form a match set and then rank that set, however the ClassAds themselves specify the policy. The language is formally defined[105] and has both C++ and Java implementations, with wrappers for several other languages. This makes it possible to implement policies through the manipulation of ClassAds at the matchmaker. Extensions have been developed to facilitate matching a job to a set of executors (set matching)[106] and for heterogeneous resource matching (gang matching)[107].

Listing 4.2 shows a sample ClassAd describing a computing resource, while Listing 4.3 and 4.4 show a generic ClassAd in both the classic and XML representations.

```
[
Type         = "Machine";
Disk         = 700000;
Memory       = 128;
State        = "Unclaimed";
Name         = "grid.physics.ox.ac.uk";
ResearchGroup = { "grid", "physics" };
Friends      = { "cioffi", "soroko" };
Untrusted    = { "mckeever", "mcarthur" };
Rank         = member(other.Owner, ResearchGroup);
Constraint   = !member(other.Owner, Untrusted) and
               other.Type == "Task";
]
```

Listing 4.2: *Sample resource ClassAd for an executor (Machine) in native syntax [21].*

```
[
w = 1;
x = w + 1.5;
y = { w, "ABC" };
z = y[1];
]
```

Listing 4.3: *Sample ClassAd in native syntax*

```
<c>
    <a name="w">
        <i>1</i>
    </a>
    <a name="x">
        <e>w+1.5</e>
    </a>
    <a name="y">
        <l>
            <e>w<e>
            <s>ABC</s>
        </l>
    </a>
    <a name="z">
        <e>y[1]</e>
    </a>
</c>
```

Listing 4.4: *Sample ClassAd in XML syntax*

Besides the generic, semi-structured ClassAd syntax, ClassAds also provide support for:

1. variables and variable referencing;

2. sets, and set operations;

3. arithmetic operations;
4. functions and boolean operators;
5. four scalar data types: strings, booleans, integers, and floating point numbers;
6. nesting and scoping of ClassAds.

ClassAds and Matchmaking have been adopted in several grid computing projects, and motivate much of the work presented later in this dissertation. ClassAds present the fundamental concepts of generic resource descriptions and arbitrary resource pairing, however these are only addressed to a degree suitable for pre-execution matching, rather than enabling a resource description throughout a resource's life cycle. ClassAds also lack semantic structure which complicates validation and processing. Aspects such as authentication and authorisation, accounting details, complex requirements (such as software dependencies), and interrelationships between characteristics cannot be specified easily, if at all. The complexity of the ClassAds specification strongly suggests that the Condor implementation will continue to be the only one available, and this is tied to an unintuitive syntax which is only suitable for describing scheduling properties of resources.

4.3.4 Grid Laboratory Uniform Environment Schema

Within the LCG project, the Grid Laboratory Uniform Environment (GLUE) Schema [108] has been adopted to provide consistent definitions of computing and storage resource attributes for use by owners (publishers) and users (consumers). GLUE provides both an abstract hierarchical model of entities, attributes, and relationships, as well as specific semantic definitions of attributes for use by various information and monitoring systems. The most prevalent use of GLUE to date is within the the LDAP information system of LCG. In this instance, the GLUE schema is mapped to a flat name/value pairing, where the hierarchical information is merged into the attribute name. Listing 4.5 provides a sample LDAP entry for a storage resource.

GLUE provides a carefully thought out abstract description model for storage and computing resources, as well as physical computing centres (sites), clusters within that site, task queues, individual computing nodes, and services. Within LCG, the GLUE information for task queues is translated from LDAP into ClassAd

```
dn: GlueSALocalID=dteam,GlueSEUniqueID=se.cern.ch,
    Mds-Vo-name=local,o=grid
objectClass:    GlueSATop
objectClass:    GlueSA
objectClass:    GlueSAPolicy
objectClass:    GlueSAState
objectClass:    GlueSAAccessControlBase
objectClass:    GlueKey
objectClass:    GlueSchemaVersion

GlueSALocalID:                dteam
GlueSAAccessControlBaseRule: dteam
GlueSARoot:                   /storage/dteam
GlueSAPath:                   /storage/dteam
GlueSAType:                   permanent

GlueSAPolicyMaxFileSize:     10000
GlueSAPolicyMinFileSize:     1
GlueSAPolicyMaxData:         100
GlueSAPolicyMaxNumFiles:     999999
GlueSAPolicyMaxPinDuration:  10
GlueSAPolicyQuota:           0
GlueSAStateAvailableSpace:   69924040
GlueSAStateUsedSpace:        2008660
GlueSAPolicyFileLifeTime:    permanent

GlueChunkKey:           GlueSEUniqueID=se.cern.ch
GlueSchemaVersionMajor: 1
GlueSchemaVersionMinor: 2
```

Listing 4.5: *Sample LDIF (LDAP) representation of GLUE information for a grid storage resource*

format to be used for matchmaking. This process is done internal to the *Resource Broker* which performs all matchmaking.

Appendix A.3 summarises the key resource attributes defined in version 1.2 of the GLUE schema. GLUE has been invaluable to the grid community by providing a pool of common terms for describing storage and executor resources. It has been adopted by numerous resource providers (iVDGL[6], CrossGRID[7], PPDG[8], Ninf/ApGRID[9], EDG[10], GriPhyN[11], IN2P3[12], DataTAG[13], INFN[14], LCG[15]), and is in use by the projects and experiments which utilise those resources.

The specific usage of GLUE in LDAP, however, has been problematic as LDAP was designed for relatively static information, with an emphasis on read operations over write operations. Centralising many resource descriptions (hundreds to thousands) into a single LDAP directory and updating them regularly (tens to hundreds of entry updates per second) overwhelms current LDAP servers, not to mention the read query load. Finally, LDAP relies on a fixed schema to describe the data. Insertion of arbitrary resource descriptions is difficult and not easily handled.

The Relational Grid Monitoring Architecture (R-GMA)[58] was developed in response to these problems and meant to provide a distributed, dynamic resource information service which handled reads and writes to an information database. R-GMA utilises a SQL-like syntax for database queries, and returns tabular information.

4.3.5 Job Description Language

The Job Description Language developed by the European Data Grid project [109] , and now being extended by the EGEE project [101] is based on the Condor ClassAds language. A particular schema is asserted which sets known attributes with semantics and defined processing behaviour, while still allowing user-defined or extended attributes. Many attributes utilise default values if not specified. The special `Requirements` and `Rank` attributes are handled differently as the Matchmaker

[6]http://www.ivdgl.org/ International Virtual Data Grid Laboratory
[7]http://www.crossgrid.org/
[8]http://www.ppdg.net/ Particle Physics Data Grid (US)
[9]http://ninf.apgrid.org/ Asia Pacific Grid
[10]http://eu-datagrid.web.cern.ch/eu-datagrid/ European Data Grid
[11]http://www.griphyn.org/ Grid Physics Network (US)
[12]http://www.in2p3.fr/ Institut national de physique nuclaire et de physique des particules (France)
[13]http://datatag.web.cern.ch/datatag/ Data Trans Atlantic Grid
[14]http://www.infn.it/ Istituto Nazionale di Fisica Nucleare
[15]http://lcg.web.cern.ch/LCG/ LHC Computing Grid

```
[
Executable      = "script.sh";
Arguments       = "60";
StdOutput       = "sim.out";
StdError        = "sim.err";
OutputSandbox   = { "sim.err", "sim.out" };
InputSandbox    = { "script.sh" };
Requirements    =
  other.GlueCEStateStatus == "Production" ;
]
```

Listing 4.6: *Sample Job Descripton Language (JDL)*

is wrapped by a Workload Management System which takes responsibility for task management and integrates grid monitoring information into the process. This allows `Requirements` and `Rank` to refer to monitoring properties specified by the GLUE Schema for the target resource. These properties are maintained via the grid monitoring system, for example via MDS LDAP entries or R-GMA databases. In these cases, dynamic information can be queried directly from a live source during the scheduling process.

By utilising new features of the Condor ClassAd Matchmaker, JDL now supports the specification of gang-matching requirements (n-way matching instead of only bilateral matching), job work-flows via Directed Acyclic Graphs (DAGs), and job partitioning. In total, 41 attributes are defined[101]. These are listed in Appendix A.2. Listing 4.6 shows a basic JDL file.

The latest revision of JDL provides a very rich grid task description language. The enhancements beyond basic ClassAds provide structure necessary for a grid environment and definitions of the semantic behaviour of the attributes. It does continue to suffer from the problems of an awkward syntactic structure, difficult JDL validation, inconsistencies in case handling, and is also specific to task descriptions, rather than allowing the original genericity of ClassAds which supported executor descriptions equally well.

4.3.6 Resource Specification Language

The Resource Specification Language (RSL)[99] has been developed by the Globus Project as part of the GRAM[16] workload management system. It allows the user

[16]http://www.globus.org/toolkit/docs/4.0/execution/

```
(* Run the compiled program*)
& (executable = a.out)
  (directory  = /home/nobody )
  (arguments  = 42 "Header␣String")
  (count = 1)
```

Listing 4.7: *Sample Resource Specification (RSL) Language*

to specify a set of quasi-arbitrary statements, the relationships between those statements, and a mechanism for defining and referencing variables. The semantics for RSL statements are asserted by the consuming application, therefore RSL is similar to ClassAds in that it is a meta-syntax for resource description. GRAM, for instance, understands approximately 25 attributes, which are listed in Appendix A.1. The NorduGrid project has extended RSL to support additional attributes in a fashion similar to the JDL extensions to ClassAds[110]. This is called xRSL and is listed in Appendix A.5.

Listing 4.7 shows a sample of RSL with the conjunction of four statements via the `AND` clause with the `&` (ampersand) operator, termed a *conjunct request*. Disjunctions are represented by the `|` (pipe) operator, termed a *disjunct request*. *Multi requests* form a parallel set of statements by utilising the `+` (plus) operator. Compound statements are possible which join together multiple operators and statements. This allows for rich logic expressions.

The RSL syntax provides variables, literals, and lists. The multi-request construct allows for the specification of parallel jobs. All current implementations have a fixed attribute set, and the syntax requires a custom parser. Furthermore, it has been developed from the perspective of task submission and scheduling, rather than providing for the full task life cycle or being compatible with executor descriptions and symmetric matchmaking.

4.3.7 Job Submission Description Language

Work by the GGF has led to the XML-based Job Submission Description Language (JSDL)[102]. It contains a rich language for describing grid jobs, however many key areas for scheduling and job management were deemed to be out of scope and are not addressed. Furthermore, it does not address the issue of resource description. Between the core attributes and those from the POSIX-Application extension ap-

```
<JobDefinition
    xmlns     ="http://schemas.ggf.org/jsdl/2005/06/jsdl"
    xmlns:jp  ="http://schemas.ggf.org/jsdl/2005/06/jsdl-posix"
    xmlns:ex  ="http://www.example.org/">
  <JobDescription>
    <JobIdentification>
        <JobName>     My Gnuplot invocation </JobName>
        <Description> Simple application invocation of gnuplot.
        </Description>
    </JobIdentification>
  </JobDescription>
  <Application>
      <ApplicationName> gnuplot </ApplicationName>
      <jp:POSIXApplication>
          <jp:Executable> /usr/local/bin/gnuplot </jp:Executable>
          <jp:Argument>   options                </jp:Argument>
          <jp:Input>      input.date             </jp:Input>
          <jp:Output>     output1.png            </jp:Output>
      </jp:POSIXApplication>
  </Application>
  <Resources>
      <IndividualPhysicalMemory>
          <LowerBoundedRange> 2097152.0 </LowerBoundedRange>
      </IndividualPhysicalMemory>
      <TotalCPUCount>
          <Exact> 1.0 </Exact>
      </TotalCPUCount>
  </Resources>
  <DataStaging>
      <FileName>            input.dat </FileName>
      <CreationFlag>        overwrite </CreationFlag>
      <DeleteOnTermination> true      </DeleteOnTermination>
      <Source>
          <URI> http://example.com/~me/input.dat </URI>
      </Source>
  </DataStaging>
</JobDefinition>
```

Listing 4.8: *Sample Job Submission Description Language file (adapted from [102])*

proximately 50 attributes are supported. These are summarised in Appendix A.4. A sample of JSDL can be seen in Listing 4.8.

JSDL specifically does not attempt to describe inter-job relationships, or to model the entire job life cycle. It refers to yet-to-be-defined specifications such as Resource Requirement, Scheduling Description, Job Lifetime Management, and Job Policy to cover many of these unaddressed areas. This means issues such as software requirements or wall-clock time limits for execution are not specified by JSDL. Furthermore, identity token binding is not addressed, leaving open the question of how to select the appropriate credential for file access, service access, execution, or accounting.

The verbosity and high degree of structuring found in JSDL limits its flexibility and complicates implementation. This is likely to reduce its ease of use and

subsequent adoption. Nonetheless, it does provide a standardised vocabulary for describing tasks.

4.3.8 Configuration Description, Deployment, and Lifecycle Management

Another GGF specification nearing its first release is that for Grid Services Configuration Description, Deployment, and Lifecycle Management (CDDLM)[103]. Although this specifically concerns itself with Grid Services, now under the framework of WSRF, the concepts presented are applicable to both software deployment in a grid environment and the execution of a single *grid task*. Two of the objectives of the specification are to define a Configuration Description Language (CDL) and a Service Deployment API.

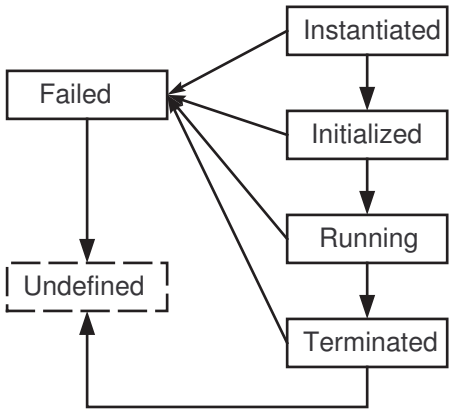

Figure 4.1: *State machine for CDDLM components.*

The CDDLM Working Group explicitly lists a number of key non-functional requirements for their proposed architecture: scalability, high availability, self healing, disaster recovery, full automation, and useability. An architecture which can achieve these aims for grid service deployment will provide valuable insights into mechanisms for grid task management. Unfortunately, CDDLM is still in its early stages which makes it difficult to evaluate its success at meeting those objectives.

Listing 4.9 shows an example of CDL for a system composed of five components. Figure 4.2 illustrates their relationship. A, B, and C are peers and operate concurrently, although they are initialised serially. D and E are sub-components of B and are started serially, only after B is successfully running.

```
<cdl:system>
  <cmp:sequence lifecycle="initialization"/>
  <cmp:flow     lifecycle="execution"/>
  <cmp:reverse  lifecycle="termination"/>
  <ComponentA/>
  <ComponentB>
    <cmp:sequence lifecycle="execution"/>
    <ComponentD/>
    <cmp:wait     lifecycle="initialization"
                  duration="10"/>
    <ComponentE/>
    <OnFailed process="notify"
              target="/ComponentA"/>
  </ComponentB>
  <ComponentC/>
</cdl:system>
```

Listing 4.9: *CDDLM Component Description Language example illustrating deployment of 5 components A-E*

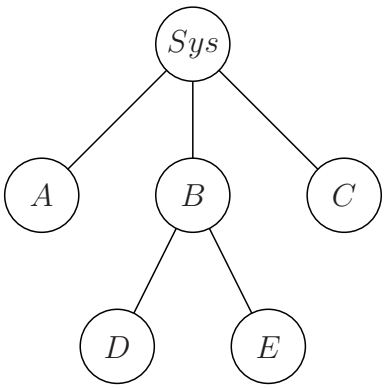

Figure 4.2: *Five component system deployed by CDDLM, matching Listing 4.9.*

CDDLM provides a basic life cycle state-machine, as shown in Figure 4.1, and mechanisms for the composition of components into a complete system. This composition is targetted entirely towards the deployment of a set of Web Services, rather than composing generic resources.

4.4 Summary

ClassAds provide the most generic resource description mechanism, as it is really a meta-syntax which can be used in arbitrary ways by different systems, and is capa-

ble of describing both tasks and executors. RSL similarly provides a meta-syntax however, in practice, a fixed attribute set is defined with a particular implementation, and the syntax requires custom parsers to be utilised. The GLUE Schema has become a *de facto* standard for describing executors, and JSDL is positioned to do the same for many aspects of task descriptions.

To approach resource description in a RESTful way, and considering the full resource life cycle, it is necessary to have an implementation independent way of describing both task and executor resources (and other resource classes) which can be easily generated by a publisher and utilised by a consumer. As was illustrated in Section 3.8, symmetric matchmaking is essential for flexibility and scalability in a grid environment. This combination of requirements suggests an XML-based meta-syntax with the properties of ClassAds, and provision for expressing the concepts found in GLUE, JSDL, JDL, CDDLM and the more traditional batch job description features of JCL and PBS. The format needs to be simple to allow ease of implementation and extensible to provide flexibility and expressiveness. The format should not assert a particular usage style or be tied to a particular implementation, and it must allow encapsulation of grid resource state throughout the entire resource's life cycle. Similar to the goals of HTML and HTTP[9], it should be cacheable and replicable. These observations are the motivation for the Grid Resource Description Language, presented in the following chapter. It presents an abstract model for generic grid resource description and an XML-representation for that model. It combines the best features of the approaches described in this chapter, while considering the features of an operating system process which are relevant for a grid process. In particular, ClassAds is generalised and extended, providing a detailed set theoretic model for RESTful representation of grid resources and resource composition strategy. This model captures a full life cycle description of a grid resource, rather than being limited to the pre-scheduling description, and focuses on a formal model, rather than providing a single specific implementation. The subsequent chapters will present this resource model, and describe a RESTful interaction via resource composition, and show how this lends itself to powerful features such as resource validation, templates, resource queues.

Chapter 5

A REST Model for Resource Matching

This chapter introduces a model for RESTful resource descriptions from the perspective of the scheduling properties required for large scale computational grids. This model consists, in part, of resource characteristics, requirements, and preferences. Specifically, it discusses the model for resource characteristics which forms the foundation for describing a grid resource and is used as a "base class" for requirements and preferences, presented in later chapters. Set theory is applied to define relationships between resource descriptions, such as equivalence, partial ordering, comparability, and composability. Three specific classes of resource comparators are presented, each containing a different degree of intelligence to facilitate resource comparison.

Unlike most other task and resource description languages which focus on the specific meanings of descriptors and bind these tightly to a particular process model and cluster or grid management system, the approach proposed here abstracts the concept of a "resource" and presents a framework for extensible resource descriptions. In the context of a large computational grid, with thousands of sites, hundreds of thousands of CPUs, and millions of tasks, it is inconceivable to require a single integrated software system. This same problem faced the developers of the early Web, and was addressed by separating the description of Web-accessible resources (HTML, to name just one) from the protocol for transacting them (HTTP), and allowed multiple implementations of both client side Web browsers and server side HTTP servers. Simplicity and extensibility have also been key to the success of the Web, and are therefore incorporated into the model proposed here [95]. The philosophy behind this design was described by Fielding as Representational State Transfer (REST)[9]. Most popular grid architectures are extensions of traditional batch processing systems and make assumptions which do not hold in a grid envi-

ronment. This work argues that the Web paradigm is a more useful basis for a grid architecture and presents a RESTful model of grid resources as a parallel for HTML in the Web domain.

The objective is to provide a common framework for grid systems to describe their various resources with well defined properties for the composition of those resources (*e.g.* assigning tasks to compute clusters, or finding storage space for files). In this domain interoperability between grid systems is largely encapsulated into a common interpretation of the resource descriptions.

A fundamental principle of federated grid environments is the autonomy of users and grid sites. For this reason, once a resource description has been accessed by a remote system that system may act in any way with the information. The resource owner at most has the ability to control access to its resources and resource descriptions and perhaps any billing/payment system which is associated with actions which follow from the resource description. In the application domain which this work is motivated by, namely particle physics computing, this situation is borne out by the reality of hundreds of independent research centres sharing the computational load of a large collaborative experiment. If administrators of a computing centre do not trust certain users they will either block them from submitting tasks or not fetch tasks from those users. Similarly users who do not trust a particular computing centre will block that centre from executing their tasks either by not submitting to them or not making their task pools available to the centre's task fetcher.

The model proposed here allows all parties to utilise a common grid resource description framework with the details of the resource description left for the collaborating groups to agree upon. The simplicity of the common framework allows different implementations to be realised, each suited to the needs of the owners. It focuses on the problem of composing grid resources, which is a more general form of the task scheduling problem in batch systems, however it is recognised this is only one aspect of the task and resource lifecycle. The model fits into a larger concept of stateful representations of resources and lays a foundation for transacting these representations and invoking operations on the resources which are not limited to any particular protocol or service API. This is fundamental to creating a RESTful grid architecture.

This chapter introduces the resource model and discusses the first aspect of it: characteristics. A set theory approach is taken, and the properties are explored. Set theory provides valuable properties for characteristic comparison which are discussed at length.

5.1 Resource Model Overview

The resource properties consist of three categories: **characteristics, requirements**, and **preferences**. Characteristics refer to properties intrinsic to the resource. Requirements refer to properties of other resources with which the current resource may be composed. Preferences provide a mechanism to select from a pool of resources which are valid for composition. Utilising the language of the GGF Grid Scheduling Architecture Research Group (GSA-RG) [1], the requirements allow a *candidate set* of compositional resources to be formed, and from that set the preferences allow *resource selection*.

In this model, requirements are a sub-class of characteristics, and preferences a sub-class of requirements, thus providing a hierarchical conceptual model of these different categories of resource properties. A formal description in Haskell of these properties can be found in Appendix C.1. This work describes relations between resources utilising these properties and discusses the features of the model in allowing various resource composition strategies. It remains largely an abstract model and emphasises a RESTful approach where the resource description is independent of the resource itself, explicitly allowing consumers of the representation full freedom of interpretation and subsequent action.

Set Theory Notation

Conventions and definitions for set theory notation are taken from [111]. Table 5.1 summarises the definitions of custom and uncommon symbols.

[1] https://forge.gridforum.org/projects/gsa-rg

Symbol	Definition
$\wedge$	conjunction (and)
$\vee$	disjunction (or)
$\triangleq$	definition
$\equiv$	equivalence
$\equiv_{bool}$	boolean equivalence, where given set S, if $S = \varnothing$, *false*, otherwise $S \neq \varnothing$, so *true*
$\sim$	operands are comparable (of same "type")
$\preceq$	operand A precedes operand B according to some partial ordering relation
$\subseteq_{char}$	subset relation only operates on characteristic sets within operands
$\subseteq_{part}$	partial subset, considering only shared dimensions
$\Rightarrow$	one-way match of operand A to operand B
$\Rightarrow_p$	one-way partial match subset, across dimensions shared between the two operands
$\Leftrightarrow$	two-way match of operand A and B
$\subseteq_{req}$	*requirement subset*, where the set of requirements in operand A are a subset and within the subspace range of the requirements in operand B
$\subseteq_{rs}$	*requirement space*, where the set of requirements in operand A define a smaller (more restrictive) subspace than the requirements in operand B
$\subseteq_m$	matching subset, equivalent to $\Rightarrow$
$\trianglerighteq_{pref}$	operand A implies a resource ordering consistent with operand B
$\dashv$	operand A is a *template* of operand B
$\triangleright$	start of comment (in algorithms)

Table 5.1: *Description of custom and uncommon symbols.*

5.2 Resource Characteristics

A resource R can be described with an arbitrary set of characteristics $Chars_R$. Each characteristic c_i (where i is the characteristic index) consists of a *(dimension, value)* pair, defining a unique characteristic of that resource. The components of a pair can be referred to individually as $c_i.dimension$ and $c_i.value$ respectively. It should be noted that $c_i.value$ may be *unspecified*. The interpretation of this special case will be discussed in later sections. The set of characteristic dimensions in a resource defines the dimensions over which the resource is described, and is denoted $\mathcal{D}_R$. This is expressed more formally in Equation 5.1. The total dimensional universe of a set of resources is denoted $\mathcal{D}$, and is defined by Equation 5.2. It is clear that Equation 5.3 holds, stating that the dimensions $\mathcal{D}_R$ of a particular resource R are a subset of the dimensional universe $\mathcal{D}$. A more formal definition of this is given in Haskell in Appendix C.2.

$$\mathcal{D}_R \triangleq \begin{cases} \text{set of characteristic dimensions in resource R} \\ \{c.dimension | c \in R.chars\} \end{cases} \tag{5.1}$$

$$\mathcal{D} \triangleq \begin{cases} \text{set of characteristic dimensions from all known resources} \\ \bigcup_i \mathcal{D}_{R_i} \end{cases} \tag{5.2}$$

$$\mathcal{D}_R \subseteq \mathcal{D} \tag{5.3}$$

The characteristic pairs for a resource form discrete points along an arbitrary set of dimensions therefore a resource's characteristics can be thought of as a discrete subset of a multi-dimensional value space. Equation 5.4 provides a definition for the set of all characteristics in R, and Equation 5.5 lists the various equivalent notations, with one or the other used depending on the situation. An indexed set of characteristics $\{c_i\}$ can also exist without any association to a particular resource R.

$$Chars_R \triangleq \begin{cases} \text{set of all characteristics in resource } R \\ \{c | c \in R.\text{chars}\} \end{cases} \tag{5.4}$$

$$Chars_R \equiv R.chars \tag{5.5}$$

$Chars_R$ is a view (or representation) of the resource R over a set of dimensions.

This presents state information concerning the resource R. Figure 5.1 provides a graphical illustration of a single characteristic value c_{x_1} in a single dimension x.

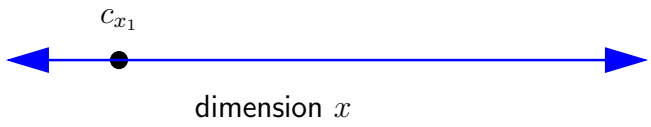

Figure 5.1: *A single characteristic in a single dimension*

A resource may have multiple characteristics in the same dimension meaning the resource presents multiple possible values (states) in that dimension. Alternatively, it may not specify a value for a particular dimension. This is interpreted as the resource being defined along that dimension, but at an unspecified point. Figures 5.2 and 5.3 illustrate these cases.

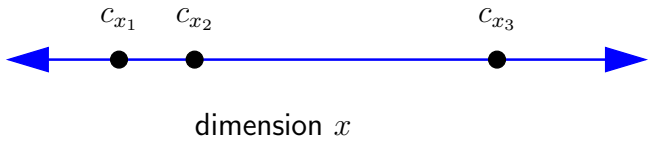

Figure 5.2: *Multiple characteristics in a single dimension*

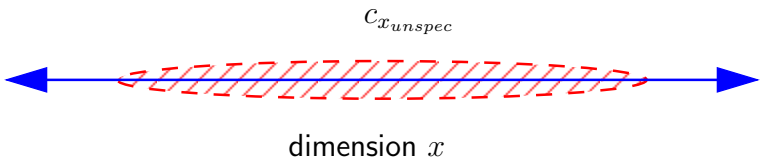

Figure 5.3: *A characteristic with an unspecified value*

It is often useful to talk about characteristics restricted to a particular set of dimensions. We use the notation dim_x to refer to the complete value space of dimension x. Using the standard concept of set intersection, we can then describe the characteristics of R restricted to those in the dimension x. This is expressed in Equation 5.6. This can be generalised to a set of dimensions as shown in Equation 5.7. This is analogous to *projections* which reduce the dimensionality of an n-tuple to an m-tuple, where $m < n$.

It is also common to want to restrict the dimensions of resource R to the set of dimensions found in resource S, or to the set of dimensions shared between resources R and S. To express these, we use the shorthand notations shown in Equation 5.8. The operator $\cap_{\mathcal{D}}$ is a shorthand because sets of characteristics (*e.g.* $Chars_R$) are not comparable to sets of dimensions (*e.g.* $\mathcal{D}_R$).

$$Chars_R \cap dim_x = \{c | c \in Chars_R \wedge c \in dim_x\} \tag{5.6}$$

$$Chars_R \cap \{dim_x, dim_y, dim_z, ...\} = \{c | c \in Chars_R \wedge c \in \cup\{dim_x, dim_y, dim_z, ...\}\} \tag{5.7}$$

$$Chars_R \cap_{\mathcal{D}} \mathcal{D}_S \triangleq \{c | c \in dim_x \wedge dim_x \in \mathcal{D}_S \wedge c \in Chars_R\} \tag{5.8}$$

A particular characteristic dimension for a particular resource R can be in a range of states:

undefined There are no characteristics defined for resource R in that dimension, therefore nothing more can be said about R over that dimension.

unspecified All characteristics in the dimension have no value specified and are therefore equivalent to a single *null*-valued characteristic. This is appropriate for characteristics which are "flags", or in cases where the resource possesses the characteristic dimension, but the value is unknown or not publicly available. This is not the same as *undefined*. It is a special case of a normal characteristic declaration.

Single definition There is exactly one characteristic defined in that dimension and it possesses a defined (specified) value.

Multiple definition There are multiple characteristics in the same dimension. Over that dimension the resource is defined by multiple values.

Figure 5.4 provides a simple model of a characteristic and its compositional relationship with a resource using standard UML notation. The *type* attribute will be discussed in the following section. The formal model of a Characteristic is described in Haskell in Appendix C.1.

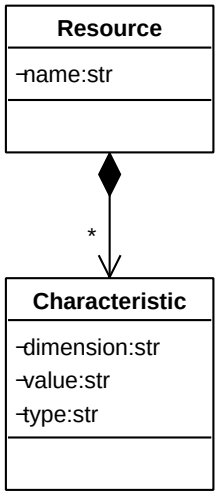

Figure 5.4: *Object model of a Characteristic and its composition relationship with a Resource.*

5.2.1 Value Types

A characteristic may have a type associated with its value. Type resolution is dependent upon the consumer of the characteristic, however the characteristic may suggest a type through a "hint" found in *c.type*. The consumer is not obligated to utilise this hint. Other possible sources of type information may be:

1. type inference from the characteristic name *c.name*
2. type guessing from the characteristic value *c.value*
3. a default specified by the resource R
4. a default built-in to the consumer
5. inferred by the nature of the operation carried out by the consumer

The consumer may utilise its understanding of the characteristic's type to perform transformations on the characteristic. For example, types may provide information regarding units, thereby allowing conversion of minutes to hours, or kilobytes to megabytes. Appendix C.3 provides Haskell definitions of a number of types, type categories, and type conversion functions. The types described are categorised into *metric, binary, speed,* and *time*. Section 5.4 discusses this in more depth.

```
<task name="testjob1">
<chars>
<norm_cpu type="hr" > 2   </norm_cpu>
<mem      type="MB" > 180 </mem>
<disk     type="MB" > 500 </disk>
</chars>
</task>
```

Listing 5.1: *Characteristics for a task resource*

5.2.2 Examples of Resource Characteristic Sets

The following listings are examples of characteristic sets for different types of resources. They utilise a simple XML syntax which will be described in this and subsequent chapters. A high level description of it would be a categorised set of name/value pairs, with typing information for each pair. These examples will be referred to in later sections either by their listing reference number or the *name* attribute on the root element.

Listing 5.1 illustrates the characteristics for a simple task. The details of the task itself are ignored for the time being. This task has only three dimensions: {*norm_cpu, mem, disk*}, with a single specified value for each dimension. Each dimension provides type hints for the consumer of the resource description.

Listing 5.2 illustrates the characteristics for two simple executors (compute nodes). The first, `pentium`, is defined across eight dimensions: {*cpu_type, mhz, os, software, bench, ram, vram, scratch*}, while the second, `amd`, is defined across six dimensions: {*cpu_type, mhz, os, bench, ram, net_bw*}. For `pentium`, one dimension, *software*, has two points defined: {*openssl, python*}, and one, *hyperthreading*, has none. For the *hyperthreading* dimension, `pentium` is in the *unspecified* state, as described earlier. As an example of a characteristic which is in the *undefined* state, consider the dimension *net_bw*. This dimension is not listed in the characteristic set for the executor `pentium` in Listing 5.2, therefore it is in the state *undefined*, whereas for `amd` it has a value of `1 Gbit/s`. The two resources `pentium` and `amd` have overlapping characteristics and can be compared.

Listing 5.3 illustrates some characteristics of a cricket player, expressed as a resource. It is defined across seven dimensions, and has no type hints. This example will be expanded upon in later sections to illustrate multi-way composition and the genericity of this approach to resource description.

```
<executor name="pentium">
<chars>
<cpu_type>  pentium </cpu_type>
<mhz>       2800    </mhz>
<os>        linux   </os>
<software>  openssl </software>
<software>  python  </software>
<hyperthreading/>

<bench    type="SI2K" > 0.9  </bench>
<ram      type="MB"   > 512  </ram>
<vram     type="MB"   > 1024 </vram>
<scratch  type="MB"   > 100  </scratch>
</chars>
</executor>

<executor name="amd">
<chars>
<cpu_type>  AMD   </cpu_type>
<mhz>       2500  </mhz>
<os>        linux </os>

<bench    type="SI2K"   > 1.1 </bench>
<ram      type="GB"     > 1   </ram>
<net_bw   type="Gbit/s" > 1   </net_bw>
</chars>
</executor>
```

Listing 5.2: *Two executor resources with different characteristic sets.*

```
<person name="James">
<chars>
<name>   James   </name>
<gender> male    </gender>
<age>    27      </age>
<weight> 85      </weight>
<height> 175     </height>
<sport>  cricket </sport>
<pos>    batsman </pos>
</chars>
</person>
```

Listing 5.3: *Characteristics describing a cricket player*

5.3 Comparability

Having defined the structure of a characteristic, characteristic dimensions, and the set of characteristics describing a resource, it is necessary to define common operations on, and relationships between, characteristics, dimensions, and resources. This starts with defining the concept of **characteristic comparability**. Ultimately this is the foundation of the compositional model for resource scheduling. Resources are described by characteristics, and compositions consist of comparing characteristics using standard set theory concepts. Briefly, a resource composes with another resource if the first resource's characteristics are found within the set of requirements defined for the second resource. This and future sections will develop this set theoretic compositional scheduling strategy.

Comparability is based on a (*dimension,type*) pair. A set of characteristics in the same dimension are only comparable if their types are the same. If the dimension is the same, but types differ, the set of characteristics can be thought of as forming an un-ordered set of sub-sets within the dimension, where each sub-set consists of characteristics with comparable (*dimension,type*) pairs. Characteristic comparability in this model is distinct from the formal definition of comparability, which is based on items being related by a partial order relation. Characteristic ordering will be discussed in the following section. The symbol $\sim$ will be used to indicate comparable characteristics. Specifically, if c_i and c_j are two characteristics, then $c_i \sim c_j$ indicates that c_i is comparable to c_j, and that c_i and c_j are in the same dimension and have the same type. This is clearly a symmetric relation, therefore $c_i \sim c_j$ implies $c_j \sim c_i$.

Figure 5.5 illustrates this concept graphically. Listing 5.4 provides an example of a set of characteristics in the same dimension with types which are not comparable. `INT95` and `INT2k` represent two integer benchmarks with a mapping function between them (*i.e.* they are comparable after transformation) while `FP95` and `FP2k` are two floating point bench marks, similarly with a transformation to a common base making them comparable, and `Alpha7` is a fictitious application specific benchmark. These form three distinct sub-sets within the *bench* dimension. Characteristic transformations that change dimensions, types, and values will be discussed in detail in Section 5.5. Without some form of transformation, same dimension characteristics with different types are **not** comparable.

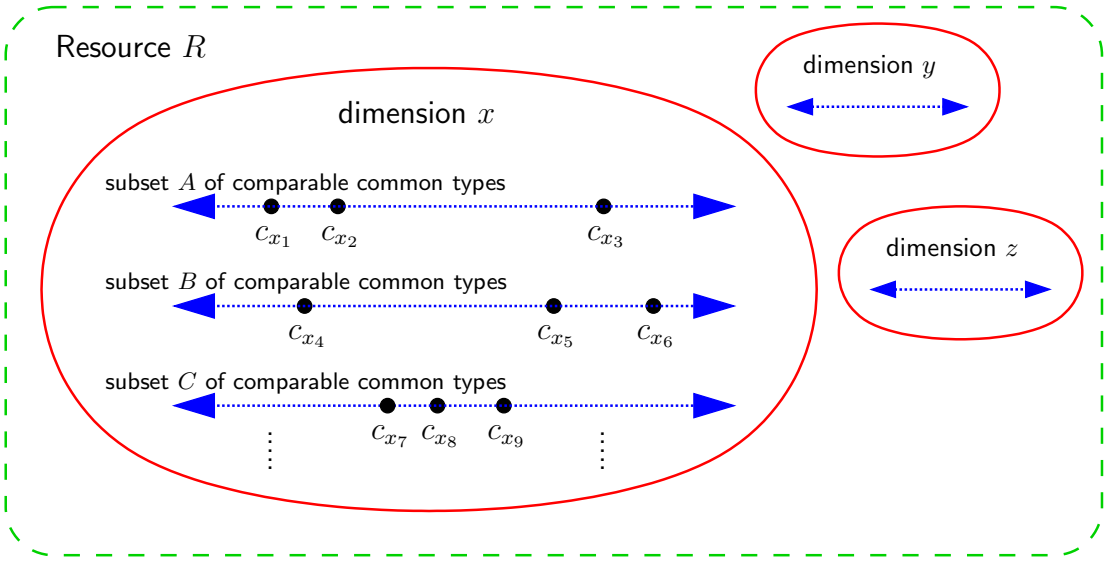

Figure 5.5: *A set of characteristics in a single dimension, but lacking a single common comparable type*

```
<bench type="INT95" > 112  </bench>
<bench type="INT2k" > 0.98 </bench>
<bench type="FP95"  > 49   </bench>
<bench type="FP2k"  > 20.1 </bench>
<bench type="Alpha7"> 32   </bench>
```

Listing 5.4: *Sub-sets of incomparable characteristics within a single dimension.*

5.4 Comparators

The first class of relations to define are those between characteristics using **comparators**. Comparators are relations on pairs or sets of characteristics which return a boolean value. Characteristics are **comparable** if they are in the same dimension and of the same type. This indicates whether or not the characteristics are in the relation's tuple set, which is to say that the relation holds for the set of characteristics in question. Put more formally for binary relations, if c_i and c_j are in the set of characteristics S, and $\mathcal{R}$ is a relation on elements of S (*i.e.* characteristics in S), then the comparator relation $\mathcal{R}$ is defined as the ordered pairs of characteristics where $c_i \mathcal{R} c_j$ holds. This is shown in Equation 5.9. The relation $\mathcal{R}$ has two closely related meanings, depending on the context. The first is that $\mathcal{R}$ is a set of pairs (c_i, c_j), wherever the relation holds. The second is in the case of $c_i \mathcal{R} c_j$, where $\mathcal{R}$ is the actual relation to evaluate between the operands. A constraint on these relations are that they are *transitive* such that $a\mathcal{R}b \wedge b\mathcal{R}c \rightarrow aRc$. The following discussion

focuses on binary relations, but the results in all cases are generalisable to n-ary relations. For a given relation, comparable characteristics either agree with the relation (*i.e.* are found within the relation set), or don't. These results correspond with a *true* or *false* value for the result of the comparator operation. In the event the characteristics are incomparable, the result of the comparison is always *false*.

$$\mathcal{R} \equiv \{(c_i, c_j) | c_i, c_j \in S \wedge c_i \mathcal{R} c_j\} \tag{5.9}$$

The specific operation the relation $\mathcal{R}$ performs is defined by the individual comparator. When $(c_a, c_b) \in \mathcal{R}$, then the comparator will return *true*, otherwise it returns *false*.

A class of comparator called the *Basic Pairwise Comparator* (BPC) is introduced below. This defines a framework for concrete characteristic comparison operations. In object oriented design terminology, the BPC is an abstract base class for comparators which operate on two characteristics. Later sections will introduce specific comparators, followed by further comparator classes (specifically those that perform transformations in an attempt to make characteristics comparable).

5.4.1 Basic Pairwise Comparator

The BPC is a class of binary relation comparator which accepts two characteristics and does not perform any transformations on them before carrying out the actual operation. If either the type or dimension differ, the result of the comparison is *false*. Its behaviour is described and illustrated in Algorithm 5.1 and Figure 5.6. The specific concrete operation is identified as *operation()* in Algorithm 5.1, and for notational reasons is shown as a parameter of the *bpc()* function. The first two conditional branches in Algorithm 5.1 implement this guard, checking for comparability. Appendix C.5 describes this algorithm formally in Haskell.

It is important to note that one comparator may find a set of same-dimension characteristics entirely comparable while another does not. This is due to the fact that the comparator is responsible for characteristic type identification, so two different comparators may identify the same characteristic as having two different types.

5.4.2 Equivalence

The most basic comparator operation is **equivalent**. This operation allows for selection of characteristics from a set, and for testing if one set is a sub-set of another. Algorithm 5.2 defines this operation, as well as in Haskell in Appendix

Algorithm 5.1 Basic Pairwise Comparator

Require: c_a, c_b characteristics
Require: *operation*() the comparison operation to perform, accepting two characteristics as arguments
Ensure: *result* is boolean {*true, false*}
  function BPC(c_a, c_b, *operation*())
    if c_a.dimension $\neq$ c_b.dimension **then** ▷ Dimensions are not the same
      result $\leftarrow$ *false*
    else if c_a.type $\neq$ c_b.type **then** ▷ Types are not the same
      result $\leftarrow$ *false*
    else ▷ Apply the operation
      result $\leftarrow$ operation(c_a, c_b)
    end if
    return result
  end function

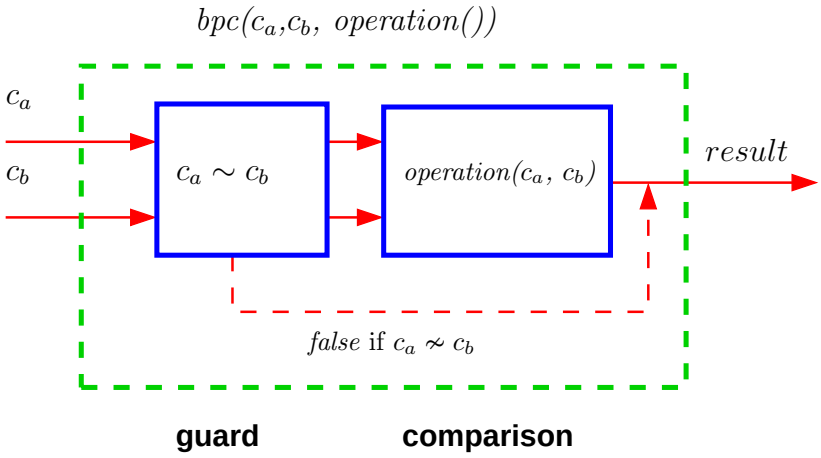

Figure 5.6: *Block diagram for a Basic Pairwise Comparator (*BPC*)*

C.8. Using the syntax from the BPC, this would be called as *bpc(c_a, c_b, equivalent())* which would first check that the two characteristics c_a and c_b were comparable before carrying out the comparator operation *equivalent(c_a,c_b)*.

The condition where $c_a \nsim c_b$ (*i.e.* the characteristics are not comparable) is handled by the "guard" operation in the BPC and would return *false*. Later sections will present the option for guard stages in the comparator to attempt to transform the characteristics to make them comparable. The concept of equivalence is still not strictly defined here, and is left to the equivalence comparator to determine on the basis of the characteristics' values. For example, a given equivalence comparator may consider "`3.00`" and "`3`" to be equivalent, while another does not. A similar

Algorithm 5.2 Equivalence comparator.

Require: c_a, c_b comparable characteristics
Ensure: *result* is boolean {*true, false*}
  function EQUIVALENT(c_a, c_b)
    if c_a.value is equivalent to c_b.value **then**
      result ← *true*
    else ▷ c_a.value is not equivalent to c_b.value
      result ← *false*
    end if
    return result
  end function

example would be the strings "`STOKES-REES`" and "`Stokes-Rees`". An equivalence comparator is bound by the usual constraint that it must be *reflexive* ($(a, a) \in \mathcal{R}$), *symmetric* ($a\mathcal{R}b \rightarrow b\mathcal{R}a$), and *transitive* ($a\mathcal{R}b \wedge b\mathcal{R}c \rightarrow a\mathcal{R}c$).

Algorithm 5.3 describes an equivalence binary comparator which tests for an exact match between the values of the two characteristics, while the negation of that is expressed in Equation 5.10.

Algorithm 5.3 Equal comparator (=).

Require: c_a, c_b comparable characteristics
Ensure: *result* is boolean {*true, false*}
  function EQUAL(c_a, c_b)
    if c_a.value = c_b.value **then**
      result ← *true*
    else ▷ c_a.value ≠ c_b.value
      result ← *false*
    end if
    return result
  end function

$$notequal(c_a, c_b) \triangleq \text{NOT } equal(c_a, c_b) \tag{5.10}$$

5.4.3 Ordering

Ordered sets of characteristics can only be formed if a comparator provides a **total ordering relation** $\mathcal{O}$ of all possible *(type, value)* pairs within a dimension. That is, the total ordering of the dimension's value space. Based on this total order it is possible to evaluate the relative ordering of characteristic sets. Where c_a and c_b are characteristics from the set S, $c_a \mathcal{O} c_b$ indicates that $(c_a, c_b) \in \mathcal{O}$ and that c_a.value $\preccurlyeq$

c_b.value. For simplicity, this relation will be expressed as $c_a \preccurlyeq c_b$. This means there is a total order on S defined by $\mathcal{O}$ and that c_a precedes c_b according to this relation. The name *ordered* is given to this abstract comparator and it is described by Algorithm 5.4. As with *equivalent*, *ordered* is guarded by the BPC to ensure the characteristics are comparable. As this is an abstract comparator, it is not strictly defined. Different ordering relations will produce different orders on the same set of characteristics. For example, given a set of characteristics describing the available storage space for a set of executors, the ordering relations "<=" and ">=" would, in general, be expected to produce two different results. A characterisitic ordering relation is bound by the usual constraint that it must be *reflexive* ($(a, a) \in \mathcal{R}$) and *anti-symmetric* (if $a\mathcal{R}b$ and $b\mathcal{R}a$ then $a = b$). Algorithm 5.5 describes the concrete ordering relation "<=", while Equation 5.11 expresses the ordering relation ">=" in terms of the "<=" relation. Appendix C.8 provides a definition in Haskell.

Algorithm 5.4 Abstract ordering comparator ($\preccurlyeq$).

Require: c_a, c_b comparable characteristics
Ensure: *result* is boolean {*true, false*}
  function ORDERED(c_a, c_b)
    if c_a.value $\preccurlyeq$ c_b.value **then**
      result $\leftarrow$ *true*
    else ▷ c_a.value $\not\preccurlyeq$ c_b.value
      result $\leftarrow$ *false*
    end if
    return result
  end function

Algorithm 5.5 Concrete LTE ordering comparator (<=).

Require: c_a, c_b comparable characteristics
Ensure: *result* is boolean {*true, false*}
  function LTE(c_a, c_b)
    if c_a.value $\leq$ c_b.value **then**
      result $\leftarrow$ *true*
    else ▷ c_a.value $\not\leq$ c_b.value
      result $\leftarrow$ *false*
    end if
    return result
  end function

$$gte(c_a, c_b) \triangleq lte(c_b, c_a) \tag{5.11}$$

Algorithm 5.6 Less than comparator ($<$).

Require: c_a, c_b comparable characteristics
Ensure: *result* is boolean {*true, false*}
  function LT(c_a, c_b)
    if c_a.value $< c_b$.value **then**
      result $\leftarrow$ *true*
    else ▷ c_a.value $\not< c_b$.value
      result $\leftarrow$ *false*
    end if
    return result
  end function

Comparator Name	Relation	Function	Equation
equivalent		*equivalent*(*char*$_a$.value, *char*$_b$.value)	Alg. 5.2
equal	=	*equal*(*char*$_a$.value, *char*$_b$.value)	Alg. 5.3
not equal	$=$	*not_equal*(*char*$_a$.value, *char*$_b$.value)	Eq. 5.10
less than	$<$	*lt*(*char*$_a$.value, *char*$_b$.value)	Alg. 5.6
greater than	$>$	*gt*(*char*$_a$.value, *char*$_b$.value)	Eq. 5.12
less than or equal	$\leq$	*lte*(*char*$_a$.value, *char*$_b$.value)	Alg. 5.5
greater than or equal	$\geq$	*gte*(*char*$_a$.value, *char*$_b$.value)	Eq. 5.11
ordered	$\preceq$	*ordered*(*char*$_a$.value, *char*$_b$.value)	Alg. 5.4

Table 5.2: *Base set of characteristic comparators*

$$gt(c_a, c_b) \triangleq lt(c_b, c_a) \tag{5.12}$$

In summary, the base set of characteristic comparators is listed in Table 5.2. These are also defined formally in Haskell in Appendix C.4.

5.5 Transforming Comparators

As was mentioned earlier, a further class of comparators will attempt dimension and/or type transformations to make characteristics comparable before the comparator operation is performed. Two variations of this class of comparator, called **transforming comparators**, are presented below. The first attempts only type transformations and requires that dimensions are the same, while the second attempts both dimension and type transformations.

5.5.1 Type Transforming Pairwise Comparator

A Type Transforming Pairwise Comparator (TTPC) adds a type transformation stage to the comparator's processing chain. If the dimensions are different, no transformation is attempted and the dimension guard stage will return *false*. If the characteristics are in the same dimension but have different types, the TTPC attempts to transform the characteristics into the same type. It may also be that type transformation is impossible, in which case the result will be *false*. Its logical behaviour is illustrated in Figure 5.7 and Algorithm 5.7. A formal definition can be found in Appendix C.6.

Algorithm 5.7 Type Transforming Pairwise Comparator

Require: c_a, c_b characteristics
Require: *operation*() the comparison operation to perform, accepting two characteristics as arguments
Require: $trans_{type}()$ a built-in function which transforms characteristic values to the same type. Throws *TransformFailed* exception if the type transformation is not possible.
Ensure: *result* is boolean {*true, false*}

```
function TTPC(c_a, c_b, operation())
    if c_a.dimension ≠ c_b.dimension then
        ▷ TTPC does not handle mis-matched dimensions
        result ← false
    else if c_a.type ≠ c_b.type then
        ▷ Attempt a type transformation
        try
            (ċ_a, ċ_b) ← trans_type(c_a, c_b)
            result ← operation(ċ_a, ċ_b)
        except TransformFailed
            result ← false
        end except
    else
        ▷ Simply pass to the operation
        result ← operation(c_a, c_b)
    end if
    return result
end function
```

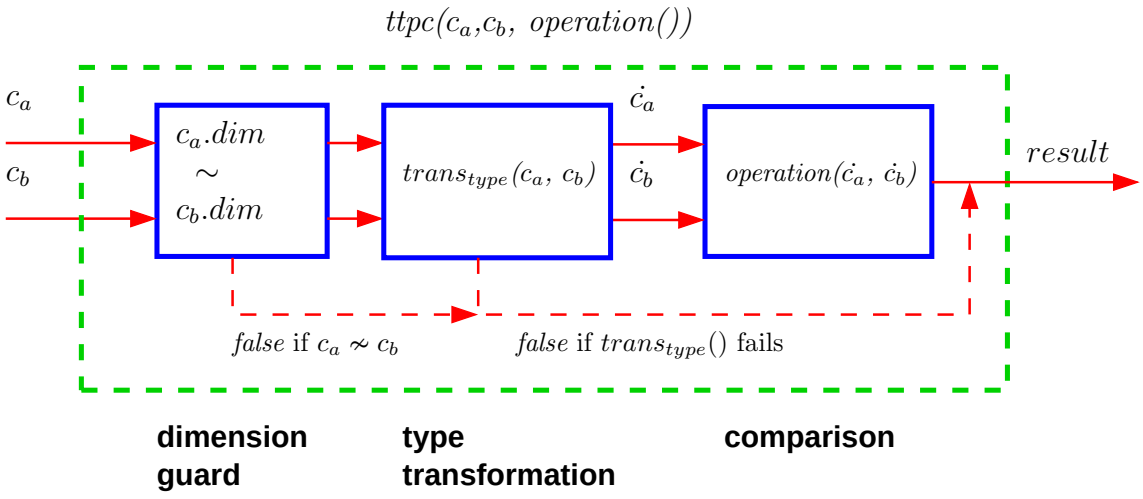

Figure 5.7: *Block diagram for a Type Transforming Pairwise Comparator (*TTPC*)*

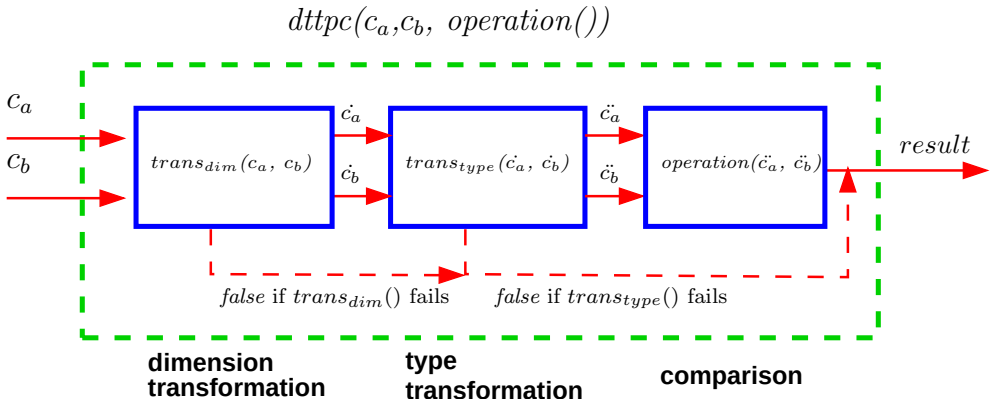

Figure 5.8: *Block diagram for a Dimension and Type Transforming Pairwise Comparator (*DTTPC*).*

5.5.2 Dimension and Type Transforming Pairwise Comparator

A Dimension and Type Transforming Pairwise Comparator (DTTPC) adds a dimension transformation stage to the TTPC processing chain. If the dimensions are different, the DTTPC attempts to transform the characteristics into the same dimension. It may be that dimension transformation is impossible, in which case the result will be *false*. If dimensions are the same, the behaviour is identical to a TTPC. Its behaviour is illustrated in Figure 5.8. It is worth noting that an implementation of a DTTPC may jointly consider the two *(dimension,type)* pairs in order to make

a transformation decision, rather than considering dimension and type sequentially. Algorithm 5.8 describes this class of comparator, and a formal definition can be found in Appendix C.7 in Haskell.

Algorithm 5.8 Dimension and Type Transforming Pairwise Comparator

Require: c_a, c_b characteristics
Require: *operation*() the comparison operation to perform, accepting two characteristics as arguments
Require: $trans_{dimension}()$ a built-in function which transforms characteristic dimensions to the same dimension. Throws *TransformFailed* exception if the dimension transformation is not possible.
Require: $trans_{type}()$ a built-in function which transforms characteristic values to the same type. Throws *TransformFailed* exception if the type transformation is not possible.
Ensure: *result* is boolean {*true, false*}
  function DTTPC($c_a, c_b, operation()$)
    if c_a.dimension $\neq$ c_b.dimension **then**
      ▷ Attempt a dimension transformation
      try
        $(\dot{c_a}, \dot{c_b}) \leftarrow \text{trans}_{dimension}(c_a, c_b)$
        $(\ddot{c_a}, \ddot{c_b}) \leftarrow \text{trans}_{type}(\dot{c_a}, \dot{c_b})$
        result $\leftarrow$ operation$(\ddot{c_a}, \ddot{c_b})$
      except TransformFailed
        result $\leftarrow$ *false*
      end except
    else if c_a.type $\neq$ c_b.type **then**
      ▷ Attempt a type transformation
      try
        $(\dot{c_a}, \dot{c_b}) \leftarrow \text{trans}_{type}(c_a, c_b)$
        result $\leftarrow$ operation$(\dot{c_a}, \dot{c_b})$
      except TransformFailed
        result $\leftarrow$ *false*
      end except
    else
      ▷ Simply pass to the operation
      result $\leftarrow$ operation(c_a, c_b)
    end if
    return result
  end function

5.6 Set Comparison

The next class of relation are those between sets of characteristics. **Set comparators** determine if one characteristic set is a subset of another characteristic set, and provides a basis for comparing resources. Set $Chars_A$ is a subset of set $Chars_B$ if and only if all the dimensions of set $Chars_A$ are found in set $Chars_B$ and if each discrete value of the characteristics in each dimension of set $Chars_A$ are found in the characteristics of set $Chars_B$. The symbol $\subseteq$ has its normal interpretation here, and characteristics can be thought of as triples of *(dimension,type,value)*, therefore $Chars_A$ is the set of characteristic triples in resource A. As a shorthand for R_A.chars $\subseteq$ R_B.chars we use the operator $\subseteq_{chars}$ on the resources as in $R_A \subseteq_{chars} R_B$. Equations 5.13 and 5.14 more formally define this relation, and Equation 5.15 expresses it as a boolean function, where $\equiv_{bool}$ is the boolean equivalent of the set operation, with *true* indicating the set relation holds, and *false* indicating that it does not. It is also described formally in Appendix C.10. In evaluating set membership, the set comparator may utilise a TTPC or DTTPC and perform transformations on the characteristics. Figure 5.9 illustrates two characteristic sets, A_1 and B_1, where A_1 is a subset of B_1.

$$Chars_A \subseteq Chars_B \triangleq \{c_i | \forall c_i \in Chars_A \exists c_i \in Chars_B\} \tag{5.13}$$

$$R_A \subseteq_{char} R_B \triangleq R_A.chars \subseteq R_B.chars \tag{5.14}$$

$$subset(Chars_A, Chars_B) \equiv_{bool} Chars_A \subseteq Chars_B \tag{5.15}$$

A **partial subset** relaxes the condition of all dimensions in $Chars_A$ existing in $Chars_B$, and instead uses $Chars_A \cap \mathcal{D}_{A \cap B}$ which is the subset of characteristics in A along the dimensions shared with B ($\mathcal{D}_{A \cap B}$). The symbol $\subseteq_{part}$ will be used for partial subsets. Equation 5.16 more formally defines this relation, and Equation 5.17 expresses it in functional notation. Figure 5.10 illustrates this. This can also be found in Appendix C.10.

$$Chars_A \subseteq_{part} Chars_B \triangleq \{c_i | \forall c_i \in Chars_A \cap \mathcal{D}_{A \cap B} \exists c_i \in \ Chars_B\} \tag{5.16}$$

$$partsubset(Chars_A, Chars_B) \equiv_{bool} Chars_A \subseteq_{part} Chars_B \tag{5.17}$$

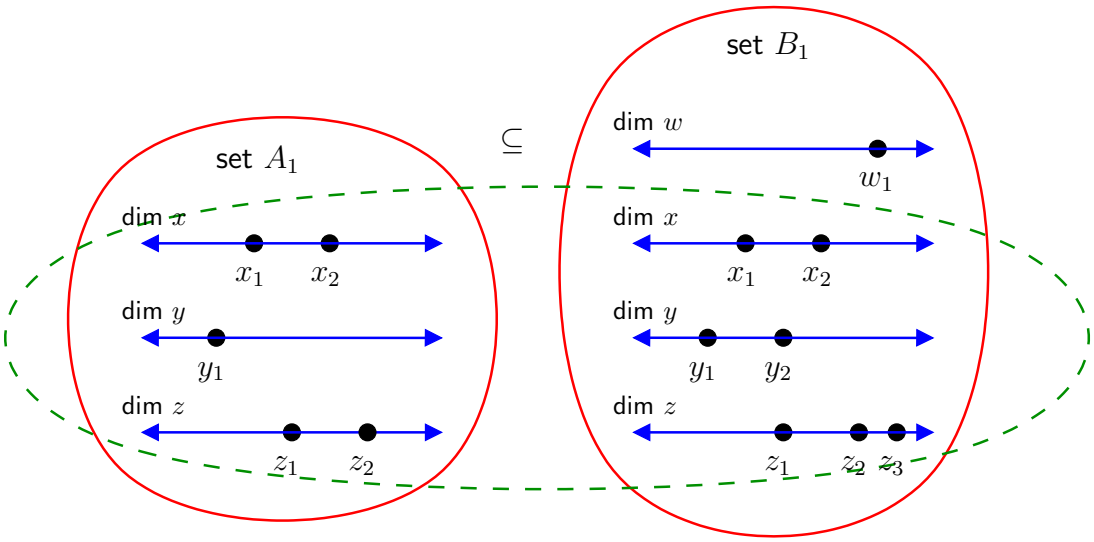

Figure 5.9: *Set A_1 is a subset of B_1*

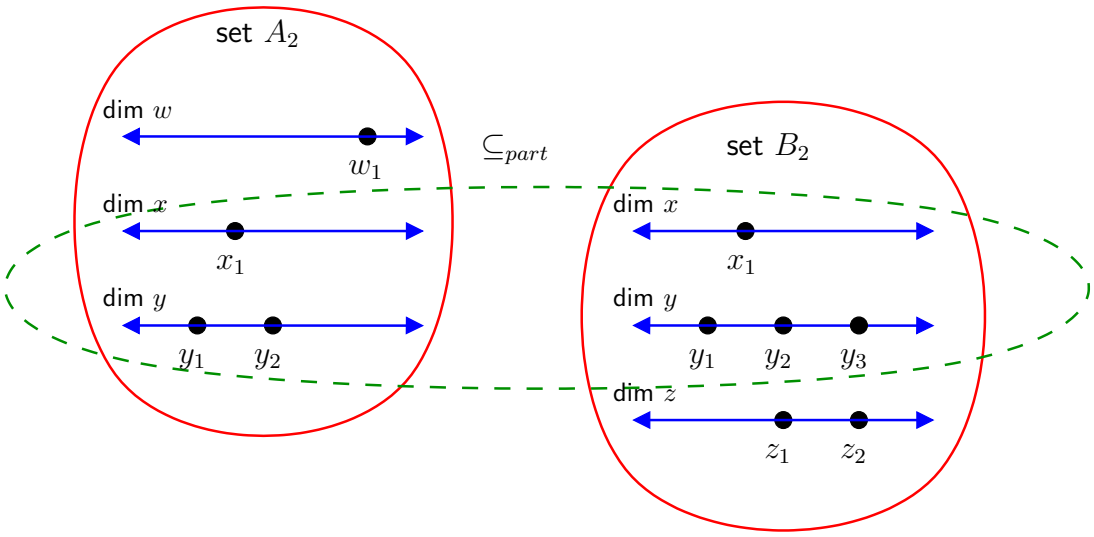

Figure 5.10: *Set A_2 is a partial subset of B_2 (over the shared dimensions x and y)*

The concept of partial subsets is important from two perspectives. The first is that it is often the case that only certain resource characteristic dimensions are of interest, and as such set comparisons which are restricted to those dimensions are valuable. The second is when considering resource templates. By utilising the transitive properties of the subset operator it is possible to significantly reduce the number of resource-pair comparisons which make optimal scheduling an NP-complete problem and therefore intractable for large numbers of resources, as is

common in a computational grid environment. This will be presented fully in Section 8.2.

5.7 Summary

This chapter has provided an overview of the Grid Resource Description Language, and discussed in depth the model for resource characteristics. The relationships between dimensions, types, values, and comparability have all been presented. This model will be extended in the subsequent chapters when looking at *requirements* and *preferences*. Different strategies for transformation of characteristics were investigated, with two broad classes of transforming comparators defined: the TTPC and the DTTPC. A number of abstract and concrete characteristic comparators were also defined. This provided a foundation for considering characteristic sets and set comparison. Examples have been used to illustrate the generic nature of this model for describing any entity as a "resource". In effect, this provides an enhanced name/value pair mechanism for describing resources, with "types" providing some flexibility for transformation and equivalence. This has captured the core features of the ClassAd model in a more abstract way, while also providing a more rigourous description of the properties of characteristics and characteristic sets. This then leads to the work in the next two chapters which cover *requirements* and *preferences* respectively, where *requirements* define the compositional constraints a resource asserts on other resources with which it may be composed, and *preferences* allow for ranking and sub-selection in the presence of multiple possible overlapping compositions.

Chapter 6

Resource Requirements

This chapter extends the concept of resource characteristics into resource requirements. The concept of a resource's "requirement space" is presented, along with a formal definition of resource matching. This definition avoids the need for tri-state logic which is required by Condor's ClassAd mechanism. A variety of matching strategies are described. In particular, the model enables, but does not force, symmetric matching.

In a grid environment, having now described our resources in a RESTful manner using the model for resource characteristics presented in the previous chapter, attention must now be turned to the essential task of marrying sets of compatible resources. This task of composing resources can take on many forms, with the most common being the pair-wise composition of a single user task to an available and compatible CPU in order to execute the task. This chapter will present a general symmetric multi-way resource composition model which can be easily replicated and distributed. It will be argued that this is realisable on an Internet scale, in contrast to the centralised scheduling approaches currently in use.

In a traditional batch computing environment, clusters of CPUs would be assigned to different queues, with each queue containing the same profile/properties. The user would be expected to know which queue met their task's requirements and submit it accordingly, or perhaps an auto-submitter would select a queue from a small set of parameters the user may specify along with the task. Most current grid systems are the same, either with users manually submitting to remote queues of which they are aware, or via "auto-submitters". Manual submission is unreasonable in a large grid environment as a user will be unable or unwilling to track the list of all queues within the system, while grid "auto-submitters" build on the master/slave model used in batch systems where a master head node has full control and full knowledge of the slave nodes and queues within the system – also unreasonable or even impossible in a large grid environment. A single task manager cannot be ex-

pected to know the state of tens of thousands of worker nodes, or even of hundreds of queues, with each possibly servicing thousands of tasks and queueing thousands more. This model, which was discussed in Section 3.8.2, is not scalable. It suffers from the classic NP-complete problem of allocating optimally thousands of tasks to hundreds or thousands of queues, and attempting to centralise management of a massive computational system as defined in Section 2.9. The solution within LCG of utilising multiple Resource Brokers led to two further problems: multiplicative increase in the distribution (or at least replication) of monitoring information to the new Resource Brokers, and users needing to remember which Resource Broker was managing which set of tasks.

One of the goals of this chapter is to present a generalisation of the symmetric resource matching strategy pioneered by the Condor Project. The intention is to enable resource descriptions to be replicated and cached, thus allowing them to be distributed to multiple locations. Once this has been done, Users, Agents, or Services can then browse or fetch these descriptions and attempt to form resource compositions utilising the resource representations. Sequential fetching of resource characteristics or characteristic sets allows for static characteristics to be fetched in advance of dynamic characteristics. In the event static characteristics do not satisfy the requirements of a composition, the dynamic characteristics can be ignored. This is in preparation for a later stage, not discussed in this work, of composition realisation, which is reserving or claiming resources and then utilising them. Furthermore, the requirements model eliminates the tri-state logic of Condor ClassAds, which significantly complicates comprehension and implementation.

6.1 Requirements

The next aspect of the resource model adds the concept of requirements and requirement sets. A resource R may specify an arbitrary set of requirements $Reqs_R$ which constrain the resources with which it may be composed. This is defined in Equation 6.1. Alternative notations are shown in Equation 6.2. This set allows the resource to specify that composition is only permissible if all the specified requirements are met by the characteristics of the resources with which it is being composed. The resource supplying the requirement set will be called the *base resource* and the resource(s) with with which it is being composed, based on its characteristic set(s), will be called the *candidate resource*(s).

The *requirement* model is a sub-class of *characteristic*, and inherits all properties

of a characteristic. The rest of this section will describe how requirements define a range within a dimension, a multi-dimensional value space, specify the relevant comparison operation, and how candidate compositions can be formed. The concept of a *matcher*, which is the analogue of a *comparator*, is introduced.

$$Reqs_R \triangleq \begin{cases} \text{set of all requirements in resource } R \\ \{r | r \in R.reqs\} \end{cases} \tag{6.1}$$

$$Reqs_R \equiv R.reqs \tag{6.2}$$

Requirements extend characteristics by adding a *match* property which specifies a comparator. For example, if *match* is `=`, the requirement is specifying the need for a characteristic with the same value, using the EQUAL comparator. If *match* is `>=`, then the requirement is specifying a value space of all values greater than or equal to the requirement's value which will be checked via the GTE comparator. Any characteristic which falls inside this value space will satisfy the requirement. In this way, requirements are specifying a value range in a dimension, with the requirement that a characteristic of a composed resource exists within this value range. The notation *range(r_i)* will be used to describe this value range of a particular requirement r_i, where "range" has the traditional set theory meaning (*e.g.* from [111], p. 113).

The relation between the requirement value and the characteristic value can be expressed either as a relation *char comparator req* or as a function: *comparator(char, req)*. Because a requirement can be interpreted in a characteristic context, the relation comparators shown in Table 5.2, which take two characteristics as operands, are all valid, with the second operand replaced by the requirement in question. Requirements test if ($char_a$.value, req_b.value) $\in \mathcal{R}$, which is to say "is the (*characteristic, requirement*) pair found in the relation's set". The relation is the comparator specified by the *match* property of the requirement. As a function, the requirement tests if a characteristic is a subset of the requirement's value space, defined by the requirement's (*dimension, type, match, value*) tuple, returning *true* if so, and *false* otherwise. These two interpretations are exactly equivalent.

We can define a set relationship between a characteristic and a requirement, as specified in Equation 6.3, and with the Haskell function `isChReqSubset` found in Appendix C.11. This states that a characteristic is a member of a requirement if the value of that characteristic is a subset of the value space (range) of the requirement. Discussion of the special cases where the characteristic or requirement values are

unspecified is left until Section 6.4.

$$c_i \subseteq r_j \triangleq c_i.\text{value} \subseteq range(r_j) \tag{6.3}$$

Figure 6.1 illustrates the requirement match $c_{x_1} < r_{x_2}$ in the dimension x, where r_{x_2} has a value of x_2 and the comparator is $<$. This means r_{x_2} has an acceptable value space of all values less than x_2. The characteristic c_{x_1} satisfies this requirement since $x_1 < x_2$.

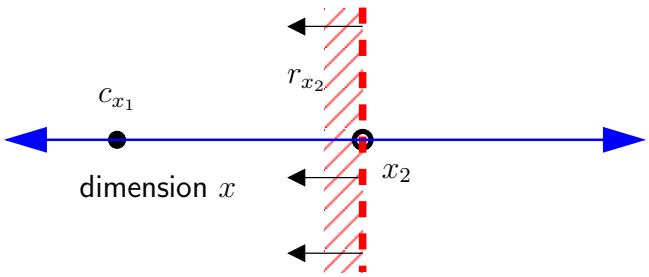

Figure 6.1: *A single requirement r_{x_2} which defines a value space, and a characteristic c_{x_1} which satisfies it.*

Because a single requirement can refer to a range of values in a dimension, it is possible to do comparisons of one requirement to another, based on their respective value spaces. Equation 6.4 defines the condition when one requirement is a subset of another. That is, $r_i \subseteq r_j$ if and only if the range of r_i is entirely within the range of r_j. From this it can be observed that any characteristic which satisfied r_i would also satisfy r_j. These requirement subset relations are described formally in Appendix C.11, specifically with the function `isReqSubset`.

$$r_i \subseteq r_j \triangleq range(r_i) \subseteq \ range(r_j) \tag{6.4}$$

Finally, it is necessary to define the range of requirements with unspecified values. Such a requirement has a range of the entire value space of the requirement dimension. This is stated in Equation 6.5. Requirements with unspecified values are discussed in more detail in Section 6.4.

$$\text{if } r_i.\text{value} = \varnothing \text{ and } r_i.\text{dimension} = x \text{ then } range(r_i) = dim_x \tag{6.5}$$

6.2 Requirement Sets

It is possible to describe relationships between requirement sets in a fashion similar to characteristics (Section 5.6). If each requirement in $Reqs_A$ is a subset of at least one requirement in $Reqs_B$, then $Reqs_A \subseteq Reqs_B$. This is expressed in Equation 6.6, and also as the Haskell function `isReqSetSubset` in Appendix C.11.

$$Reqs_A \subseteq Reqs_B \triangleq \forall r_i \in Reqs_A \exists r_j \in Reqs_B \wedge r_i \subseteq r_j \tag{6.6}$$

$$A \subseteq_{req} B \triangleq A.reqs \subseteq B.\ reqs \tag{6.7}$$

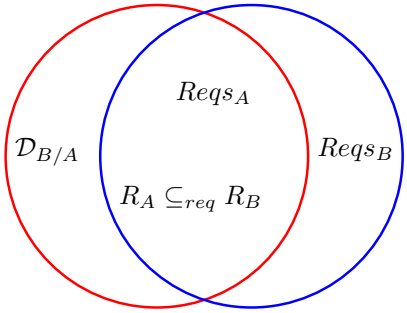

Figure 6.2: *Overlapping requirement spaces for R_A and R_B where $R_A \subseteq_{req} R_B$.*

In practice, however, this definition is not useful. This is based on the fact that a dimension with no requirements is equivalent to a requirement for that dimension which will compose with any resource, regardless of whether or not that resource is defined over that dimension. It is worth noting this is in contrast to an *unspecified value* requirement which requires any characteristic in the dimension, but forbids a candidate resource with **no** characteristics in that dimension. The effect of this is that a resource A with a subset of the requirements in B, as defined by Equation 6.6, may define a subset of acceptable values over the dimensions shared with B, however it will accept any values over all *undefined* dimensions. If B contains requirement dimensions not found in A then B may be more restrictive, meaning over these dimensions A defines a superset of acceptable values (noted as $\mathcal{D}_{B/A}$ in Figure 6.2). As Figure 6.2 illustrates, knowing a candidate resource satisfies the requirements of A provides no knowledge as to whether it satisfies the requirements of B (given $A \subseteq_{req} B$), and *vice versa*.

A simple example of this is shown in Listing 6.1, where $A \subseteq_{req} B$. This example shows how the relation $\subseteq_{req}$ provides no insight into the matching of candidate

```
<A>
    <reqs>
        <mhz     match=">="> 2000 </mhz>
        <ram     match=">="> 512 </ram>
    </reqs>
</A>
<B>
    <reqs>
        <mhz     match=">="> 1500 </mhz>
        <ram     match=">="> 256 </ram>
        <storage match=">="> 500 </storage>
    </reqs>
</B>
<C>
    <chars>
        <mhz     > 3200 </mhz>
        <ram     > 1024 </ram>
        <storage > 200 </storage>
    </chars>
</C>
<D>
    <chars>
        <mhz     > 1500 </mhz>
        <ram     > 256 </ram>
        <storage > 1000 </storage>
    </chars>
</D>
```

Listing 6.1: *The shortcoming of the $\subseteq_{req}$ relation is shown by this example. $A \subseteq_{req} B$ however this provides no additional information about A or B's ability to match resources which the other has matched.*

resources C and D, where C satisfies A but not B, and D satisfies B but not A, so there is no transitivity of the $\subseteq_{req}$ relation.

It is exactly this issue of undefined values which introduces the significant complexity of tri-state logic into the Condor ClassAds Matchmaking mechanism. All logical operators relating characteristics within the "self" ClassAd (equivalent to the base resource in the RESTful model) to the "other" ClassAd (equivalent to the target or candidate resource in the RESTful model) must cater for the condition of the "other" ClassAd not defining that attribute (*i.e.* characteristic), and the interpretation of that condition. It results in the creation of new logical operators `is` and `isnt` and a correspondingly more complicated set of truth tables for what are otherwise well known logic operations. Furthermore, given the user constructs the logic statement for the requirements, it is necessary to add a fourth condition to the result of all logical operations: *error*.

6.2.1 Requirement Space

A variation in the requirement subset definition can address this shortcoming by introducing the concept of *requirement spaces*. Equation 6.8 defines the relationship $A \subseteq_{rs} B$ which specifies that resource A has a smaller (*i.e.* more restrictive) requirement space than B, meaning $\mathcal{D}_{A.reqs} \supseteq \mathcal{D}_{B.reqs}$ (A has the same or more requirement dimensions compared to B) and over every shared dimension $\mathcal{D}_{A \cap B}$ (which is equivalent to $\mathcal{D}_B$) the requirement value space is the same or smaller for the requirements from A compared to those from B. Figure 6.3 illustrates this, showing the requirement space of R_A as a subset of R_B. Under this condition, any resources with characteristics which satisfied $Reqs_A$ would also satisfy $Reqs_B$, therefore it can be inferred that any resource which composes (one-way) with A would also compose with B. This is described formally in the Haskell function `isReqSetRSSubset` found in Appendix C.11.

$$Reqs_A \subseteq_{rs} Reqs_B \triangleq \forall r_j \in Reqs_B \exists r_i \in Reqs_B \\ \wedge\, r_i \subseteq r_j \wedge \mathcal{D}_{A.reqs} \supseteq \mathcal{D}_{B.reqs} \quad (6.8)$$

$$R_A \subseteq_{rs} R_B \triangleq R_A.reqs \subseteq_{rs} R_B.reqs \quad (6.9)$$

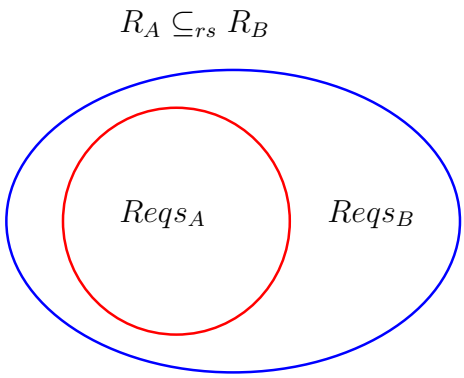

Figure 6.3: *Nested requirement spaces for R_A and R_B where $R_A \subseteq_{rs} R_B$.*

6.2.2 Multiple Requirements Within the Same Dimension

It is possible to have multiple requirements on the same dimension. Figure 6.4 illustrates this. In this case it is possible that one characteristic will satisfy multiple requirements, or that each requirement in the same dimension will be satisfied by different characteristics. Figures 6.5, 6.6 and 6.7 illustrate some of these cases. For the *requirement set* to be satisfied, it is simply necessary that each requirement be satisfied by at least one characteristic, rather than every characteristic in that dimension having to satisfy all in-scope (*i.e.* same dimension) requirements.

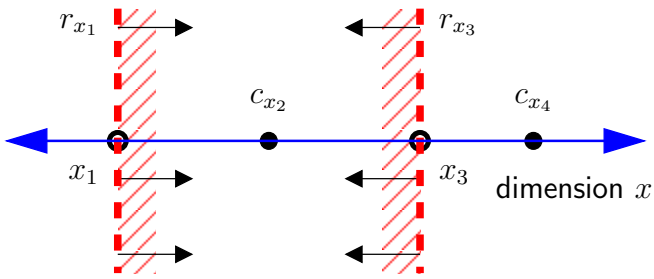

Figure 6.4: *Two requirements in the same dimensions, collectively defining a fixed range of acceptable characteristics, and therefore each satisfied by the same characteristic c_{x_2}. Also, c_{x_4} satisfies r_{x_1} but not r_{x_3}.*

6.2.3 Multi-dimensional Requirement Sets

The union of all requirements defines the overall acceptable multi-dimensional value space for a compositional candidate resource. Figure 6.8 illustrates two requirements in two dimensions. The requirement r_{x_3} in dimension x (interpreted as "$char < x_3$")

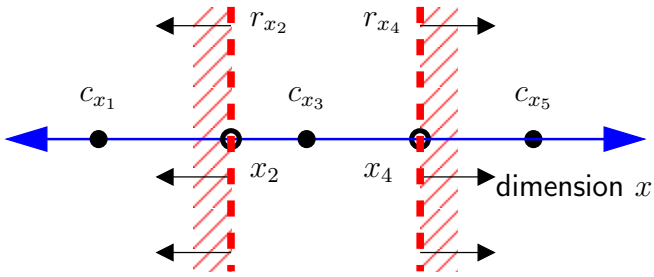

Figure 6.5: *Two requirements in the same dimensions, defining mutually exclusive ranges, however each is satisfied by a different characteristic. r_{x_2} by c_{x_1}, and r_{x_4} by c_{x_5}. c_{x_3} does not satisfy either of the requirements.*

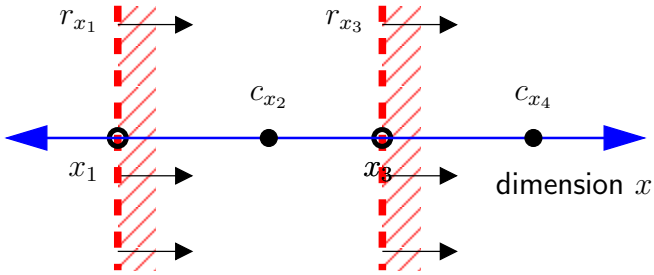

Figure 6.6: *Two requirements in the same dimensions, with overlapping value spaces. r_{x_1} is satisfied by $\{c_{x_2}, c_{x_4}\}$ and r_{x_3} is satisfied only by c_{x_4}.*

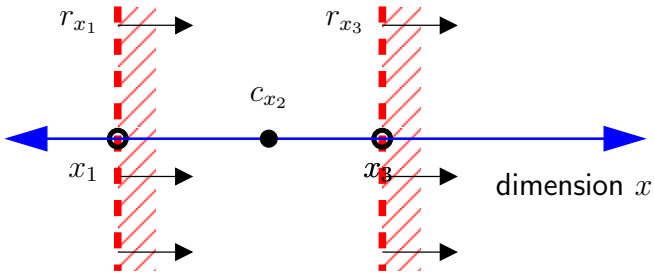

Figure 6.7: *Two requirements in the same dimensions, with overlapping value spaces, but only one requirement, r_{x_1} is satisfied by c_{x_2}, while r_{x_3} is unsatisfied.*

is satisfied by $\{c_{x_1}, c_{x_2}\}$ but not by c_{x_4}, and the requirement r_{y_2} in dimension y (interpreted as "$char > y_2$") is satisfied by $\{c_{y_3}, c_{y_4}\}$ but not by c_{y_1}. Note that the intersection of these two dimension axes is for illustrative purposes only, communicating the concept of orthogonality of dimensions. There is no fixed intersection point, and, in this context, no concept that characteristics in dimension x are related

to the value space of requirements in dimension y (although, of course, *transformers* may map characteristics from x to y). For example, it does not make sense to state whether or not characteristics $\{c_{x_1}, c_{x_2}, c_{x_4}\}$ are "inside" the value space of r_{y_2}, although the figure suggests this is possible. Requirements can only be applied to characteristics with which they are comparable – that is, based on their *(dimension,type)* pair, possibly after transformations by the comparator. As discussed earlier in Section 5.4 any comparison operation with values which are not comparable must return *false*.

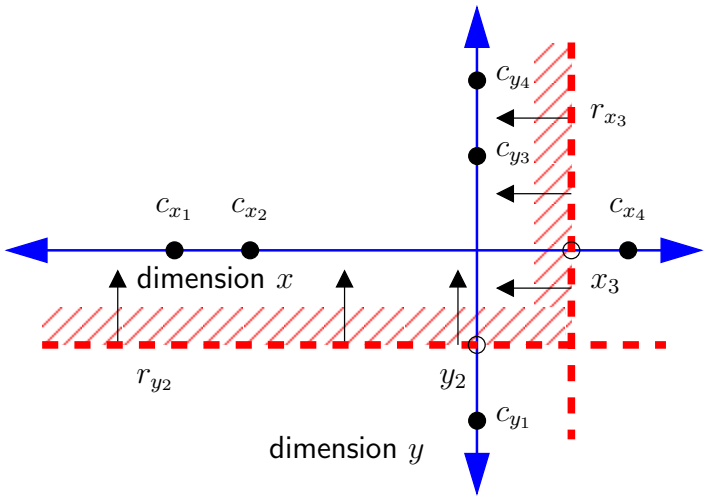

Figure 6.8: *Two requirements in different dimensions, collectively defining a multi-dimensional value space.*

6.3 Matching Semantics

The semantics of composing R_a with R_b are that every requirement in resource R_b must be matched (satisfied) by at least one characteristic in resource R_a. The default match operation is equality (*i.e.* the comparator *equal()* or operator "="). As before, the matching operation will utilise comparators which may perform transformations on either the requirement or characteristic as part of the matching process. The operator $\Rightarrow$ is used to express this resource matching relationship. The equivalent in the functional form is $asymatch(R_a, R_b)$, as shown in Equation 6.10, where $\equiv_{bool}$ represents boolean equivalence, meaning $asymatch()$ returns *true* when the $\Rightarrow$ relation holds and *false* otherwise. This will be discussed further in Section 6.6.1. Appendix C.12 also provides a formal definition of this and the other matching functions in Haskell.

$$R_a.chars \Rightarrow R_b.reqs \equiv_{bool} asymatch(R_a, R_b) \tag{6.10}$$

Matching is a special form of *set comparison* (see Section 5.6) where the super-set of characteristics (the second operand) is replaced with a requirement set from the base resource R_b. It is acceptable to have characteristics in the candidate resource R_a outside of the value space of this set, so long as at least one candidate resource characteristic is found for each base resource requirement. This is described formally in Equation 6.11 and can be expressed more succinctly as $R_a \Rightarrow R_b$, as shown in Equation 6.12. $R_a.chars_{R_b}$ is the set of characteristics in R_a that satisfy the requirements of R_b. The universal quantifier $\forall$ in Equation 6.11 is essential. If it is not satisfied (*i.e.* if any requirement is not satisfied) then the matching characteristic set is empty. The idea is that the match set for a candidate composition contains the selection of characteristics from one resource which have satisfied the requirements of another resource. If the set of requirements have not been satisfied, then there is no match, and the match set must be empty. Partial match sets, which will be discussed in more detail later, provide a mechanism for capturing the set of characteristics which partially match a set of requirements. It is important to observe that a single requirement may define a range of acceptable values, while each characteristic defines at most a single value (unspecified characteristics are an exception to this).

$$R_a.chars \Rightarrow R_b.reqs \triangleq \begin{cases} \{c_i | \forall r_j \in R_b.reqs \quad \exists c_i \in R_a.\ chars \text{ such that } c_i \subseteq r_j\} \\ R_a.chars_{R_b} \end{cases} \tag{6.11}$$

$$R_a.chars \Rightarrow R_b.reqs \equiv \begin{cases} R_a \Rightarrow R_b \\ Chars_{R_a} \Rightarrow Reqs_{R_b} \\ R_a.chars \subseteq_m R_b.reqs \\ asymatch(R_a, R_b) \end{cases} \tag{6.12}$$

The semantics of a requirement are that any single characteristic in its value space will satisfy it, therefore a candidate resource with characteristics outside the base resource's requirements set may be acceptable. Figure 6.9 illustrates this, showing $Chars_{A_1}$ to have additional characteristics in dimensions x and z, but for all requirements in $Reqs_{B_1}$, there is a characteristic in $Chars_{A_1}$. These same two resources are shown in Listing 6.2. At first appearance it seems the concept of

```
<A>
    <chars>
        <x>1</x>
        <x>2</x>
        <y>1</y>
        <z>1</z>
        <z>2</z>
    </chars>
</A>

<B>
    <reqs>
        <x>1</x>
        <y>1</y>
        <z>1</z>
    </reqs>
</B>
```

Listing 6.2: *Two resources A and B which satisfy $A \Rightarrow B$.*

"subset", as presented in Equation 6.3, has been reversed. However, if another requirement set $Reqs_{B_2}$ is considered which includes a requirement w_1 in addition to those in $Reqs_{B_1}$, then it can be seen that $Chars_{A_1}$ does not satisfy this requirements set. $Chars_{A_1}$ has no characteristics in the dimension w and therefore cannot satisfy the new requirement. This is illustrated in Figure 6.10. For this reason, the relation $\Rightarrow$ is synonymous with the relation $\subseteq_m$ (indicating a matching subset), where the subset symbol clearly indicates that – at least some of – the characteristics form a non-null subset of the collective value space of the requirement set. This property is essential when the issue of transitivity of the matching relation $\Rightarrow$ is discussed.

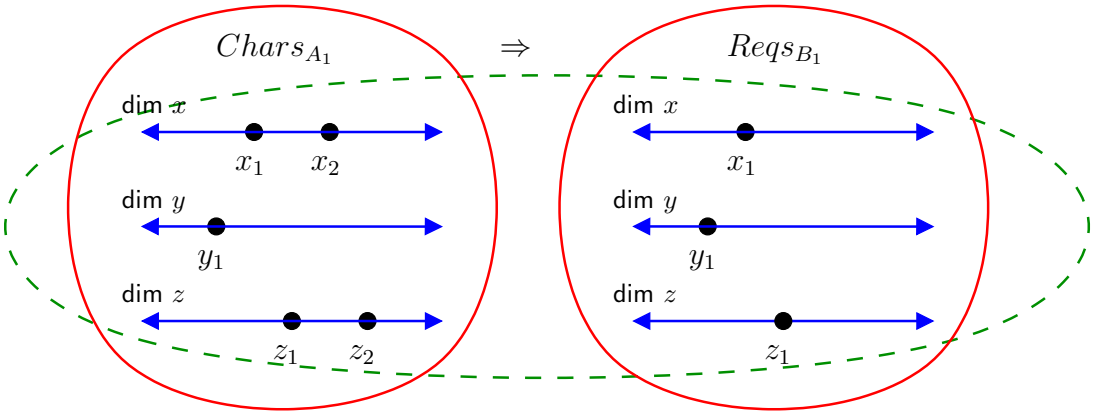

Figure 6.9: *Characteristic set $Chars_{A_1}$ is a matching subset of requirements set $Reqs_{B_1}$*

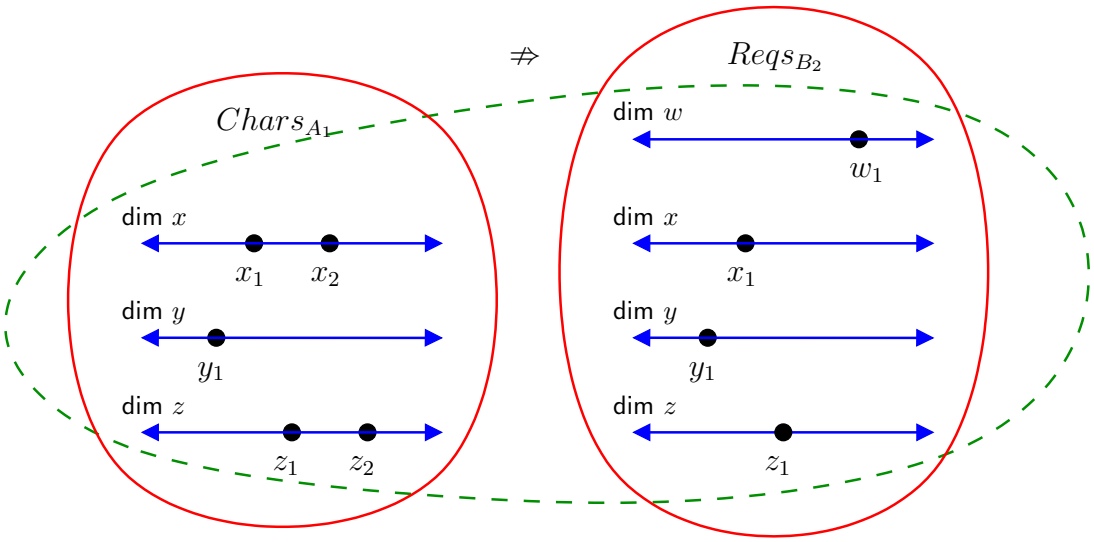

Figure 6.10: *Characteristic set $Chars_{A_1}$ is not a matching subset of requirements set $Reqs_{B_2}$ because it cannot satisfy the requirement r_{w_1}.*

6.3.1 Symmetric Matching

Given that both resources in a match operation may have characteristics and requirements, the concept of symmetric matching is defined in Equation 6.13. This is where R_a is matched to R_b (*i.e.* R_a satisfies R_b's requirements) and R_b is matched to R_a. This will be discussed in more detail later in Section 6.6. The symmetric match operator $\Leftrightarrow$ is used to indicate this, and the equivalent function is *peermatch2()* as shown in Equation 6.14.

$$R_a \Leftrightarrow R_b \triangleq R_a \Rightarrow R_b \wedge R_b \Rightarrow R_a \tag{6.13}$$

$$R_a \Leftrightarrow R_b \equiv_{bool} \text{peermatch2}(R_a, R_b) \tag{6.14}$$

6.3.2 Transitive Properties

Combining Equation 6.11 with the earlier discussion on characteristic and requirement subsets (Sections 5.6 and 6.1), there are two important observations which will be used later in the discussion of *resource templates* in Section 8.2. These are the foundation of an efficient resource matching mechanism. The first concerns characteristic supersets. If $R_a \subseteq_{char} R_b$, then the characteristics of R_b are a superset of those in R_a. If $R_a \Rightarrow R_x$ then the set of characteristics in R_a satisfies the requirements of R_x. It is clear that a superset of those characteristics (namely those in R_b)

will also satisfy R_x, therefore $R_b \Rightarrow R_x$ also holds. This is expressed in Theorem 6.3.1.

Theorem 6.3.1. *If* $R_a \subseteq_{char} R_b$ *and* $R_a \Rightarrow R_x$ *then* $R_b \Rightarrow R_x$

Proof.

$$
\begin{aligned}
R_a \subseteq_{char} R_b &\equiv R_a.\text{chars} \subseteq R_b.\text{chars} && \text{by equivalence (Eq. 5.14)} \\
R_a \Rightarrow R_x &\equiv R_a.\text{chars} \subseteq_m R_x.\text{reqs} && \text{by equivalence (Eq. 6.12)} \\
&= R_b.\text{chars} \subseteq_m R_x.\text{reqs} && \text{by definition (Eq. 6.11)} \\
&\equiv R_b \Rightarrow R_x && \text{by equivalence (Eq. 6.12)}
\end{aligned}
$$

□

The second observation is the requirement analog of this. If $R_x \subseteq_{rs} R_y$, then the requirements of R_x are more restrictive than those in R_y. If $R_a \Rightarrow R_x$, then the set of requirements in R_x are satisfied by the characteristics of R_a, then it is clear that a less restrictive set of those requirements (namely those in R_y) will also be satisfied by R_a, therefore $R_a \Rightarrow R_y$ also holds. This is expressed in Theorem 6.3.2.

Theorem 6.3.2. *If* $R_a \Rightarrow R_x$ *and* $R_x \subseteq_{rs} R_y$ *then* $R_a \Rightarrow R_y$

Proof.

$$
\begin{aligned}
R_a \Rightarrow R_x \equiv R_a.\text{chars} &\subseteq_m R_x.\text{reqs} && \text{by equivalence (Eq. 6.12)} \\
R_a.\text{chars} &\subseteq_m R_y.\text{reqs} && \text{by definition of } R_x \subseteq_{rs} R_y (equation 6.8) \\
R_a &\Rightarrow R_y && \text{by equivalence (Eq. 6.12)}
\end{aligned}
$$

□

6.3.3 Partial Match

The idea of a *partial subset*, introduced in Section 5.6, has an analogue in the context of matching: a **partial matcher**. This is relevant when seeking to compose resources selected from a large pool, where the composition will possibly follow a two stage strategy, with the first stage evaluating compositions based on a limited set of dimensions, and a subsequent stage which evaluates the full set of dimensions or the remainder of the dimensions. Two examples of this are in the presence of dynamic properties, where the initial partial match considers only the static properties and subsequent match operation utilises the dynamic dimensions, and with

resource templates where an initial partial match is done against a representative resource template which does not contain the full set of dimensions of the final "actual" resource. The partial matching operation only operates on the set of dimensions shared with both resources. This is indicated by $\Rightarrow_p$ or in functional form as *pmatch()*. It is defined in Equation 6.15 where $Reqs_B/\mathcal{D}_{\overline{A\cap B}}$ indicates the dimensions **not** shared with A are hidden from B before the match is done (that is, the dimensions $\mathcal{D}_{\overline{A\cap B}}$ are temporarily removed). Hiding these dimensions leaves only the desired requirements in B.

$$Chars_A \Rightarrow_p Reqs_B \triangleq Chars_A \Rightarrow Reqs_B/\mathcal{D}_{\overline{A\cap B}} \tag{6.15}$$

$$Chars_A \Rightarrow_p Reqs_B \equiv \begin{cases} pmatch(Chars_A, Reqs_B) \\ R_b.chars \Rightarrow_p R_a.reqs \\ R_b \Rightarrow_p R_a \end{cases} \tag{6.16}$$

6.3.4 Matching Examples

Listing 6.3 shows two abstract resources C and D where $C \Rightarrow D$. This is illustrated in Figure 6.11. Here some of the requirements specify a range. Note that c_{x_3} and c_{z_4} in resource C are not part of the matching set, but other characteristics in those dimensions satisfy the requirements.

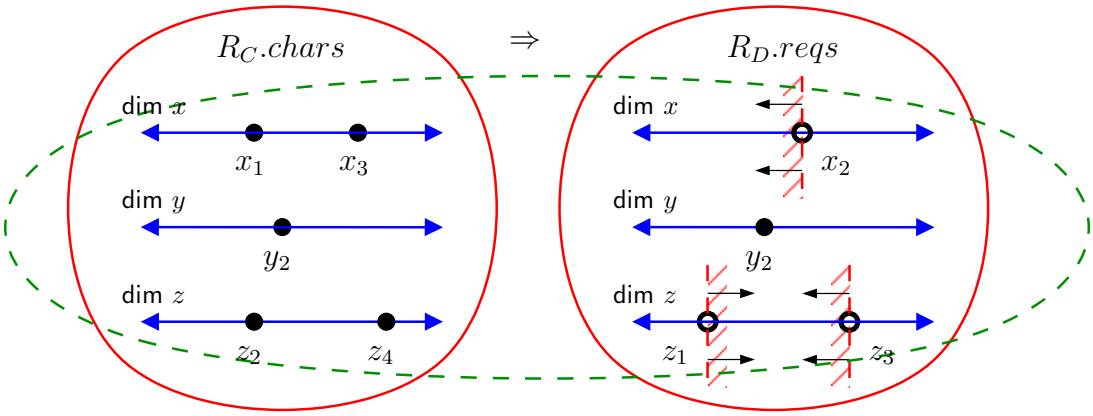

Figure 6.11: *Characteristic set $R_C.chars$ is a matching subset of requirements set $R_D.reqs$*

Listings 6.4 and 6.5 augment the example task and executor descriptions shown earlier in Listings 5.1 and 5.2 respectively. They add requirements to the resource descriptions. Here `pentium` $\Rightarrow$ `testjob1`, meaning the executor `pentium` satisfies

```
<C>
    <chars>
        <x> 1 </x>
        <x> 3 </x>
        <y> 2 </y>
        <z> 2 </z>
        <z> 4 </z>
    </chars>
</C>

<D>
    <reqs>
        <x match="<" > 2 </x>
        <y           > 2 </y>
        <z match=">" > 1 </z>
        <z match="<" > 3 </z>
    </reqs>
</D>
```

Listing 6.3: *Two resources C and D which satisfy $C \Rightarrow D$.*

```
<task name="testjob1">
  <chars>
    <norm_cpu type="hr" > 2   </norm_cpu>
    <mem      type="MB" > 180 </mem>
    <disk     type="MB" > 500 </disk>
  </chars>
  <reqs>
    <os>              linux  </os>
    <software>        python </software>
    <mhz match=">=" > 2500   </mhz>
  </reqs>
</task>
```

Listing 6.4: *Requirements for a task resource*

the requirements of task `testjob1`, however `testjob1` $\not\Rightarrow$ `pentium`. This example will be discussed in the following sections.

```
<executor name="pentium">
  <chars>
    <cpu_type>  pentium </cpu_type>
    <mhz>       2800    </mhz>
    <os>        linux   </os>
    <software>  openssl </software>
    <software>  python  </software>
    <hyperthreading/>

    <bench    type="SI2K" > 0.9  </bench>
    <ram      type="MB"   > 512  </ram>
    <vram     type="MB"   > 1024 </vram>
    <scratch  type="MB"   > 100  </scratch>
  </chars>
  <reqs>
    <norm_cpu match="<=" type="min" > 60  </norm_cpu>
    <norm_cpu match=">=" type="min" > 5   </norm_cpu>
    <mem      match="<=" type="MB"  > 256 </mhz>
  </reqs>
</executor>
```

Listing 6.5: *Requirements for an executor resource*

6.4 Interpretation of Unspecified Values

An unspecified requirement value is interpreted as any point in the *(dimension,type)* value space. This is quite different from a non-existent (*i.e.* undefined) requirement for a particular dimension. Of the base comparators for the *match* property on the requirement, only "`=`" and "`!=`" are open to sensible definitions when the requirement value is unspecified. Equals ("`=`", the default) means any characteristic in the same *(dimension,type)* value space will satisfy the requirement. Figure 6.12 illustrates this case.

Listing 6.6 shows an example of this where the task `encrypt` requires the `openssl` characteristic, with no specified value (*i.e. unspecified*), and the resources `standard-server` and `crypto-engine` both satisfy the requirement, the first with a *unspecified* value for `openssl` and the second with an explicit version number.

The match comparator "`!=`" would imply that no characteristic in that *(dimension,type)* value space can be present. Notice that rather than an operation which acts on each characteristic in isolation, with the requirement that at least one characteristic in the candidate resource matches the requirement (an "any" relationship), this operation is across the entire set of characteristics, requiring that no characteristics of this sort exist (an "all" relationship). To establish this, it is necessary to perform and remember every comparison between the requirement and all

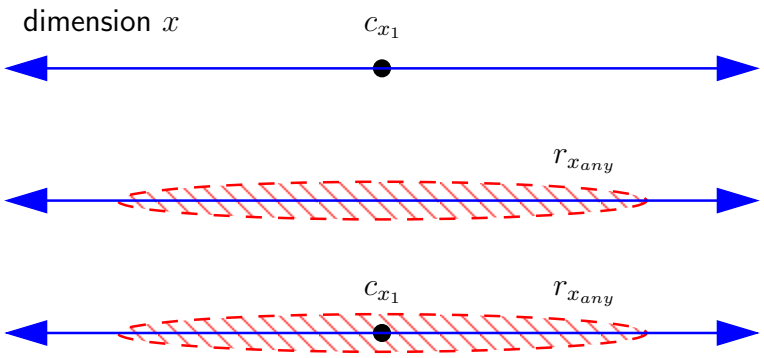

Figure 6.12: *A requirement with an unspecified value is specifying the entire value space of the dimension, therefore any particular characteristic within that dimension will satisfy the requirement.*

characteristics within the dimension. Because this changes the basic requirements matching model, the "`!=`" operation in conjunction with an unspecified value may not be accepted by comparators. Only comparators which are prepared to perform complete set operations (see Section 6.5) may accept this condition. Many of the derivations presented in this work would be invalidated by this sort of operation, for instance those in Sections 6.3, 6.2 and 8.2. In general, then, unspecified value requirements would only be associated with the "`=`" comparison.

A characteristic with an unspecified value will only match a requirement (in the same dimension) which similarly has an unspecified value. This is interpreted as the characteristic existing at some unspecified point in the dimension's value space, and the requirement specifying any value from the full value space of the dimension (that is, the requirement will accept any characteristic within the dimension), therefore the characteristic is a subset of the requirement. Figure 6.13 illustrates this.

If the characteristic has an unspecified valued and the requirement has a fixed value, then the characteristic's value space is not necessarily a subset of the requirement's. Figure 6.14 illustrates this.

Returning to the example in Listing 6.6, if the `encrypt` task were to specify the requirement `<openssl> 0.9.8a </openssl>` then only the `crypto-engine` executor would match (compose) with it, and the unspecified-valued `standard-server` would not.

```
<task name="encrypt">
  <reqs>
    <openssl/>
  </reqs>
</task>

<executor name="standard-server">
  <chars>
    <os>  linux </os>
    <mhz> 3500  </mhz>
    <openssl/>
  </chars>
</executor>

<executor name="crypto-engine">
  <chars>
    <os>      linux  </os>
    <mhz>     1500   </mhz>
    <cpus>    8      </cpus>
    <openssl> 0.9.8a </openssl>
  </chars>
</executor>
```

Listing 6.6: *An example task with an unspecified valued requirement*

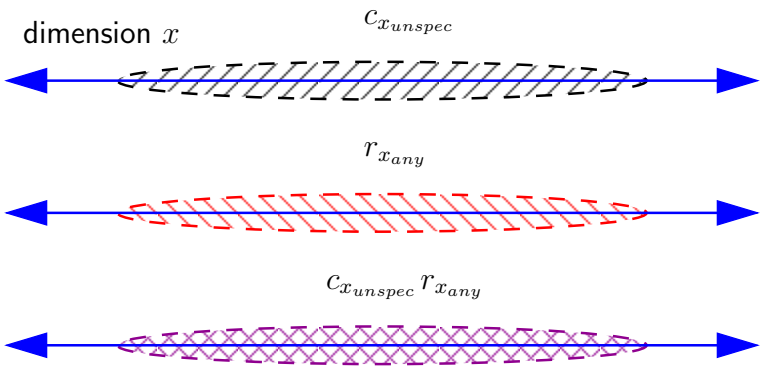

Figure 6.13: *An unspecified value characteristic with a corresponding unspecified value requirement in the same dimension. This illustrates how the characteristic's value space is a subset of the requirement's.*

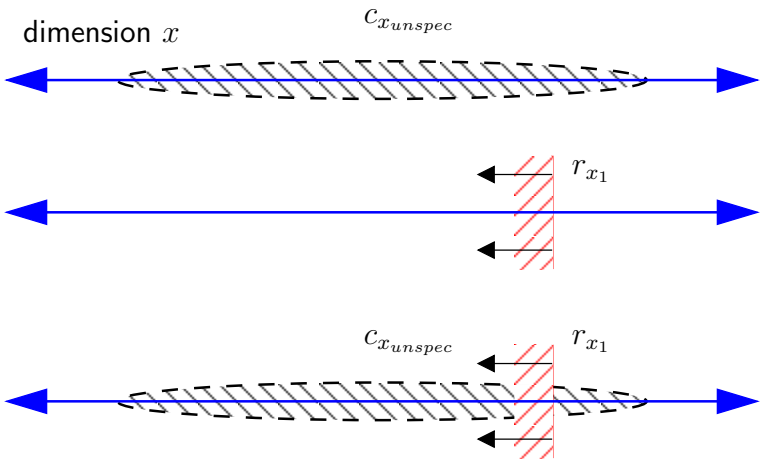

Figure 6.14: *A specific requirement matched against an unspecified value characteristic. Since the characteristic value may be outside of the acceptable range of the requirement, this characteristic does not satisfy the requirement.*

6.5 Complete Set Requirements

The semantics of the base comparators (see Table 5.2) for guiding compositional decisions assume that all requirements must be met by at least one characteristic within the candidate compositional resource. This concept facilitates efficient implementation of resource matchers and is essential for the concept of resource templates, discussed later. In fact, this raises the same problem as interpreting an unspecified value requirement with the "`!=`" comparator, as discussed in Section 6.4. Much of the matching semantics described in earlier sections, as well as the transitive properties and templates discussed elsewhere, fundamentally rely on this assumption. Any behaviour which violates this will invalidate many of the resource composition properties described here.

Even with that consideration, there may be conditions when it is required that all characteristics are satisfied by the set of requirements within a given dimension, rather than simply "at least one". There are two approaches for doing this. The first is to use special matchers which will interpret requirements and the standard operators in this fashion. The second is to define new comparator operations. Matchers or comparators will only be able to operate on these requirements if they understand the operator. Table 6.1 provides the set analogues of the basic comparators listed in Table 5.2. As an example, consider a task which requires 10 CPUs all with over 2 GB of RAM. An executor resource may represent an entire cluster, with an assortment of system configurations, perhaps containing multiple RAM characteristics, some for 512 MB, some for 1 GB, and some for 2 GB. In general a task requirement stating `<ram match='>=' type='GB'> 2 <ram>` would be satisfied by this executor. In contrast, when using set-wise requirements such as `<ram match='set_ge' type='GB'> 2 <ram>` would require **all** `<ram>` executor characteristics to be greater than or equal to 2 GB, so it would not accept an executor publishing **any** characteristics with values less than 2 GB. While this is an interesting and important area for grid resource composition, further discussion is beyond the scope of this work and left as a future research topic.

Comparator Name	Operation	Meaning
set equality	`set_eq`	All characteristics in the set equal the requirement value
set negation	`set_ne`	No characteristics in the set equal the requirement value
set greater than	`set_gt`	All characteristics in the set are greater than the requirement value
set less than	`set_lt`	All characteristics in the set are less than the requirement value
set greater than or equal	`set_ge`	All characteristics in the set are greater than or equal to the requirement value
set less than or equal	`set_le`	All characteristics in the set are less than or equal to the requirement value

Table 6.1: *Comparators for set-wise requirements*

6.6 Matchers

Matchers operate on sets of resources and have custom internal algorithms which determine what sorts of compositions they are attempting to form, and how to form those compositions. Control information may be supplied to guide the composition algorithm. Matchers do not actually compose the resources, but rather propose candidate compositions. For simplicity, the following discussion will generally use the word "compose" to mean "evaluate a candidate composition". *Basic matchers* will take the form described in the following sections and operate exclusively on the input resources' requirements and characteristics.

Single matchers, which are the most basic matchers, take a set of resource descriptions as input and attempt to compose a single composite resource, possibly with an ordered list of alternative compositions from the set. The alternatives may not be mutually exclusive. If one proposed composition is realised it may utilise resources in such a way as to invalidate other alternative compositions proposed by the single matcher. *Multi matchers* propose up to a fixed number of mutually exclusive composite resources from the input set. These compositions can all, by definition, be concurrently realised. The matching function may fail to produce any candidate compositions, or to reach the desired number in the case of multi matchers – this is to say, there is, of course, no guarantee that an arbitrary set of resources will produce a specific number of compositions of a particular form.

By way of example, consider a set of eight resources $\{R_i | i \in [1, 8]\}$. A single matcher may propose the list of five overlapping pair compositions in Listing 6.7, while a multi matcher may propose the three mutually exclusive pairs in Listing 6.8. The first list contains multiple compositions which contain the same resources,

```
[
    (R1,R3),
    (R1,R4),
    (R3,R5),
    (R3,R8),
    (R7,R8)
]
```

Listing 6.7: *A list of candidate compositions from a single matcher. Notice the matches are not mutually exclusive.*

```
[
    (R1,R4),
    (R3,R5),
    (R7,R8)
]
```

Listing 6.8: *A list of candidate compositions from a multi matcher. Notice the matches are mutually exclusive.*

while the second contains each resource at most once. Some resources are not found in any composition.

If the same matching algorithm is used for both approaches, the input set of resources is the same, and the single matcher proposes all possible pair-wise resource compositions, then the inequality in Equation 6.18 holds. This states that the set of matches proposed by a multi-matcher will always be a subset of that proposed by a single matcher, which will in turn be a subset of the cartesian product of the requirement set (that is, a subset of all possible combinations of resources).

$$\begin{aligned} \mathcal{R} &\triangleq \{R_1, R_2, \ldots, R_i | R_i \in \text{set of resources under consideration}\} \\ \mathcal{S} &\triangleq \{(R_i, R_j) | R_i, R_j \in \mathcal{R} \wedge \mathit{singlematch}(R_i, R_j) = \mathit{true}\} \\ \mathcal{M} &\triangleq \{(R_i, R_j) | R_i, R_j \in \mathcal{R} \wedge \mathit{multimatch}(R_i, R_j) = \mathit{true}\} \end{aligned} \tag{6.17}$$

$$\mathcal{M} \subseteq \mathcal{S} \subseteq \mathcal{R} \times \mathcal{R} \tag{6.18}$$

6.6.1 One-Way Pair Match

An asymmetric match between two resources *asymatch(R_a,R_b)* tests if the requirements of resource R_b are met by the characteristics of resource R_a. If this condition holds, then a one-way composition of R_a with R_b is possible. This could also be called *one-way pair composition, hierarchical composition*, or *master/slave composition*. It is illustrated in Figure 6.15.

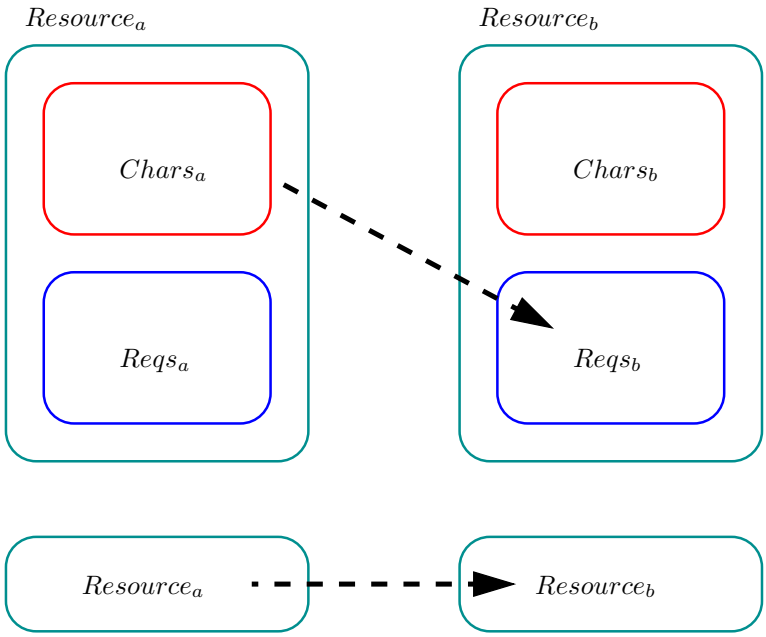

Figure 6.15: *Graphical representation of an asymmetric (one way) match between $Resource_a$ characteristics and $Resource_b$ requirements*

For example, referring back to Listings 6.4 and 6.5, the operation *asymatch(pentium,testjob1)* will attempt to compose the `pentium` resource with the `testjob1` resource, comparing the requirements of `testjob1` to the characteristics of `pentium`. The three requirements are satisfied: i) the operating system is "linux", ii) the software "python" is available, and iii) the processor speed "mhz" is greater than or equal to 2500. On this basis, the one way composition of `pentium` with `testjob1` is permitted.

6.6.2 Hierarchical Match

The extension of this to more than two resources has a single master resource and a set of slave resources. Only the requirements of the master resource are compared against the characteristics of the slave resources. Equation 6.19 describes this, and it is illustrated in Figure 6.16. A hierarchical match is made if every slave resource has a one way match to the master resource.

$$hiermatch(\{R_{slave}\}, R_{master}) \triangleq \\ \text{AND } [asymatch(R_i, R_{master}) | \forall R_i \in \{R_{slave}\}] \quad (6.19)$$

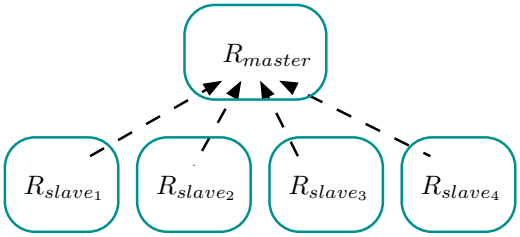

Figure 6.16: *Hierarchical matching of five resources. The master resource applies its requirements to the four slave resources' characteristics. The slaves' requirements are ignored.*

6.6.3 Peer Pair Match

Conceptually, one-way and hierarchical matching are valuable to define, especially as they describe the mechanism utilised by many existing task management systems. Unfortunately, in a grid environment it is unlikely that a resource (or the resource manager) which has specified a set of requirements will be satisfied with it being composed without regard to these requirements. More likely it will reject the so-called "master" resource when the match is realised, or the composition will fail during its lifetime due to the violation of some requirement specified (but ignored) by the "slave" resource (for example, disk, memory, or CPU utilisation). This makes it important to define resource composition mechanisms which consider all resources' requirements. This is what the Condor Project terms *symmetric matchmaking* and is a key aspect to the success of the ClassAd matchmaking strategy developed by Raman, and now used independently in many grid computing projects.

For peer (or symmetric) matching, both sets of requirements must be satisfied by the other resource's characteristics. Equation 6.20 describes this, and it is illustrated in Figure 6.17. This is the resource matching policy which is always in effect with the Condor ClassAd Matchmaker.

$$peermatch2(R_a, R_b) \triangleq asymatch(R_a, R_b) \wedge asymatch(R_b, R_a) \tag{6.20}$$

As an example, considering Listings 6.4 and 6.5 it can be seen that the executor `pentium` has a requirement for a normalised CPU time between 5 and 60 minutes. The task `testjob1` has a normalised CPU time of 2 hours, or 120 minutes, therefore peer composition of these two resources is not possible (even though all the requirements of `testjob1` are met by the characteristics of `pentium`). If `pentium` changed

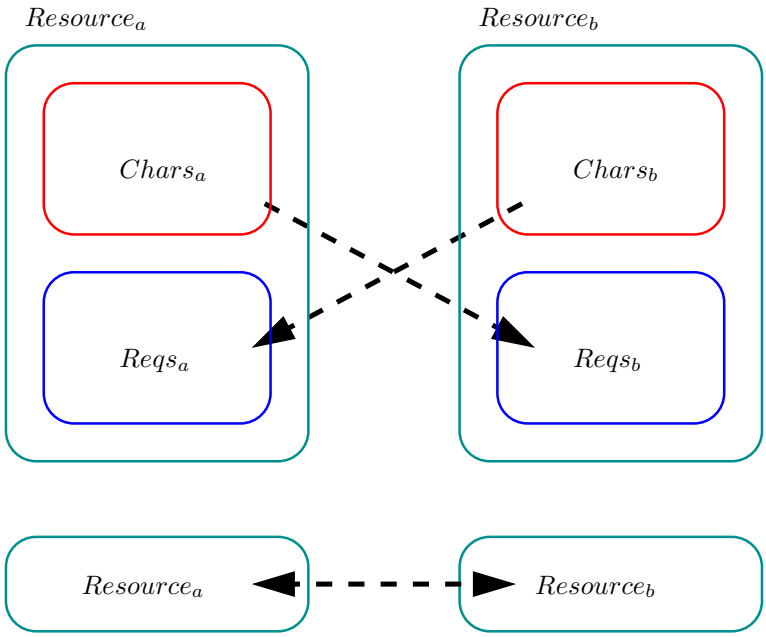

Figure 6.17: *Symmetric (peer) matching of resources. Each resource's requirements are matched against the other's characteristics.*

its upper limit to 480 minutes (8 hours), then all its requirements would be met by `testjob1` and the composition would be allowed.

6.6.4 Multi-Resource Peer Match

This concept can be extended to the matching of an arbitrary number of peer resources simply by including all combinations of resource pairs (except those containing the same resource twice) in symmetric matches. Equation 6.21 describes this and it is illustrated in Figure 6.18.

$$peermatchn(\{R\}) \triangleq \text{AND } [asymatch(R_x, R_y) | \forall R_x, R_y \in \{R\}, R_x \neq R_y] \quad (6.21)$$

Listing 6.9 expands on Listing 5.3 by describing four cricket players and their requirements for team-mates. The players (resources) `James`, `Stephen`, and `Harry` can form a 3-way peer-match (*i.e.* team) as they all satisfy each others compositional requirements. While `Susan` could team up with `Stephen`, she could not form a team with `James` or `Harry` as they have specified they will only play (compose) with male

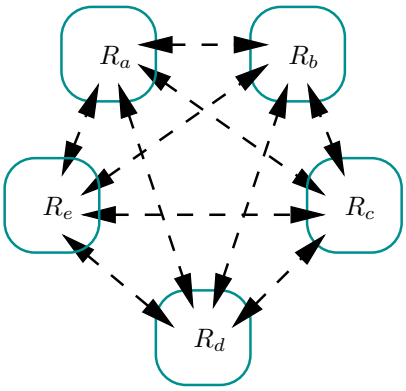

Figure 6.18: *Five resources each matching all their requirements against each of the other resources characteristics.*

players, and she has specified she will only play with those under 26, which neither `Harry` nor `James` are.

```
<person name="James">
  <chars>
    <name>   James   </name>
    <gender> male    </gender>
    <age>    27      </age>
    <weight> 85      </weight>
    <height> 175     </height>
    <sport>  cricket </sport>
    <pos>    batsman </pos>
  </chars>
  <reqs>
    <gender>          male    </gender>
    <sport>           cricket </sport>
    <age match="<=" > 45      </age>
  </reqs>
</person>

<person name="Stephen">
  <chars>
    <name>   Stephen </name>
    <gender> male    </gender>
    <age>    22      </age>
    <weight> 72      </weight>
    <height> 183     </height>
    <sport>  cricket </sport>
    <pos>    bowler </pos>
  </chars>
  <reqs>
    <sport>           cricket </sport>
    <age match="<=" > 45      </age>
  </reqs>
</person>
```

```
<person name="Harry">
  <chars>
    <name>   Harry </name>
    <gender> male    </gender>
    <age>    31      </age>
    <weight> 80      </weight>
    <height> 185     </height>
    <sport>  cricket </sport>
    <pos>    wicketkeeper </pos>
  </chars>
  <reqs>
    <gender>          male    </gender>
    <sport>           cricket </sport>
    <age match="<=" > 45      </age>
  </reqs>
</person>

<person name="Susan">
  <chars>
    <name>   Susan </name>
    <gender> female    </gender>
    <age>    19      </age>
    <weight> 58      </weight>
    <height> 165     </height>
    <sport>  cricket </sport>
    <pos>    bowler </pos>
  </chars>
  <reqs>
    <sport>           cricket </sport>
    <age match="<=" > 25      </age>
  </reqs>
</person>
```

Listing 6.9: *Resource descriptions of cricket players illustrating requirements for team mates*

6.6.5 Aggregated Match

An aggregator is a particular type of matcher which may compose a set of resources in a piecewise fashion. Given a set of resources, the aggregator will select a subset of them to compose. Once composed, the aggregator may then merge the composed resource descriptions in such a way that further compositions with the remaining uncomposed resources are now possible. The aggregator is free to choose the mechanism by which it selects resource subsets for composition, and how it produces the description of the composed resource from the composite resource descriptions (that is, how to merge the characteristic and requirement sets). Figure 6.19 illustrates the aggregated composition of five resources via three piecewise aggregation steps, resulting in a final peer pair composition.

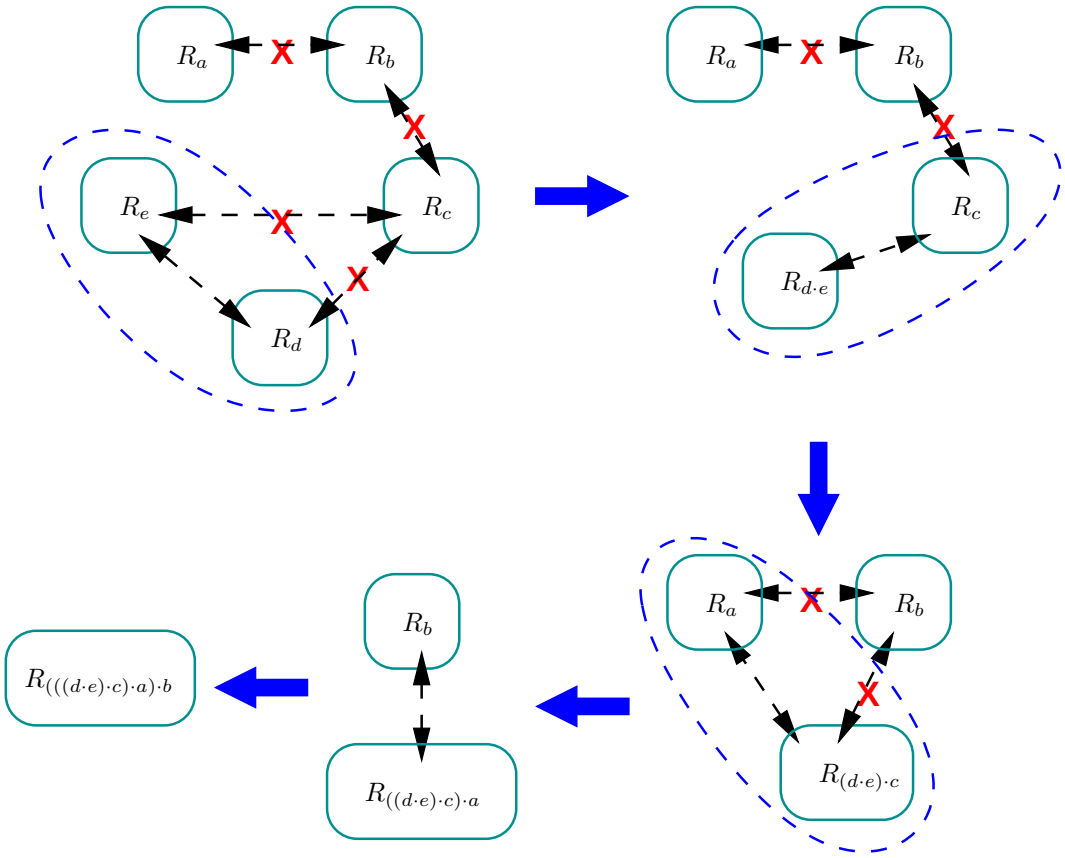

Figure 6.19: *Five resources which cannot initially compose undergo three aggregation steps followed by a peer pair composition. "X"s indicate some of the failed matches. The dashed circles represent the piecewise aggregated resource compositions.*

For example, consider Listing 6.10 which specifies that the task requires two CPUs with at least an SI2K benchmark value of 1.5. An aggregator which was aware of resources described in Listing 5.2 may be able to unify these into an aggregated resource description as shown in Listing 6.11. While neither of the resources in Listing 5.2 could satisfy the requirements of `testjob2`, `aggregated-executor`, formed from `pentium` and `amd`, is able to.

Another common example for multi-lateral resource composition (or "gang matching") is the shared software license problem. A task needs to be composed with a computational resource and simultaneously acquire a floating software license for a particular application. A simple aggregator which was able to perform multiple partial subset matches (see Section 6.3) with different resources may be able to aggregate resources. This aggregated resource would then match the full requirements of the task. Listing 6.12 illustrates this case. The `matlab-license` resource forms a

```
<task name="testjob2">
    <chars>
        <norm_cpu type="hr" > 2   </norm_cpu>
        <mem      type="MB" > 180 </mem>
        <disk     type="MB" > 500 </disk>
    </chars>
    <reqs>
        <cpus>                          2   </cpus>
        <bench match=">=" type="SI2K" > 1.5 </bench>
    </reqs>
</task>
```

Listing 6.10: *A task requiring multiple CPUs for parallel processing*

```
<executor name="aggregated-executor">
  <chars>
    <os>                linux </os>
    <cpus>              2     </cpus>
    <bench type="SI2K" > 2.0   </bench>
  </chars>
</executor>
```

Listing 6.11: *Aggregated executor resource*

partial subset match of the task resource `licensejob1` over the `license` dimension, while the `pentium3ghz` resource forms a partial subset match with the task over the `mflops` dimension. The union of these two dimensions provides full coverage of the requirement dimensions of the task, therefore by simply merging their characteristics an aggregated resource which will compose with the task can be formed.

```
<task name="licensejob1">
  <reqs>
    <mflops match=">=" > 4000 </mflops>
    <license>            matlab </license>
  </reqs>
</task>

<license name="matlab-license">
  <chars>
    <license> matlab </license>
  </chars>
</license>

<executor name="pentium3ghz">
  <chars>
    <mflops> 6000 </mflops>
  </chars>
</executor>
```

Listing 6.12: *Task license and executor resource descriptions for multi-lateral composition of the license management problem.*

6.7 Summary

This chapter has formalised the concept of ClassAd matching from the Condor project and presented an abstract model which avoids implementation details, along with an XML-based syntax for expressing it. Rather than overload a single property, as ClassAds do with the `Requirements` attribute, this has been done by extending (sub-classing) the model for characteristics presented in the last chapter. A further improvement over ClassAds is the elimination of tri-state logic, however the flexible construction of ClassAd Requirements with arbitrarily nested full boolean expressions is lost. GRDL Requirements only currently support conjunction, meaning all requirements must be satisfied for the requirement set to be satisfied. More advanced relations between requirements within a requirement set are theoretically possible, but left as an area for future work. The important concept of *requirement space* has been introduced. A mechanism for specifying compositional requirements of a resource must be accompanied by mechanisms for resolving whether a set of resources are, in fact, composable. This is handled by *matchers*, and a number of matching approaches have been presented, along with examples illustrating the matching process. This model has been implemented in Haskell (see Appendix C), and in Python, in order to validate the operational properties. The matcher concept has similarities to the optimisers implemented in the DIRAC system[3], however they are formalised to use exclusively a set theoretic selection mechanism. Overall, this

model for resource description is a meta-model of those described in Chapter 4, such that each of them could be expressed in GRDL, with the exception of any arbitrary boolean expressions. The final aspect related to the scheduling properties of the resource description is the ranking mechanism used to select from multiple possible resource compositions. This is the topic of the following chapter.

Chapter 7

Resource Preferences

This chapter extends the concept of resource requirements into resource preferences. The difficulties of applying preferences are discussed, and a ranking algorithm is presented, along with a concept of preference set equivalence. Two multi-lateral ranking algorithms are presented, and a small multi-lateral ranking scenario is explored.

It is now necessary to consider the situation where a single composition must be selected from a set of alternatives. This may arise in our distributed REST environment by a User, Agent, or Service querying multiple resource pools for potential compositional resources, or from a single resource set providing multiple possible matches. In some cases, it may be that M compositions must be selected from N candidates, where $M < N$. This chapter will develop GRDL and the REST model to include *preferences* which accommodate this.

7.1 Sub-selection of Resource Composition Alternatives

It is usually the case that multiple possible resource compositions may be formed, either from *single matchers* or *multi matchers*. In general the ranking and selection algorithm can be arbitrarily complex and therefore would often be left to a custom matcher to implement as it chooses. It is, however, possible to specify a simple hierarchical selection algorithm using *preferences* following in the model of *characteristics* and *requirements*. A *preference* is a sub-class of a *requirement*, adding the attribute *rank*. Figure 7.1 illustrates this. A resource R can have a set of preferences $Prefs_R$ associated with it. Equation 7.1 defines this, while Equation 7.2 shows the different ways preference sets can be referred to.

$$Prefs_R \triangleq \begin{cases} \text{set of all preferences in resource } R \\ \{p | p \in R.\text{prefs}\} \end{cases} \tag{7.1}$$

$$Prefs_R \equiv R.prefs \tag{7.2}$$

Matchers are not obligated to consider preferences. For example, a specific matcher may select randomly from the matching set, return all matches, only the first, or only the last. Similarly, a matcher may utilise heuristics or some built-in ranking policy to select from a set of candidate matches. Compositional preferences may be completely independent of compositional requirements, so there is no need for the preference set to be associated with the requirements set, although in practice it likely would be. Conceptually, if preferences are supported by a particular matcher, their objective is to provide guidance on selection from multiple possible compositions. The *rank* attribute on each preference is used to determine the order in which preferences are applied to sorting a set of compositions. Ordering begins with the highest-ranked preference being applied. Where that results in ties between candidate compositions or resources, the next lower-ranked preference may be considered to order this tied set. This will be developed in detail in later sections.

This section describes the semantics of preferences, the algorithm for applying them, and the interpretation of the *match* attribute and preference values.

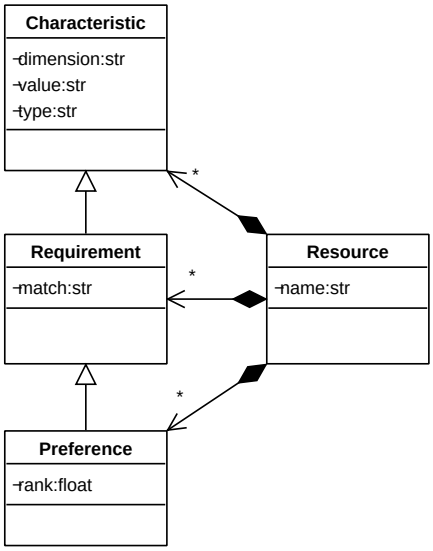

Figure 7.1: *Resource object model. A resource consists of characteristics, requirements, and preferences. Requirements are a sub-class of characteristics, and preferences are a sub-class of requirements.*

7.2 Preference Semantics

While requirements allow a decision on a possible composition through *matching*, preferences allow a process of *ranking* alternative valid compositions. The *matcher* supplies the *ranker* with a set of resources and the preferences to base the ranking on. It may also specify how many resource compositions need to be returned, allowing the *ranker* to stop once that has been achieved, although this discussion will not dwell on such implementation optimisations. An alternative use of a ranker is to screen resources prior to matching, then feed the "most preferred" resources from a pool to the matcher. This is an area for future research and is not discussed further here. In any case, preferences are applied to resources and resource compositions independently of requirements – only resource characteristics are considered when ranking based on preferences.

All preferences must have a *rank* attribute with a numeric value. Lower values indicate a higher rank. Preferences are processed in rank order, and applied to the characteristics of the initial resource set. For any tied preference ranks, the ranker may make an arbitrary and possibly random choice of order between tied preferences. The preference name specifies the characteristic dimension to which the preference applies. As with characteristics and requirements, the ranker is free to apply transformations to the dimensions of preferences or characteristics and their

values when performing the ranking operation.

7.2.1 Ranking Algorithm

The first iteration of the ranking algorithm operates on the full set of resources supplied to the ranker, and considers only the single highest rank preference. Subsequent recursive iterations use lower rank (or "next-tied-rank") preferences in order, and are only applied to sub-sets of resources which were ranked identically by the earlier iterations. If there remain tied resources after all preference ranking has been applied, the ranker may arbitrarily or randomly order the tied resources. Algorithm 7.1 describes the Ranking procedure, although ignoring the "top-N" optimisation parameter which allows early termination once N "best" resources are found.

Listing 7.1 illustrates a single task resource and five executor resources. For simplicity, there are no requirements (meaning all executors will match the task), and only the task specifies selection preferences. It can be seen that the first priority is to order the executors by `cost`, from low to high (*i.e.* preferring lower cost executors). The second priority is to order the executors by `mhz` (which, in this example, represents processor speed and is a rough measure of relative performance), from high to low (*i.e.* preferring faster CPUs). The first preference based on `cost` will result in a resource ordering of: $\langle \{B, C\}_{tied}, E, A, D \rangle$. Only executors `B` and `C` tied, therefore the next ordering based on `mhz` will only be applied to them. This results in the sub-ranking of: $\langle C, B \rangle$, and therefore the final executor resource ordering would be: $\rangle C, B, E, A, D \langle$. `D` comes last as it is not ranked by either of the two task preferences.

7.2.2 Preference Ordering Operators

The *match* attribute, which is mandatory for preferences, specifies the ordering of resources, while the optional preference *value* (and associated *type* attribute) act as a modifier on that ordering. Note that "ordering" may only be evaluating set membership of a resource (*e.g.* "prefer resources located in France"). Table 7.1 describes a base set of preference operators and their interpretation both with and without a value.

The ranking operators are very similar to the requirements matching comparators of Table 5.2, however with different interpretations of *specified* and *unspecified* values. In addition, two distance operators *close* and *far* are added. Most of these operators require the ranker to have a concept of ordering in the value space (`<`, `>=`,

Algorithm 7.1 Resource ranking algorithm, using a hierarchical preference set.

Require: $\{R\}$ resource set
Require: $\{p\}$ preference set
Ensure: $\langle R \rangle_{ranked}$ is ordered according to the preference set
  function RANK($\{R\}, \{p\}$)
    if $\{R\}$.length $<= 1$ **then**
      ▷ One or fewer resources in set, therefore no need to rank
      $\langle R \rangle_{ranked} \leftarrow \{R\}$
    else if $\{p\} = \varnothing$ **then**
      ▷ No preferences, therefore nothing to base ranking on
      $\langle R \rangle_{ranked} \leftarrow \{R\}$
    else
      ▷ Ranked result set of resources is initially empty
      $\langle R \rangle_{ranked} \leftarrow \varnothing$
      ▷ Sort preferences $\{p\}$ in ascending order according to their rank
      $\langle p \rangle_{sorted} \leftarrow$ SORT($\{p\}$, *key*=p.rank, *order*="<")
      ▷ Sort resources according to top preference (may produce tied resource sets)
      $p_{top} \leftarrow \langle p \rangle_{sorted}$.head
      $\langle \{R_{tie}\}_i \rangle \leftarrow$ SORT($\{$R$\}$, *key*=p_{top}.dimension, *order*=p_{top}.match)
      assert $\cup_i \langle \{R_{tie}\}_i \rangle = \{R\}$ ▷ This is a partition of $\{R\}$
      assert $\cap_i \langle \{R_{tie}\}_i \rangle = \varnothing$ ▷ All resources in $\{R\}$ are in exactly one $\{R_{tie}\}_i$ set
      foreach $\{R_{tie}\}_i$ **in** $\langle \{R_{tie}\}_i \rangle$
        ▷ Recursively rank tied subset $\{R_{tie}\}_i$
        ▷ based on remaining preferences $\langle p \rangle_{sorted}$.tail
        $\langle R_i \rangle_{ranked} \leftarrow$ RANK($\{R_{tie}\}_i, \langle p \rangle_{sorted}$.tail)
        ▷ Append ranked list $\{R_i\}_{ranked}$ onto final result
        $\{R\}_{ranked} \leftarrow \langle R \rangle_{ranked}$.append($\langle R_i \rangle_{ranked}$)
      end foreach
    end if
    return $\langle R \rangle_{ranked}$
  end function

```
<task name="prefjob">
  <prefs>
    <cost    rank="1" match="<" />
    <mhz     rank="2" match=">" />
  </prefs>
</task>

<executor name="A">
  <chars>
    <cost> 15   </cost>
    <mhz>  3800 </mhz>
  </chars>
</executor>

<executor name="B">
  <chars>
    <cost> 10   </cost>
    <mhz>  2400 </mhz>
  </chars>
</executor>

<executor name="C">
  <chars>
    <cost> 10   </cost>
    <mhz>  3000 </mhz>
  </chars>
</executor>

<executor name="D">
  <chars>
    <si2k> 2.0 </si2k>
  </chars>
</executor>

<executor name="E">
  <chars>
    <cost> 12   </cost>
  </chars>
</executor>
```

Listing 7.1: *A task specifying a set of preferences to sub-select from a group matched executors.*

Comparator	Sym.	Meaning (value unspecified)	Meaning (value specified)
less than	<	prefer lower	prefer resources closer but less than preference value, then any resources with equal or higher values
less than or equal	<=	prefer lower	prefer resources closer but less than or equal to preference value, then those with higher values
greater than	>	prefer higher	prefer resources closer but but greater than preference value, then those with equal or lower values
greater than or equal	>=	prefer higher	prefer resources closer but greater than or equal to preference value, then those with lower values
equal	=	prefer resources with dimension	prefer *value*
not equal	=	prefer resources without dimension	prefer any other value
close	><	n/a	prefer resource values closer to preference value
far	<>	n/a	prefer resource values further from preference value

Table 7.1: *The base set of preference operators for resource ranking, and their interpretation with* specified *and* unspecified *values.*

> and >=), or a measure of distance (<> and ><). Two (= and !=) only require the ability to evaluate set membership. It is left to the specific implementation to determine a definition of ordering and distance, as well as behaviour in the event that the ranker, for a particular dimension, cannot order or determine distance between characteristics.

7.3 Preference Comparisons

Unlike characteristics and requirements, which lead themselves easily to definitions of individual and set comparison, preferences pose a difficulty in defining comparison operations. An individual preference can be compared with another preference as either *the same* or *not the same*. Other than dimension and type transformations, as discussed earlier, two preferences are the same only if their match operators and rank are identical, and their dimension, type, and value are equivalent (possibly after transformations). If the dimension and value are the same, but rank or match operator differ, then there is no general way of quantifying this difference beyond stating "the preferences are different".

```
r1 : <mhz match=">=" > 3000 </mhz>
r2 : <mhz match=">=" > 2000 </mhz>
```

Listing 7.2: *Two comparable requirements.*

```
p1 : <mhz match="==" rank="1" > 3400 </mhz>
p2 : <mhz match=">=" rank="1" > 3400 </mhz>
p3 : <mhz match=">=" rank="2" > 3400 </mhz>
p4 : <ghz match="==" rank="1" > 3.4  </ghz>
```

Listing 7.3: *Four preferences illustrating the difficulty of preference comparison.*

This is best illustrated by an example. Consider two requirements on the clock speed of a processor, as shown in Listing 7.2. Requirement `r1` specifies a need for a CPU speed greater than or equal to 3 GHz. Requirement `r2` specifies a need for a CPU speed greater than or equal to 2 GHz. Clearly the requirement space of `r1` is a subset of `r2`, meaning `r1` is more restrictive than `r2` or $r1 \subseteq_{rs} r2$.

In Listing 7.3 preference `p1` will select resources with a CPU speed of 3.4 GHz first. Preference `p2` will select resources with a CPU speed of 3.4 GHz or greater first, preferring those closer to 3.4 GHz, then anything below 3.4 GHz. At first, it may seem that $p1 \subset p2$ thereby providing some means of comparing different preferences, however this is not the case. Preference `p1` ranks resources with `mhz` = 3400 first, and then all other resources are tied. The next highest rank preference will be applied to this binary partition of the resouce set: first to the tied resources with `mhz` = 3400, then to all others. In contrast, preference `p2` ranks all resources with `mhz` >= 3400 in order, and all lower valued resources (`mhz` < 3400) are tied. Resources with `mhz` >= 3400 with the same `mhz` value are, of course, also tied. The next highest rank preference is then used to break these ties.

Preferences `p2` and `p3` are the same only on the condition that 1 is the highest ranked preference for the resource containing `p2` and 2 is the highest in the resource with `p3`. In any other circumstance, intervening preferences may result in different orderings of resources.

Finally, preference `p4` may be evaluated as the same as preference `p1`, on the condition that 1 is the highest ranked preference in the resources containing `p1` and `p4`, and if the ranker is able to equate the dimensions `mhz` and `ghz` via a value transformation on the preference. This again reinforces the idea that the only comparison possible for preferences are either *same* or *not the same*.

7.4 Consistency of Preference Set Ordering

In the context of preference sets, A will produce a resource order consistent with B if A has equivalent preferences for all those in B and in the same order, possibly with additional preferences, but only of relatively lower rank. This means A will order resources based on preferences in the same way as B, and possibly assert additional order from preferences on any resources which B would leave as tied. This relation is described using the symbol $\trianglerighteq_{pref}$ and shown in Equation 7.3, where the function *order()* returns the canonical order of the preferences in the set and *min/maxrank()* returns the canonical minimum (or maximum) rank of the preference set. Canonical ordering is the relative ordering of preferences within a set. A more formal definition in Haskell found in Appendix C.14. This is an anti-symmetric relation, so $A \trianglerighteq_{pref} B \wedge B \trianglerighteq_{pref} A \rightarrow A = B$.

$$\begin{aligned} Prefs_A \trianglerighteq_{pref} Prefs_B \triangleq \quad & \forall p_j \in Prefs_B \quad \exists p_i \in Prefs_A, \quad p_i \equiv p_j, \\ & order(Prefs_B) \equiv order(Prefs_A \cap \ Prefs_B), \\ & maxrank(Prefs_B) <= minrank(Prefs_A / \ Prefs_B) \end{aligned} \tag{7.3}$$

7.5 Multilateral Ranking

The discussion of ranking has so far focused exclusively on unilateral ranking for pair composition, where only one resource applies preferences to select from alternatives. When considering multiple matches (from single or multi matchers), multi-resource matching (three or more resources in a single composition), and multilateral ranking (preferences from all resources in a composition contribute to the ranking procedure), it is appropriate to return to the initial observation that this can be arbitrarily complex and therefore best left to specific matcher/ranker implementations to handle in a manner suitable to their environment.

Multilateral ranking is only relevant in the presence of two or more resources on each "side" of the match. If there is only a single resource on one side then this is effectively unilateral ranking, as the resources on the opposing side(s) have no choice over which to apply their preferences – they must always choose the single resource "first". Figure 7.2 illustrates this, with resource t ranking compositions with resources x, y, z in the order specified on the t-side of the joining edge: $\{(t, y), (t, x), (t, z)\}$. Resources x, y, z have no choice but to rank t first in all cases.

The complexity of ranking multiple resource matches can be illustrated by a

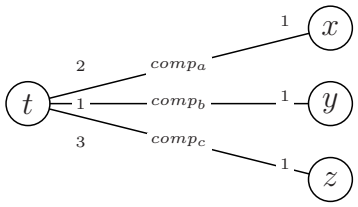

Figure 7.2: *Unilateral ranking with resources on two sides. Notice x, y, z have no ability to specify preferences as there is only one resource t to choose from.*

simple example consisting of only two tasks, t_1, t_2, and two executors, x_1, x_2 as shown in Figure 7.2. The connecting lines illustrate valid matches (compositions).

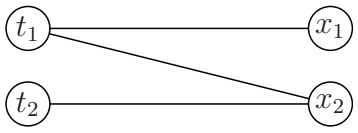

Figure 7.3: *Example of multilateral ranking complexity*

Notice that t_1 can compose with both x_1 and x_2, and that x_2 can compose with both t_1 and t_2. t_2 cannot compose with x_1. In this circumstance, there are two possible overall composition results: $\{(t_1, x_1), (t_2, x_2)\}$ or $\{(t_1, x_2)\}$. There is no general *a priori* way to determine which is preferable. In essence, there is no clear way to combine multilateral preferences. It may be that while the first is *possible*, and has the "benefit" of matching both tasks to both executors, the contrasting benefit of the alternative composition of executing task t_1 on executor x_2 outweighs this. The "preference" in this context is a heuristic which cannot be captured in a generic way by the preference ranking model described here. Figure 7.3 did not include any ranking preferences, in order to highlight the general problem. Considering the four possible rankings which could appear in this circumstance, Figure 7.4 shows thick edges in all cases where both sides of the composition chooses the opposite side "first". The thin solid edges then represent the remaining compositions, while the thin dashed edges represent the compositions which will not be possible after the preferences have been applied.

Three of the ranking orders result in the same composition set $\{(t_1, x_1), (t_2, x_2)\}$, while one ranking results in $\{(t_1, x_2)\}$. This provides insight that the sum or average of the ranking orders may be used to guide the multilateral ranking decision, with

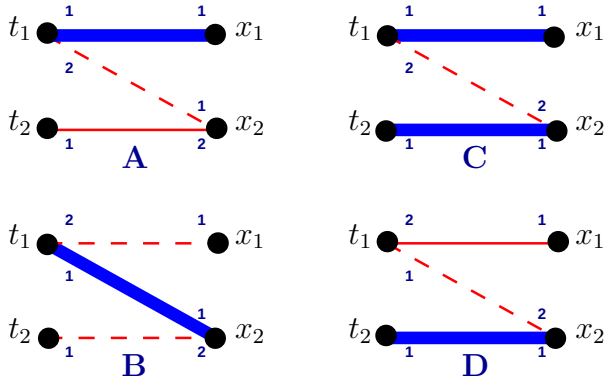

Figure 7.4: *The four ranking combinations for a simple example illustrating the complexity of multilateral ranking.*

minimums being preferable. In general, this is only possible when the preferences of all resources are considered equally important. If this is not the case, it is necessary to apply a bias, however there is no straight forward way to do this. Biasing preferences will, therefore, not be discussed further. These observations suggest two simple multilateral ranking algorithms:

MinSum: Sum the preferences for each side of every match, and select matches in order from lowest to highest sum. This emphasises matching according to preferences, possibly at the expense of leaving resources un-matched. Lower sum matches may invalidate higher sum matches.

Singles: Select matches which contain resources which participate in only a single match. This emphasises maximising the number or resources which are matched, possibly at the expense of "sub-optimal" matches, based on the preferences.

The MinSum algorithm does not necessarily avoid ties. A simple example of this is Figure 7.5 where all four composition orderings sum to 3. The ranker would then either decide arbitrarily or randomly between the the options. It is also unspecified how this algorithm would handle lower-sum matches invalidating higher-sum matches. There is the option to re-calculate preferences after each "top-ranked" match is removed from the set to be ranked.

The Singles algorithm is also not complete, in that it may not select all resource pairs. Instead, Singles could be repeated until no compositions containing a resource in only one composition remained, then an alternate algorithm such as MinSum could be applied once, and then back to Singles. Furthermore, Singles

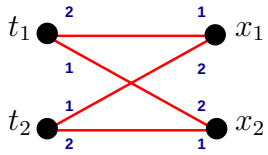

Figure 7.5: *Four possible compositions, all with the same MinSum score of 3.*

may need to choose between multiple compositions which all contain a "single" matching resource. Figure 7.2 is an example of this, where there are three possible compositions containing resources which can only be in one composition. Selecting any one of these will invalidate the other two. Again, it is beyond the scope of this work to discuss specific strategies for anything but the most basic ranking scenarios. These different ranking strategies need to be evaluated in the context of real grid computing work loads. To date they have only been developed in theory.

7.6 Task and Executor Batch Pair Matching and Ranking

It is now possible to discuss a small real world example of M tasks matched to N executors, each specifying characteristics, requirements, and preferences. Listing 7.4 lists the tasks, and Listing 7.5 lists the executors. The "type" modifier on the value is omitted and some common base type for each dimension is assumed. Figure 7.6 illustrates the batch pair matching, with directed dotted lines indicating one-way matches, solid lines indicating pair-wise matches, and numbers indicating the preference order of a resource for the alternative pair-wise matches it can participate in.

```
<task name="t1">
  <chars>
    <norm_cpu> 240  </norm_cpu>
    <xfer_in>  2000 </xfer_in>
    <xfer_out> 100  </xfer_out>
    <image>    300  </image>
    <exec>     reco  </exec>
  </chars>
  <reqs>
    <vo>  physics </vo>
    <eta  match="<"> 12   </eta>
    <cost match="<"> 30   </cost>
    <mhz  match=">"> 3000 </mhz>
  </reqs>
```

```
  <prefs>
    <cost    rank="1" match="<" />
    <mhz     rank="2" match=">" />
    <dataset rank="3" match="==">
        2008/03/22/00234.raw </dataset>
  </prefs>
</task>

<task name="t2">
  <chars>
    <norm_cpu> 960  </norm_cpu>
    <exec>     reco </exec>
    <image>    800  </image>
    <threads>  6    </threads>
    <xfer_in>  5000 </xfer_in>
    <xfer_out> 1000 </xfer_out>
  </chars>
  <prefs>
    <cost rank="1" match="<" />
    <mhz  rank="2" match=">" />
  </prefs>
</task>

<task name="t3">
  <chars>
    <vo>       genegrid </vo>
    <norm_cpu> 60       </norm_cpu>
    <xfer_in>  1000     </xfer_in>
    <xfer_out> 400      </xfer_out>
    <image>    2048     </image>
  </chars>
  <prefs>
    <storage rank="1" match=">" />
    <eta     rank="2" match="><" >
        2008-09-30T12:00 </eta>
  </prefs>
</task>

<task name="t4">
  <chars>
    <norm_cpu> 10       </norm_cpu>
    <temp>     300      </image>
    <exec>     openssl  </exec>
    <xfer_in>  6000     </xfer_in>
  </chars>
  <reqs>
    <openssl match=">="> 0.9.7d </openssl>
    <ram     match=">="> 1024   </ram>
  </reqs>
  <prefs>
    <cost match="<" />
    <ram  match=">" />
  </prefs>
</task>
```

```
<task name="t5">
  <chars>
    <exec> rootkit </exec>
  </chars>
  <reqs>
    <openssl />
  </reqs>
</task>
```

Listing 7.4: *A set of tasks seeking executors*

```
<executor name="x1">
  <chars>
    <cost>    15    </cost>
    <mhz>     3800  </mhz>
    <temp>    10000 </temp>
    <storage> 500   </storage>
    <mem>     1024  </mem>
    <eta>     2     </eta>
    <host>    grid.ox.ac.uk </host>
    <vo>      physics       </vo>
  </chars>
  <reqs>
    <xfer_in  match="<"> 4000 </xfer_in>
    <xfer_out match="<"> 1000 </xfer_out>
  </reqs>
  <prefs>
    <vo rank="1" > physics </vo>
  </prefs>
</executor>

<executor name="x2">
  <chars>
    <cost>    10      </cost>
    <mhz>     3800    </mhz>
    <ram>     2048    </ram>
    <openssl> 0.9.8a  </openssl>
    <vo>      physics </vo>
    <eta>     10      </eta>
  </chars>
  <reqs>
    <exec> openssl </exec>
  </reqs>
</executor>

<executor name="x3">
  <chars>
    <cost> 5   </cost>
    <cpus> 8    </cpus>
    <mhz>  3200 </mhz>
  </chars>
  <reqs>
    <threads match=">="> 4 </threads>
```

```
  </reqs>
</executor>

<executor name="x4">
  <chars>
    <cost>    5      </cost>
    <mhz>     1800   </mhz>
    <ram>     1024   </ram>
    <openssl> 0.9.7f </openssl>
  </chars>
  <prefs>
    <xfer_in  rank="1" match=">" />
    <xfer_out rank="2" match=">" />
  </prefs>
<SS/executor>
```

Listing 7.5: *A set of executors seeking tasks*

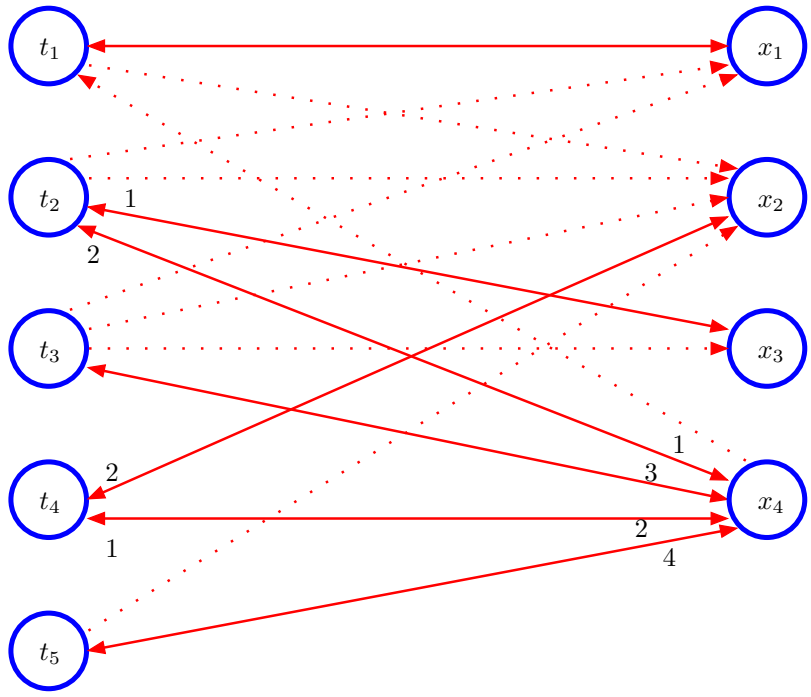

Figure 7.6: *An example of batched pair-wise matching, with 5 tasks and 4 executors. The directed dotted lines indicate one-way matches, solid lines indicating pair-wise matches, and numbers indicating the preference order of a resource for the alternative pair-wise matches it can participate in.*

It is clear from this example that the matcher, even with the ranking information provided by preferences, will require its own arbitrary policy regarding how to select between t_3 and t_5 both of which have only x_4 as their possible match pair. Nonetheless, this illustrates the flexibility and generic nature of resource descriptions and scheduling properties provided by this framework.

7.7 Summary

This chapter has discussed one mechanism for sub-selection of compositions from a set based on a *preferences* model which extends the *characteristics* and *requirements* models presented in earlier chapters. It identifies the difficulties of doing this in a general way and suggests that custom heuristics may often be used in practice in a matcher in order to perform composition sub-selection and ranking. A ranking algorithm is developed, as are two variations which accommodate multilateral ranking (that is, merging preferences from all sides of a match). The motivation has been to provide a REST mechanism by which candidate compositions can be gathered from multiple sources, and then evaluated collectively to select the most appropriate. An example of this may be a task allocation Agent which fetches executor descriptions from the local system, the departmental computing cluster, the university computing cluster, and then from a selection of authorised external clusters. While all task resources managed by the Agent may match to all the executors, the preferences will provide a mechanism to identify the preferred assignment.

Having completed the description of the core features of GRDL, those being the models for *characteristics*, *requirements*, and *preferences*, it is now possible to consider some of the interesting features this model enables. This will be the topic of the following chapter.

Chapter 8

Applications of the Grid Resource Description Language

This chapter discusses various applications of GRDL and possible extensions to the resource description facilities described in earlier chapters. Composition contracts and resource templates are presented, which facilitate scheduling and task management. GRDL validation based on XML Schema and DTDs is presented, as well as the mechanism for extensions to GRDL.

One of the objectives of GRDL is to provide the basis for a RESTful model of grid resources, allowing all resources within the grid infrastructure to make use of a common model for describing their state (*i.e.* representation). Another objective is to provide an efficient and effective mechanism for resource composition, specifically for allocating tasks to executors. These are the two main areas of discussion in this chapter, however other important topics around realising a RESTful grid are also discussed.

The system of resource templates, described in this chapter, provides the foundation for pooling of similar resources, based on a common template. This interacts with the idea of partial matching and partial templates, and provides a significant reduction in complexity of the otherwise NP-complete matching problem, given the degree of heterogeneity in computational grids, from a scheduling perspective, is small and "clumpy". Tasks, in particular, follow this pattern, with groups of tasks all looking similar to each other, but inter-group variation being enormous. This results from users or VOs submitting numerous similar tasks, while different user groups have distinctly different usage patterns. On the executor side, ClassAd-style resource descriptions for the same physical hardware, which are generated each time the resource enters the "available" state, will largely appear the same. For example, over time small variations in available storage space or, in systems with

advanced reservation, the size of the execution time slot may change, or the number of available nodes on a cluster, however other properties will likely remain fixed. The commonality of resources is captured by templates such that resource representations are then registered in those pools which are "headed" by a compatible template. Composition can then be carried out via pool templates, rather than with each individual resource. As experience with DIRAC for LHCb established, there are many cases where the number of template pools M is much smaller than the number of resources N which need to be composed ($M \ll N$), thus making the matching task computationally feasible.

8.1 Composition Profiles and Contracts

This section discusses a mechanism by which information concerning a particular composition of resources can be captured in a standard manner and utilised at later stages in the life cycle of the composition. In a batch system such a level of detail can be entirely implementation specific, as a single task manager will retain "live" stateful information concerning active tasks and the policies which led to a particular utilisation of a resource. Within a grid environment, however, resource compositions transition between many independent systems through their life cycle so that information is not necessarily available or discernable by Services acting on the composition "down-stream" of the stage at which the composition decision was made. This is also true from the perspective that matchers have liberty to interpret dimensions, types, values, preferences, and transformations in their own ways. Users and system administrators often wonder why a particular task has not been scheduled to a particular computing resource, particularly when that resource has free "slots". It is also a specific criticism of Condor that it is often unclear why a particular composition (match) has or has not been made, although this also stems from the complexity of the "`Rank`" expressions used within ClassAds (and the recent addition of the "analyzer" mode to Condor now provides more details of the matching decision). To address this in the RESTful model the results of an attempted composition can be reported using *profiles*.

This provides a richer level of detail concerning the composition than is given by a boolean result of *success* or *failure* for a given composition attempt. Profiles are formed from tuples of characteristic and resource sets from the resources involved in a composition. This requires introducing two new sets: $Chars_{match}$ and $Reqs_{unsatisfied}$. The first, $Chars_{match}$, contains those characteristics from R_a which

match requirements in R_b, as defined in Equation 8.1. This set of characteristics is possibly only a partial match for the requirements of R_b, and can be compared to Equation 6.11 which, by contrast, describes a complete resource match. The second, $Reqs_{unsatisfied}$, contains those requirements in R_b which are unsatisfied by R_a – that is, R_a contains no characteristics which satisfy the subset of R_b's requirements in $Reqs_{unsatisfied}$. This is defined in Equation 8.2. Described in this way, a resource match of R_a to R_b is successful if and only if $Reqs_{unsatisfied} = \varnothing$ (that is, there are no unsatisfied requirements so the set is empty). These relations are also described in Haskell in Appendix C.12.

$$Chars_{match} \triangleq \{c_i | c_i \in R_a.chars, r_j \in R_b.reqs, c_i \subseteq r_j\} \tag{8.1}$$

$$Reqs_{unsatisfied} \triangleq \{r_j | r_j \in R_b.reqs, \forall c_i \in R_a.chars, c_i \nsubseteq r_j\} \tag{8.2}$$

When considering a matcher implementation, the resource matching operation is likely to terminate when the first unsatisfied requirement is found, therefore the $Chars_{match}$ and $Reqs_{unsatisfied}$ sets may not be complete. From a profiling and performance perspective, it is desirable to identify the requirements most likely to be unsatisfied and to test for them first, allowing the match to be terminated with the minimum number of resource/characteristic match tests. Clearly in all cases where the resource match is successful all requirements will be evaluated. Similarly, it is valuable to identify those characteristics which are most likely to satisfy requirements and check them first, again minimising the number of resource/characteristic match tests.

With these two new sets, in addition to standard resource characteristic and requirement sets, it is possible to define three types of profiles:

$\mathcal{P}_{minimal}$ (*minimal profile*): Characteristics which satisfied requirements in a successful composition;

$\mathcal{P}_{full}$ (*full profile*): Characteristics and requirements of all resources participating in the composition;

$\mathcal{P}_{failure}$ (*failure profile*): Requirements which were unsatisfied in the composition.

Profiles only make sense when discussing compositions which are going to be or have already been realised, therefore preferences are not part of profiles.

The minimal profile $\mathcal{P}_{minimal}$ is valuable when the composition is realised and utilised, informing resources what specific quantities of interest have been agreed,

acting as a service contract. It only contains the matching characteristic sets $Chars_{match}$ for the resources in a successful composition. It is defined for a pair match in Equation 8.3.

$$\mathcal{P}_{minimal} \triangleq (Chars_{match_a}, Chars_{match_b})|\ match(R_a, R_b) = true \tag{8.3}$$

Full profiles provide a record of the state of resources at the time of composition. This is valuable if the resources need to change their properties, and allows the effect of the change on the validity of the composition to be checked. The full profile may also supply information regarding the resources in the composition to stages which follow resource matching and scheduling. It consists of sub-tuples of characteristics and requirements for each resource participating in the composition. It is defined in Equation 8.4 for a pair composition. In both these cases the *match()* function is a place holder for the actual matching operation.

$$\mathcal{P}_{full} \triangleq ((Chars_a, Reqs_a), (Char_b, Reqs_b))|\ match(R_a, R_b) = true \tag{8.4}$$

Because a matcher has the freedom to transform and test requirements and characteristics in different ways, it is possible that two different matchers may form different compositions from a set of resources, or may utilise different characteristics in satisfying the requirements of compositions. For this reason it may be desireable to produce both a minimal profile, which captures the exact characteristics a particular matcher selected to satisfy requirements, and a full profile which records the details of the resources in the composition. This is to say that a full profile does not imply exactly the same minimal profile for a composition in all cases – the minimal profile is dependent upon the behaviour of the matcher.

Failure profiles, $\mathcal{P}_{failure}$, provide information regarding which requirements are not being satisfied, thus explaining the failure of a resource to form a particular composition. Failure profiles can also be used to optimise the requirement checking order. They are defined by Equation 8.5 for pair composition.

$$\mathcal{P}_{failure} \triangleq (Reqs_{unsatisfied_a}, Reqs_{unsatisfied_b})|match(R_a, R_b) = false \tag{8.5}$$

8.2 Matching Transitivity and Templates for Resource Composition

One of the critical problems in large scale computational grids is the complexity of task scheduling. Finding an optimal schedule is an NP-complete problem. In this context of symmetric resource composition, the standard task scheduling problem can be re-stated as the composition of task and executor resources. As previously mentioned, the model proposed so far is, at a conceptual level, largely a formalisation of the ClassAd mechanism developed by Raman for the Condor Project[21]. This section will expand on Raman's ideas of selective indexing and indexing optimisation of ClassAd attributes with a *resource template* model. It follows from both the idea found in conventional batch queues which contain tasks with similar characteristics (*e.g.* run time, architecture, memory usage), and from the task queue model implemented in the DIRAC system (see Section 3.5). It will be shown that under certain conditions matching operations are transitive. This will form the basis of the two templating approaches: one based on a match-equivalent template, and one on a shared-composition template.

A resource template defines a dimensional value space of characteristics, requirements, and preferences in exactly the same way as a resource description. Figure 8.1 illustrates the template T as representative of a queue of resources $\{Q\}$, and a candidate composition resource C. How T is representative of each Q, and subsequently how the candidate C is evaluated against T varies between the two templating approaches. It should be noted that resource queues can be generalised to unordered resource pools, the difference being queues will feed resources in a fixed order, while resource pools will provide resources in a random order.

The use of templates and queues can significantly reduce the complexity of matching resources. Rather than examining every set of resources for a match and then ranking between the matched set (a problem which is infeasible in a large computational grid), it rather becomes a matter of querying resource queue templates for a possible composition. As queue templates are expected to be reasonably static, these can be easily replicated and cached. While the total number of queues within such a grid environment is unbounded, the matching process would be configured in such a way as to refer to a (relatively) small set of queues either sequentially until a match is found or all at once, and then select the best available alternative. There are two arguments for why the number of resource queues would be much smaller than the number of queued resources. The first is that many resources will look the

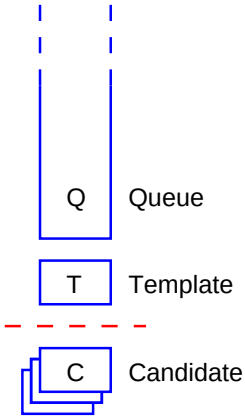

Figure 8.1: *The general template model, showing a queue of resources, a template, and a compositional candidate.*

same or sufficiently similar to be gathered into a single queue. The second is that the use of queues to perform partial matching mean the degree of similarity between a set of same-queue resources can be adapted to suit the desired properties, and certainly means the degree of similarity between common queue resources can be less than an exact match. Once template-based partial matching has successfully been performed, a "real" resource (in fact, its representation) can be fetched from the queue to confirm a full match is possible. Alternatively the queue may only contain partial descriptions of resources, specifically the properties most common for composition or those that are (relatively) static. Prior to realisation of a particular composition, the most up to date resource representation, including full dynamic information, can be fetched from the master resource representation source. In any case, different implementations may utilise a hierarchy of resource queues and queue templates to reduce the number of match comparisons substantially. As an example of this, DIRAC queued tens of thousands of jobs in a handful of ten or so task queues, each representing a "typical" LCG or Computing Centre task queue or computing resource, therefore when Executor Agents made compositional requests by sending an executor resource description to the DIRAC Workload Management System it was only necessary to compare this resource description against (at most) ten queues to determine if a match was possible. The remainder of this section will formally develop this idea.

It should also be noted that queue optimisation is an interesting area of research, however it is not covered by this work. Both DIRAC and the REST model described here support the concept of "optimisers" which actively re-order resources in queues in order to meet certain goals according to some heuristics policy. This concept has

been developed in [83].

8.2.1 Transitivity of the Matching Operation

Combining Equation 5.14 (characteristic subsets) and Equation 6.8 (requirement space subsets) produces the concept of a resource template defined in Equation 8.6. A is a *template* ($\dashv$) of B if all characteristics in A are also found in B and the requirement space of A is a subset of B (that is, A is more restrictive than B). A two dimensional depiction of this is found in Figure 8.2 showing characteristic and requirement value spaces for a resource and its template. If this relation on A and B holds, it can be shown that given a third resource C, where $A \Leftrightarrow C$, that $B \Leftrightarrow C$ also holds. This theorem and its proof are given in Theorem 8.2.1. This is defined in Haskell in Appendix C.15.

$$A \dashv B \triangleq \quad A \subseteq_{char} B \wedge A \subseteq_{rs} B \tag{8.6a}$$

$$= \quad A.\text{chars} \subseteq B.\text{chars} \wedge A.\text{reqs} \subseteq_{rs} B.\text{reqs} \tag{8.6b}$$

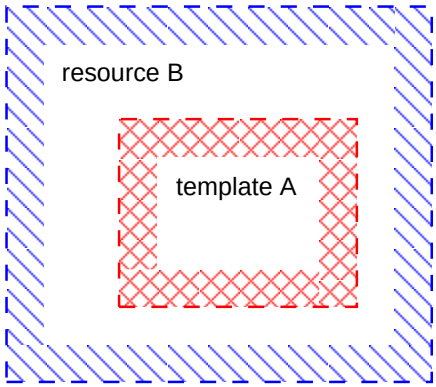

Figure 8.2: *Value spaces for a resource and its template over two dimensions, illustrating how the template has fewer characteristics and a more restrictive requirements space.*

Note that the template relation is not symmetric. From the definition, it can be shown that if $A \dashv B$ and $B \dashv A$ then $A = B$, at least in terms of their characteristics and requirements.

Theorem 8.2.1. *If* $A \dashv B$ *and* $A \Leftrightarrow C$ *then* $B \Leftrightarrow C$

Proof.

$$
\begin{aligned}
A \Leftrightarrow C &\triangleq A \Rightarrow C \wedge C \Rightarrow A && \text{by definition (Eq. 6.13)} \\
&= A.\text{chars} \subseteq_m C.\text{reqs} \wedge C.\text{chars} \subseteq_m A.\text{reqs} && \text{by equivalence (Eq. 6.12)} \\
&= B.\text{chars} \subseteq_m C.\text{reqs} \wedge C.\text{chars} \subseteq_m B.\text{reqs} && \text{by definition (Eq. 8.6)} \\
&= B \Rightarrow C \wedge C \Rightarrow B && \text{by definition (Eq. 6.12)} \\
&\triangleq B \Leftrightarrow C && \text{by equivalence (Eq. 6.13)}
\end{aligned}
$$

□

8.2.2 Match-Equivalent Templates

T_{me} is used to denote a match-equivalent template. This category of template is related to all resources in the queue by the relation $T_{me} \Leftrightarrow Q$. Here the template for the queue represents a resource with which all the queue members could successfully compose. A candidate resource C must be checked to see if it is template equivalent to T_{me}, that is, if $T_{me} \dashv C$. If this holds, then, by Theorem 8.2.1, $C \Leftrightarrow Q$ is known to hold and any resource from the queue may be composed with C.

Match-equivalent queues (*i.e.* those characterised by a template T_{me}) are similar to traditional batch queues, where the template describes the properties of the system (computing resource) on which the queued tasks will run. The queued resources are tasks, and candidate resources are descriptions of available executors which are fetching tasks.

8.2.3 Shared-Composition Templates

In contrast, T_{sc} is used to denote a shared-composition template. In this case $T_{sc} \dashv Q$ describes the relation between the template and all the resources in the queue. Put loosely, the template "is like" the resources in the queue. If $T_{sc} \Leftrightarrow C$ holds for a candidate resource, then by Theorem 8.2.1 $Q \Leftrightarrow C$ is known to hold, and any resource in the queue may be composed with the candidate resource.

Shared-composition queues (*i.e.* those characterised by a template T_{sc}) are useful when multi-way compositions must be formed. The queue template can then be used as a representative handle for the resources in the queue. Any multi-way composition which involves the template will also be satisfied by the queued resources.

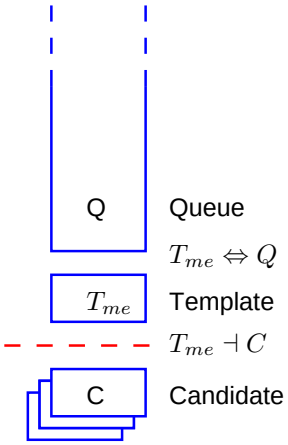

Figure 8.3: *The match-equivalent template model.*

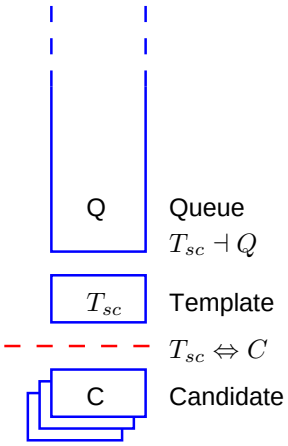

Figure 8.4: *The shared-composition template model.*

8.2.4 Template Preferences

Broadly, there are three strategies for handling preferences with templates:

1. *No template preferences* – Templates contain only characteristics and requirements. Preferences are evaluated between the queued resources and the candidate resource. Note that the candidate resource is free to apply preferences to the template if trying to select between different queues.

2. *No queued resource preferences* – The templates' preferences are used by queue managers to select an optimal candidate resource. All queued resources accept the preference ranking of the template.

3. *Equivalent order* – Since preferences are an extension of requirements, the same templating rules as requirements can be applied, with the addition that the ranking of the preferences must form an equivalent order of either the queued resources (for shared-composition templates) or the candidate resource (for match-equivalent templates). This is equivalent to adding the term $B \trianglerighteq_{pref} A$ to Equation 8.6 (see Section 7.4).

Some out-of-band mechanism is required to describe which strategy a particular queue expects to utilise, however this is not discussed further here.

8.3 XML Schema Validation of GRDL

Many of the examples of GRDL provided so far have used an XML syntax. The use of XML, as compared to other syntaxes, has three clear advantages: wide spread tool support, powerful schema validating parsers, and ease of extensibility. This section describes the XML structure and schema.

GRDL is divided into modular XML Schema files, which are used in layers. The first layer provides a common set of property types, unit abbreviations and groupings, as shown in Listing B.1 of Appendix B. Table 8.1 summarises these types.

The second layer defines the actual dimensions which are known by the system. In general, these would be specific to a given system and would therefore be customised. Listing B.2 of Appendix B provides a selection of possible resource dimensions taken from a number of existing grid description languages. Each dimension is defined for each property class (*i.e.* characteristic, requirement, preference) with a *type category*, which constrains its value space and its *type* attribute. For example, a dimension `cpu_speed`, which is in the *Frequency* type category would have three element definitions, one for each type:
`CharacteristicFrequencyType`,
`RequirementFrequencyType`, and
`PreferenceFrequencyType`.

The dimension elements are defined within the context of their property class element (*i.e.* `<chars>`, `<reqs>`, or `<prefs>`). The next layer is for extensions to the resource model, and allows arbitrary XML structures and types to be created. The fourth and final layer provides the overall resource description, composed of the various components defined in the earlier layers. Listing B.3 in Appendix B shows this construction for four resource types: {`resource, executor, task, storage`}.

This layered approach encapsulates changes to the model, with successively higher layers expected to be changed more frequently. Figure 8.5 illustrates this layering.

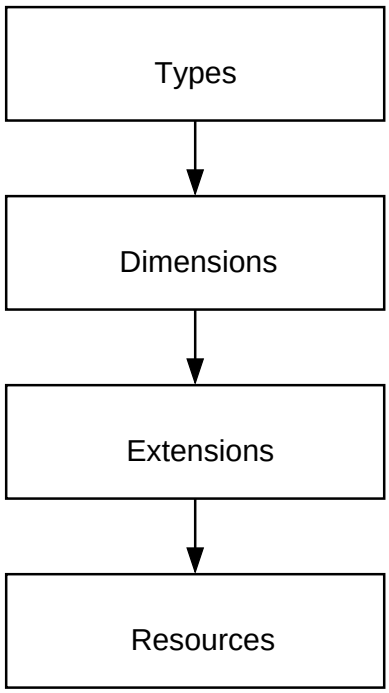

Figure 8.5: *Layered schema design for GRDL.*

The base schemas do not make use of XML Namespaces. This is to simplify the use of GRDL and for comprehension. In an environment where namespacing was required, it is simply a matter of including the null-namespaced schema into a namespaced schema, as is shown in Listing B.4 of Appendix B.

For completeness, a DTD XML definition for GRDL has also been created, however due to the limitations of DTDs it is only possible to create one property definition for each dimension, namely that for the *preference* property, and then have it as a matter of policy that the relevant attributes are not used for *characteristics* or *requirements*. This can be found in Appendix B.5 Listing B.5.

Time	
ns	1e-9
us	1e-6
ms	1e-3
s	1
min	60
h	60*60
hr	60*60
hour	60*60
d	60*60*24
day	60*60*24
w	60*60*24*7
week	60*60*24*7
month	60*60*24*7*30
mon	60*60*24*7*30
yr	60*60*24*365
year	60*60*24*365
Binary	
B	2**0
KB	2**10
MB	2**20
GB	2**30
TB	2**40
PB	2**50
Frequency	
Hz	1e0
KHz	1e3
MHz	1e6
GHz	1e9
THz	1e12
Metric	
f	1e-15
p	1e-12
n	1e-9
u	1e-6
m	1e-3
base	1e0
K	1e3
M	1e6
G	1e9
T	1e12
P	1e15

Table 8.1: *Categorised GRDL types (labels and values).*

8.4 Consuming Characteristics

Characteristics are not "consumed" when they satisfy a requirement, although it is clear that an advanced feature of resource matching would be able to consider "consumable" characteristics, for example in the case of file storage space, memory, or CPU power. This aspect is not discussed further here, as this work has focused on the problem of resource description and resource composition. The act of consuming characteristics would only take place when resources were committed to a composition, when a composition is realised, or at another later point in the lifecycle of a resource. Whether or not a characteristic was consumed by a composition, and how a composition affected the consumption of the characteristic would be very dependent on the definition of particular dimensions. The model proposed here is meant to support resource dynamism and "characteristic consumption", but does not attempt to propose any particular approach to these topics.

8.5 Block Reservation

The GRDL model easily supports advanced block reservation utilising characteristics and requirements. Single node reservation can be handled by adding world clock characteristics to resource representations, describing the start and end times when they are available, or when they require the compositional resource to be available. The common operation of executor advanced reservation can be handled by executor representations defining a latest start and earliest finish time window through two requirements, and a task having characteristics which fix a particular start and end time for the reservation. An intelligent resource manager could then backfill or re-publish any unused time blocks for the node. In the case of multi-node reservations, an aggregator could be utilised to gather a set of executor resources together which will collectively satisfy the block reservation, or again the resource manager may publish the availability of blocks of nodes and then intelligently republish un-utilised sub-blocks if a particular reservation does not utilise all available nodes.

8.6 Priority Queues

One of the key features of this RESTful model is the flexibility for creating different resource queue hierarchies. It is usually the case that resource owners are biased

towards who gains access to their resource, whether that resource is a task, an executor, storage, or software. Although the concept of priority queues is common in batch scheduling systems (*e.g.* Maui[91], IBM LoadLeveler[112], PBS[104], and Platform LSF[92]), in all cases these are single system implementations of scheduling policies for task queues entirely managed within a single operational domain. The RESTful model facilitates a scenario where resource owners create a hierarchy of resource queues, such that the most important users are given preferential access while the wider community is considered last. This model also allows resource users to hedge their bets by placing their resource usage requests into multiple queues, in order to improve the chances they are successfully composed in a timely manner. This concept of priority queues is fundamental to achieving Key Goals 1 and 2 (Scalability and Reliability).

Furthermore, the generality of the RESTful priority queue model decouples it from either the execution resource manager (*i.e.* cluster management system, such as PBS), or a client side task submitter or task pool. The first generally represents a "push" style task management mechanism controlled by a compute centre, where a cluster manager has complete control over slave nodes and selects tasks from a queue to "push" them onto a particular compute node. The second is more typical of "pull" style scheduling where tasks have "late-binding" to a particular compute node (and possibly even physical computing site, in a grid domain), and instead are held by users or a third party (*e.g.* the LCG Resource Broker) and allocated to compute nodes "on demand" when a compute node makes a "pull" request. DIRAC demonstrated the efficiency and benefits of "pull" scheduling (see Section 3.9), while "push" scheduling has benefits in terms of the planned management of resources and for advanced reservation. By decoupling and generalising the priority queue model into this RESTful approach, "push", "pull", and third-party scheduling options are all available.

This situation is best described by way of two examples. The first centres on the manager of a University's departmental computing cluster. The manager has a requirement to prioritise access to a select group of individuals within the department, and therefore sets up a task queue which only they can access and submit to. Next comes other departmental users, then the wider university community, then a selection of international "grid" communities with which the cluster is shared in the event it is otherwise idle. The most direct priority round-robin approach would exclusively select tasks from higher priority queues in a sequential manner until those queues were empty. Once empty, the next lower priority queue would be considered.

If the highest priority queues were constantly full, the lower priority queues would never be serviced. This could be refined to provide a rolling percentage share of the computing cluster to other queues, such that a limited number of tasks were selected from lower priority queues even when higher priority queues contained tasks. By judicious use of access control policies, this model would allow a cluster manager to allocate their resources according to the desired usage policy.

The second example considers an experimental collaboration which has a large number of computational tasks to be executed over an extended period of time. Two strategies exist for these users. The first is to manage their own task queue and then to notify various computing centres of its availability. Depending on the relationship with the computing centre, and the centre's other demands, the task queue would be placed in a hierarchy or weighted round-robin for the computing centre to fetch tasks from. The collaboration could control access to the task queue to limit it to "authorised" computing centres. The alternative is to spread the tasks to many different remote task queues to which the collaboration has access. The tasks would be submitted to the highest priority queue possible at each computing centre. This may rely on using privileged "local-user-only" queues which members of the collaboration local to the site have access to. This maximises the likelihood tasks will be executed in a timely manner while limiting them to "trusted" computing centres. If a particular task is sent to multiple queues, then the first queue to claim it "wins", while the remainder will discover the queued task is "stale" and has expired when they attempt to realise the composition. The collaborations task manager may also choose to withdraw replica queued tasks once one has successfully been claimed or begins executing. In fact, both these strategies can be used simultaneously, which is an inherent feature of a RESTful model where the representation is meant to be replicated and is not, itself, the "resource" in question.

8.7 Security

Security is a key issue in computational grids, where resource interaction passes between different administrative regions over a federated and geographically distributed network. GRDL and the overall model presented here make no assumptions or in any way limit the approach to resource security. Non-repudiation of GRDL resource representations can be achieved through the use of digital signatures, and in an XML context this is particularly convenient as the XML Digital Signature standard allows different sections of GRDL to be signed by different identities. Ac-

cess control to queues is the other major point for security, and again this is left to implementations to specify. No work has been done on approaches for specifying interaction policies within GRDL, and only limited work (not reported here) on how various identities can be specified, transported, and utilised via GRDL. This is part of the larger work on an overall RESTful grid process model which remains in the early stages. In this respect, security is the least satisfied Key Goal described in the introduction.

8.8 Summary

This chapter has discussed a number of applications of GRDL and the REST model for a computational grid. It has described the value of composition profiles, and established the theoretical basis for resource queues and templating. The strategy of priority queues and templating combine to overcome the NP-complete limitation of optimal resource scheduling and centralised workload management, providing a scalable distributed architecture for grid resource management. While discussion of integrating many other desireable features of computational grids into the REST model are possible, these are left as areas for future work.

Chapter 9

Future Work and Conclusions

9.1 A Scalable Computational Grid Architecture

As was described in the introduction, this dissertation is a work of two parts: one part presenting the design and operational experience of a large computational grid infrastructure, the other part presenting an abstract and general model for grid resource descriptions as the basis for a RESTful grid, where this second part is motivated by experience from the first.

The success of the DIRAC infrastructure and strategies employed in its implementation contrasted with the difficulties of utilising LCG thus revealing the need to investigate alternative grid architectures. DIRAC was deemed to be successful because it employed a simple, distributed, service oriented architecture which followed a "high throughput" scheduling model utilising a "task-pull" approach rather than the traditional "high performance" model which utilises "task-push". A further contributing factor to its success was the emphasis on robustness through various mechanisms: asynchronous messaging between services, a light weight client, "pilot" jobs, service watchdogs, dynamic configuration, dynamic software deployment, and decoupled file transfer queues. It also provided insight into the use of instant messaging as a light weight communications infrastructure for grid resources. Finally, it served as a testing ground for real deployment of OGSI/GT3 Grid Services, and established many short comings both with the OGSI approach and the available implementations.

In contrast, it was shown that LCG could provide a usable distributed computational infrastructure, albeit with a failure rate exceeding 30%. Even this was only achieved through careful management and augmentation of the LCG system. The centralised Resource Broker was a major bottle neck and source of failures,

while the overall infrastructure did not provide the level of control, programmatic APIs, or logging to make it easily usable. It was a unanimous decision of the LHCb computing team that the attempts to create an omnipotent and omniscient Resource Broker were impossible to realise, and an architecture which required this could not be part of a long term, robust, grid solution. The distribution of state information throughout a grid is such that it is impossible to maintain in a single location complete, consistent, and timely details of all grid resources. Furthermore, an architecture which apparently could only be realised by an opaque, monolithic, homogeneous system was a very long way from the vision of a grid infrastructure of heterogeneous hardware and software, with federated administrative domains and a plethora of dynamic virtual organisations[15]. This redoubled the emphasis on an ARDA-like services model[47] which facilitated multiple implementation, transparency, and extensibility.

Analysis of the LHCb Data Challenge 2004 results confirmed the high degree of heterogeneity in a grid across all dimensions, for example network bandwidth, CPU architecture, processor loading, and memory distribution. It highlighted the need for task logging throughout the task lifecycle and the need for a handle to the executing task in order to debug or recover stalled tasks. It also demonstrated the reality of a large computational grid with tens of thousands of queued tasks, tens of thousands of executing tasks, hundreds of sites, and thousands of nodes. Contrary to common distributed computing systems which focus on the management of a single or small numbers of concurrent processes, a grid environment must support operations on thousands of concurrent processes. Issues around security, roles, virtual organisations, and delegation were also discovered. It is essential that users can operate with a selection of identities and roles at different times or with respect to different tasks. LCG did not make any of this easy, if it was possible at all.

The work on DIRAC and experience from DC04 motivated a number of further refinements to the DIRAC architecture. This dissertation focused on one of them: outlining a REST model for computational grids, which emphasised a common representation of grid resources, and in particular refined the Condor ClassAd model for symmetric resource matching. The REST approach was radical in that it emphasised the description of the resources within the system while saying little about the operations on those descriptions (representations). This was in contrast to an Object Oriented approach which hides the description and focuses on interface and behaviour, or a Service Oriented Architecture which describes the system in terms of interacting services. Both XML and HTTP/HTML have benefited from this REST-

ful approach. In a grid environment it is argued that resource consumers will wish to act on a resource description in arbitrary ways, therefore the most effective aspect to specify is a common resource description, rather than a common service interface. While a sketch was provided of an overall RESTful grid architecture, the work here was limited to the aspects concerning generic resource description and composition. In particular, the REST principles described in Section 2.3 have guided the RESTful grid model in the several ways:

1. All entities can be described in a common way as resources (GRDL).
2. All resources can interact in a common way, via the set theoretic compositional model.
3. Grid resources have a hidden resource state, with a public representation of that state.
4. Content negotiation to present a representation of a resource relevant to the client or customised based on the client request.
5. Cacheable representations.
6. Dynamic representations.
7. Client-driven rendering or interpretation of resources.
8. Stateless services to transact representations.
9. Elimination of any specific services or "resource stores", enabling decentralisation and therefore scalability.

This built on DIRAC in a number of ways. While DIRAC contained the principles of distributed services and flexible resource matching, it was, nonetheless, focused on a Service Oriented Architecture with a central task queue, and performed explicit task/executor matching, as opposed to general resource composition. Furthermore, the key entities within the system were the services and the architecture consisted of the service configuration and service APIs. In contrast, the REST model focuses on the description of resources within the system and making those resources directly accessible. DIRAC's proto-RESTful features were its stateless interactions, replicable services, independent clients, and simple/light-weight API.

The model has been developed from a strong foundation in set theory in order to benefit from the properties of sets. The model consisted of characteristics, requirements, and preferences, with each inheriting the structure of the more basic property, thus providing a generic basis for interaction between the different property classes. The semantics of each of these properties was explored in depth. The model eliminated the complexity of tri-state logic, used in ClassAds for requirements, and generalised the concept of "type" and attribute comparability via equivalence classes and a formal structure for the transformation of properties. The entire model was also presented formally in Haskell in Appendix C.

One of the greatest features of the model was the ability to combine priority queues with resource templates, thus allowing resource representations to be replicated to multiple queues in order to maximise the likelihood of finding a "good" match, and reducing the computational effort of evaluating candidate compositions through the use of templates. This allows a relatively small number of distributed resource queues to hold, in principle, an unlimited number of resources, and for the scheduling problem to be rephrased as matching through resource composition with queue templates. These concepts were not present in Condor, nor in any other grid scheduling system, which generally struggle with more than $10e4$ queued tasks and rely on a single central scheduling engine, or a set of centralised schedulers. This new strategy was initially investigated in DIRAC and have been formally presented in this work. The variations of asymmetric, symmetric, pair, multi-way, and aggregated resource matching were all developed providing a comprehensive range of resource composition alternatives. This fits into a framework of comparators, matchers, and rankers which can be used to evaluate resource compositions and select from valid alternatives.

Finally, a selection of applications of GRDL were discussed. These covered issues such as composition contracts, validation, extension, reservation, and security. They illustrated some of the properties which can be derived from the formal set theory model of GRDL.

In total, the REST model presented here provided the foundation for RESTful generic computational grids in a Condor ClassAds style, however with a significantly more robust relationship between characteristics, requirements, and preferences. The simplicity and consistency of the model makes it easily realisable thus facilitating multiple implementations. By decoupling resource descriptions from a particular service interface or scheduling strategy, the REST paradigm allows resource representations to be replicated and cached which, when combined with pri-

ority queues and the transitive properties of templates, enables scalable distributed resource scheduling. A complete implementation utilised in a production environment is required to fully validate this model, however the concept has been verified both in practice via the DIRAC implementation and performance results from DC04, and established in theory via the work presented in the later half of this dissertation.

9.2 Examination of Key Goals

Of the six Key Goals, only Security (Key Goal 6) has been minimally addressed, however the model has made efforts to take an approach which will allow various trust models to be utilised. By avoiding any centralised services the REST model is inherently scalable (Key Goal 1), while replication of resource representations also increases reliability (Key Goal 2). Within DIRAC, reliability was achieved through various concrete mechanisms – replication, fail over, caching, and retries – as well as a simple service oriented architecture, thus reducing the risk of bugs and complex behaviour. Usability (Key Goal 3) is very subjective, however the fact that the DIRAC infrastructure for DC04 was utilised by a handful of users who managed to harness hundreds of years of computing power over a few months is testament to the success of its approach. The REST model aims for usability via simplicity, consistency, and comprehensibility of GRDL and the operations which are performed by comparators, matchers, and rankers. Extensibility (Key Goal 4) is foundational and in DIRAC is achieved through the modular service oriented architecture, while in the REST model is enabled through the generic nature of GRDL and the implicit extensibility of XML. Manageability (Key Goal 5), finally, has been achieved in DIRAC through the use of instant messaging interfaces to services and agents, and a small, simple, architecture. The Configuration Service also greatly facilitated service discovery, configuration, and management. This Key Goal cannot be evaluated for the REST model as no operational implementations exist, however the design of the model has emphasised simplicity and comprehensibility which are key to achieving ease of management.

9.3 Future Work

The objective of this work was to develop a RESTful grid model, starting with a generic model for representing resources via GRDL. There remain many areas where this can be more fully developed. The first step is the formalisation of a RESTful grid

process model. Early work has begun on this and needs further development. The objective is to enable GRDL to represent resource state throughout the resource's life cycle.

There have been many small points which have been touched on but not fully developed. Each of these has potential for further exploration in contributing to an effective RESTful grid model, or as optimisation points for implementations or useability. Within the context of the work which has been described here, two clear areas remain: simulation of the model in the context of a large dynamic computational grid, and performance evaluation of a full implementation. Simulation of large dynamic generic computational grids is a difficult task, as it is not well supported by any of the available simulation tools. SimGrid[113] is the best available tool, however it caters for simulation of a single fixed distributed algorithm with well defined workflow/task properties throughout the simulation. Dynamic traces of resource behaviour are possible, but creating such a model was beyond the scope of what was achievable in this work. It is only on the scale of a large, long running time, heterogenous dynamic computational grid that the key characteristics of the model's performance can be observed. Realisable simulations are sufficiently simplistic as to show the same results between the REST model and a traditional batch management/scheduling system. In order to simulate a large computational grid it is necessary to characterise the dynamic profiles of various aspects of a grid over a long period: storage, network, processing power, and workload generation. It is necessary to inject failures into all of those aspects, and represent the RESTful model within the simulation.

As has been discussed at length, the concepts presented in the REST model were motivated by the proto-REST implementation found in the DIRAC architecture, therefore there is some real-world validation for these concepts. The two implementations of the REST scheduling model which were prepared in conjunction with this work (one in Python and one in Haskell) were developed for exploring the properties of the model, rather than as part of a complete grid resource management system. As such, they were unsuitable for performance benchmarking. A full implementation would enable performance measures to be taken and provide a comparison against DIRAC and LCG task management. The final aspect which requires careful consideration is a security model. This ties in with both the GRDL model, and an overall model for a RESTful grid process.

In the context of particle physics computing a RESTful grid implementation would empower users to experiment with, extend, and improve the strategies for

grid resource management. A staged grid process finite state machine model would facilitate task management from creation, to scheduling, to staging, to execution, to completion, to archiving. It is even conceivable a GRDL-like model, coupled with a finite state machine, could allow checkpointing, recovery, and work flow management. It is still necessary to focus on the basic objective of large scale distribution, execution, and management of single grid tasks (*i.e.* embarrassingly parallel problems, or high volume decoupled tasks). To achieve this, greater degrees of resource logging, experimentation, and simulation are required, with particular attention given to translating the experience from preemptive operating systems to a grid domain. Added to that is the requirement for a comprehensible security infrastructure with a strong emphasis on groups (Virtual Organisations) and roles. The experience from DC04 suggests that a single identity approach such as basic X.509 certificates is insufficient – the reality is that users have many different identity tokens all of which need to be made accessible in a grid environment and which may form part of an operation within a grid task. The complexity of current X.509 systems, security policies, and role-based access control clearly indicate that a significant amount of work remains to simplify this to a level which is usable by the ordinary user.

The principles which underlie improving computational grid architectures must be independent of any particular technology or implementation approach, thinking in particular of Web Services. While Web Services provide a strong foundation for a service oriented grid architecture, their weaknesses in the grid domain quickly became clear. Again, this focuses attention on the plethora of Internet standards which manage to inter-operate or co-exist as part of a global computing infrastructure. Decoupling aspects of a grid architecture into simple, efficient, scalable, and reliable services and protocols has the benefit of following a path which the Internet has proven can lead to success. A REST approach for representing the entities within the grid opens the way for different user groups to operate on grid entities in their own way, and allows the best mechanisms and implementations to rise to the top "organically", rather than by asserting *a priori* a particular set of services, or worse an entire grid infrastructure.

The experience and examples found in the successful Internet standards should form the basis of future work in computational grids. While the model presented here breaks from tradition in the distributed computing sense, it very much builds on a long tradition of large scale computing as established through standards such as DNS, HTTP, HTML, and XML. It is hoped that this RESTful approach will provide a new perspective on scheduling strategies and large grid architectures which

will move grid computing closer to its desired goal of Internet-scale federated heterogenous dynamic distributed computing.

Appendix A

Task and Executor Description Languages

A.1 Globus Resource Specification Language

The following table is a summary of the properties described in the RSL Specification [99].

RSL	description
directory	specifies the path of the directory the jobmanager will use as the default directory for the requested job.
executable	The name of the executable file to run on the remote machine.
arguments	The command line arguments for the executable.
stdin	The name of the file to be used as standard input for the executable on the remote machine.
stdout	The name of the remote file to store the standard output from the job.
stderr	The name of the remote file to store the standard error from the job.
count	The number of executions of the executable.
environment	The environment variables that will be defined for the executable in addition to default set that is given to the job by the jobmanager.
maxTime	The maximum walltime or cputime for a single execution of the executable.
maxWallTime	Explicitly set the maximum walltime for a single execution of the executable.
maxCpuTime	Explicitly set the maximum cputime for a single execution of the executable.
jobType	This specifies how the jobmanager should start the job (single, multiple, mpi, condor).
gramMyJob	This specifies how the gram myjob interface will behave in the started processes.
queue	Target the job to a queue (class) name as defined by the scheduler at the defined (remote) resource.
project	Target the job to be allocated to a project account as defined by the scheduler at the defined (remote) resource.
hostCount	Defines the number of nodes ("pizza boxes") to distribute the "count" processes across.
dryRun	If dryrun = yes then the jobmanager will not submit the job for execution and will return success.
minMemory	Specify the minimum amount of memory required for this job.
maxMemory	Specify the maximum amount of memory required for this job.
save_state	Causes the jobmanager to save job state/information to a persistent file on disk.
two_phase	Implement a two-phase commit for job submission and completion.
restart	Start a new jobmanager but instead of submitting a new job, start watching over an existing job.
stdout_position	Specifies where in the file streaming should be restarted from for streamed output.
stderr_position	Specifies where in the file streaming should be restarted from for streamed error.
remote_io_url	Provides the base URL prefix for remote IO operations.

Table A.1: *Summary of RSL properties.*

A.2 Job Description Language

The following is a summary of the attributes provided by the latest published version of the EGEE Project Job Description Language[101]. These attributes are utilised in Condor ClassAd, and processed by the LCG Workload Management System (WMS). This attribute set originated with the EDG project. Its purpose is to describe in a standard way the pre-conditions and execution details for a task or task set to be executed on a computational grid. Table A.2 provides an overview of the key attributes in JDL. It is not definitive.

Table A.2: *Job Description Language attributes*

Item	Example	Description
Type	Job,DAG	Whether the JDL contains a single task or a DAG task graph
JobType	Normal, Interactive, MPICH, Checkpointable, Partitionable	What type of job is to be executed
Executable	/path/to/binary	Command to execute
Arguments	-s 1.2 input.dat	Arguments to pass to executable
StdInput	filename	File to pass as input
StdOutput	filename	Where to send output
StdError	filename	Where to send errors
InputSandbox	filenames	Files to be sent from local system to remote system at start of task
OutputSandbox	filenames	Files to be sent from remote system to local system at end of task
ExpiryTime	1112339655	Latest start time for task (abort scheduling attempts past this time)
Environment	JAVABIN=/usr/local/java	Declare remote environment settings
InputData	filenames	Files to be sent from grid storage to remote system at start of task
OutputData	filenames	Files to be sent from remote system to grid storage at end of task
DataAccessProtocol	file, gridftp	A list of protocols which the application can use to access data
StorageIndex	URL	Which file catalog the WMS should use to resolve the location of StorageIndex files listed in InputData
DataCatalog	URL	Which file catalog the WMS should use to resolve the location of non-StorageIndex files listed in InputData
OutputSE	URL	The storage element to be used for OutputData files
VirtualOrganisation	vo-name	Which VO the task should be run as
RetryCount	3	Number of re-tries in the event of grid middleware failure

MyProxyServer	URL	Address of a MyProxy server which holds the user's proxy certificate for retrieval by various authorised services
NodeNumber	10	Number of independent CPUs (nodes) required for execution of an MPICH job
JobSteps	100	Number of steps or checkpoints for completion of task
CurrentStep	5	Current step or checkpoint to utilise when starting the task
ListenerPort	44000	Port on UI (submit) node to which task sends interactive updates
Requirements		Specify conditions for WMS to utilise in matching task to executor
Rank		Specify ranking order for WMS candidate set
FuzzyRank	true,false	Enable some randmoness in rank selection process
UserTags		Nested classad containing name/value pairs which are logged to facilitate task queries
max_nodes_running	10	Maximum number of nodes in a DAG which can be executing at any given time
nodes		Set of ClassAds describing a DAG of tasks
dependencies		DAG relationship of a set of ClassAds
StepWeight		Hint regarding relative computational complexity of steps in a partitionable job
PreJob		Task ClassAd which is executed before any steps or task partitions are executed
PostJob		Task ClassAd which is executed after all steps or task partitions have been executed

A.3 GLUE Resource Description Schema

In order to make scheduling decisions in a computational grid it is necessary for computing resources to export information regarding their state. This is a combination of static and dynamic properties. While information and monitoring services may provide generic interfaces for updating or querying items, it is still necessary to define a common information model. The Grid Lab project developed the Grid Laboratory Uniform Environment information schema (GLUE Schema) to address the definition of a site, and the computing and storage resources available there. This is an evolving schema. Initially it contained exclusively "flat" name/value pairs, however it has recently incorporated structural relationships between through the use of identifiers.

Table A.3: *Categorised properties defined in the GLUE Schema.*

Site
Name
Description
UserSupportContact (list of email addresses)
SysAdminContact (list of email addresses)
SecurityContact (list of email addresses)
Location (address)
Latitude (real)
Longitude (real)
Web (URL to webpage for site)
Sponsor (VO representation at site)
OtherInfo
Associations: Services, Computing Elements, Storage Elements.

Service
Name
Type
Version
Endpoint (URI)
Status ([OK, Warning, Critical, Unknown, Other])
StatusInfo (string)
WSDL (URI to WSDL file)
Semantics (URL to detailed service description)
StartTime (last start time of service)
Owner
ServiceData (list of name/value pairs)
Associations: Site, other Services

Computing Resource
Name
InformationService (close or local to computing resource)
LRMSType
LRMSVersion
GRAMVersion
HostName
GatekeeperPort
JobManager
ContactString
TotalCPUs
ApplicationDir (path to install applications)
DataDir (path to shared application data)
DefaultSE (SE associated with this computing resource)

Job
LocalOwner
GlobalOwner
Status
SchedulerSpecific (catch all)

A.4 Outline of JSDL

The following provides an outline of the XML structure provided by JSDL, as specified in [102]. Cardinality is indicated by '?' for zero or one occurrence, and '*' for zero or more occurrences.

```
JobDefinition

  JobDescription ?
    JobIdentification ?
      JobName ?
      Description ?
      JobAnnotation *
      JobProject *

  Application name version ?
    POSIXApplication name
      Executable ?
      Argument *
```

```
      Input ?
      Output ?
      Error ?
      WorkingDirectory ?
      Environment *
      WallTimeLimit ?
      FileSizeLimit ?
      CoreDumpLimit ?
      DataSegmentLimit ?
      LockedMemoryLimit ?
      MemoryLimit ?
      OpenDescriptorsLimit ?
      PipeSizeLimit ?
      StackSizeLimit ?
      CPUTimeLimit ?
      ProcessCountLimit ?
      VirtualMemoryLimit ?
      ThreadCountLimit ?
      UserName ?
      GroupName ?

  Resources ?
    CandidateHosts ?
    CPUArchitecture ?
    ExlusiveExecution ?
    FileSystem *
    IndividualCPUCount ?
    TotalCPUCount ?
    IndividualCPUSpeed ?
    IndividualCPUTime ?
    TotalCPUTime ?
    IndividualDiskSpace ?
    TotalDiskSpace ?
    IndividualNetworkBandwidth ?
    IndividualPhysicalMemory ?
    TotalPhysicalMemory ?
    IndividualVirtualMemory ?
    TotalVirtualMemory ?
    TotalResourceCount ?
    OperatingSystem name version ?

  DataStaging *
    FileName
    FileSystemName ?
```

```
CreationFlag
DeleteOnTermination ?
Source ?
Target ?
```

Listing A.1: *Outline of JSDL with POSIX extensions*

A.5 Comparison of JSDL, JDL, GLUE and RSL

Table A.4: *Comparison of JSDL, JDL, GLUE and (x)RSL*

Description	JSDL	JDL	GLUE	(x)RSL
Job Description				
Resource type (e.g. Job, DAG, Reservation, Co-allocation)		Type		
Job type (e.g. Normal, Interactive, MPICH, Checkpointable, Partionable)		JobType		jobType
	JobName		LocalID, GlobalID	jobName, jobid
	Description			label
	JobAnnotation			
	JobProject			project
		MyProxyServer		
Executable Description				
	ApplicationName			
	ApplicationVersion			
	POSIXApplication name			
	Executable	Executable		executable
	Argument	Arguments		arguments
Environment Settings				
	Input	StdInput		stdin
	Output	StdOutput		stdout
	Error	StdError		stderr
	WorkingDirectory		DataDir	directory
Set environment for task execution	Environment	Environment		environment
Assert *a priori* environment settings			ApplicationSoftware:: RuntimeEnvironment	runTime Environment, middleware
	OperatingSystem@name		OperatingSystemName	

Continued on Next Page. . .

Table A.4 – Continued

Description	JSDL	JDL	GLUE	(x)RSL
	OperatingSystem@version		OperatingSystemRelease, OperatingSystemVersion	
Job Requirements				
		Requirements		
		Rank		
		FuzzyRank		
	CandidateHosts		HostName	cluster, queue
	ExlusiveExecution			
Earliest start time of task				startTime
Duration to save local data and task details after task completes				lifeTime
		JobSteps		count
		CurrentStep		
		RetryCount		rerun
				dryRun
				gramMyJob
Memory and File Limits				
	FileSizeLimit		MaxFileSize	
	CoreDumpLimit			
	DataSegmentLimit			
	LockedMemoryLimit			
	MemoryLimit			memory
	OpenDescriptorsLimit			
	PipeSizeLimit			
	StackSizeLimit			
	VirtualMemoryLimit			
	ThreadCountLimit			
	ProcessCountLimit			

Continued on Next Page. . .

Table A.4 – Continued

Description	JSDL	JDL	GLUE	(x)RSL
	IndividualPhysicalMemory		MainMemory:: RAMSize, RAMAvailable	maxMemory
				minMemory
				cache
	IndividualVirtualMemory		MainMemory:: VirtualSize, VirtualAvailable	
	TotalPhysicalMemory			
	TotalVirtualMemory			
Contact and Notification details				
Task owner contact details			LocalOwner, GlobalOwner	notify
Administrator contact details			SysAdminContact	resource ManagerContact
User Support contact details			UserSupportContact	
Security contact details			SecurityContact	
Home Location Register URL for accounting		HLRLocation		jobreport
Access Control				
Specify user access control on task	UserName		AccessControlBaseRule	acl
Specify group access control on task	GroupName	VirtualOrganisation	AccessControlBaseRule	acl
CPU Description and Limits				
	CPUTimeLimit		maxTime	maxCpuTime, gridTime
	WallTimeLimit			maxWallTime
CPU Architecture	CPUArchitecture		Architecture:: PlatformType	architecture
Processor details			Processor::Vendor	

Continued on Next Page. . .

Table A.4 – Continued

Description	JSDL	JDL	GLUE	(x)RSL
Processor details			Processor::Model	
Processor details			Processor::Version	
Processor details			Processor:: InstructionSet	
Processor details			Processor:: OtherProcessorDescription	
Processor details			Processor::CacheL1	
Processor details			Processor::CacheL1I	
Processor details			Processor::CacheL1D	
Processor details			Processor::CacheL2	
Actual CPUs per node	IndividualCPUCount		Architecture:: SMPSize	
Virtual CPUs per node (hyperthreading)			Architecture:: SMTSize	
	TotalCPUCount	NodeNumber		hostCount, count
	IndividualCPUSpeed		ProcessorClockSpeed	
	IndividualCPUTime			
	TotalCPUTime			cpuTime, gridTime
	TotalResourceCount			hostCount
Local Resource Management System	LRMSType			lrmstype
			Benchmark::SI00	
			Benchmark::SF00	
			Load::Last1Min	
			Load::Last5Min	
			Load::Last15Min	
Data Staging and File Requirements				
	FileSystem		Name	
	MountPoint		Root	directory
	DiskSpace		System::Size	

Continued on Next Page. . .

Table A.4 – Continued

Description	JSDL	JDL	GLUE	(x)RSL
	FileSystemType		Type	
	IndividualDiskSpace			disk
	TotalDiskSpace			
	FileName			
	FileSystemName			
	CreationFlag			
	DeleteOnTermination			
	Source	InputSandbox, InputData		inputFiles
	Target	OutputSandbox, OutputData		outputFiles
		DataAccessProtocol		
		OutputSE		
Networking Requirements				
	IndividualNetworkBandwidth			
			Name	
			MTU	
			InboundIP	nodeAccess
			OutboundIP	nodeAccess
			IPAddress	

A.6 GLUE SE properties

Table A.5: *GLUE Storage Element properties*

Storage Area
Path
Type
State::UsedSpace
State::AvailableSpace
Policy::Quota
Policy::MinFileSize
Policy::MaxFileSize
Policy::MaxData
Policy::MaxNumFiles
Policy::MaxPinDuration
AccessControlBase::Rule
Storage Element
SizeTotal
SizeFree
Architecture
InformationServiceURL
ControlProtocol::Endpoint
ControlProtocol::Type
ControlProtocol::Version
ControlProtocol::Capability
AccessProtocol::Endpoint
AccessProtocol::Type
AccessProtocol::Version
AccessProtocol::Capability
Storage Device
Name
Type
Size
TransferRate
Storage Partition
Name
Size
ReadRate
WriteRate
File System
AvailableSpace
ReadOnly

A.7 Summary of CDDLM State Machine and API

```
create()        -> Instantiated
initialize()    -> Initialized
run()           -> Running
[fault]         -> Failed
terminate()     -> Terminated
destroy()       -> Destroyed

AddFile()
Initialize()
Resolve()
Ping()
Run()
Terminate()
Create()
Resolve()
LookupSystem()
```

Appendix B

Grid Resource Description Language

The examples of GRDL presented in this work have been done in XML. Nothing in the GRDL model pre-supposes the use of XML as a representational language, however it does provide the benefit of a widely supported and extensible meta-syntax with which to construct GRDL. This appendix defines the XML Schema for GRDL which gives both a description of GRDL and a mechanism by which GRDL may be validated. For simplicity, the base GRDL language does not make use of XML Namespaces, however these are easily applied by including the *null* namespaced schema in a wrapping schema with a namespace. The namespace chosen for this version of GRDL is `urn:cern-ch:grid:GRDL:2006:1.0`. A DTD definition is also provided, however this defines a more limited subset of GRDL and provides a lower level of syntactic validation.

B.1 GRDL Property and Type List Schema

Listing B.1: *The no-namespaced GRDL schema GRDL-noNS-PropertyTypes-v1.0.xsd. This provides the base types for the dimension definitions.*

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:complexType name="CharacteristicType">
        <xs:simpleContent>
            <xs:extension base="xs:string">
                <xs:attribute name="type" type="TypeList"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
```

```
<xs:complexType name="RequirementType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicType">
            <xs:attribute name="match" type="MatchList" default="=="/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="PreferenceType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicType">
            <xs:attribute name="match" type="RankList" default="=="/>
            <xs:attribute name="rank" type="xs:float" default="1"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="CharacteristicNoTypeType">
    <xs:simpleContent>
        <xs:extension base="xs:string"/>

    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="RequirementNoTypeType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicNoTypeType">
            <xs:attribute name="match" type="MatchList" default="=="/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="PreferenceNoTypeType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicNoTypeType">
            <xs:attribute name="match" type="RankList" default="=="/>
            <xs:attribute name="rank" type="xs:float" default="1"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="CharacteristicIntType">
    <xs:simpleContent>
        <xs:extension base="xs:integer">
            <xs:attribute name="type" type="TypeList"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="RequirementIntType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicIntType">
            <xs:attribute name="match" type="MatchList" default="=="/>
        </xs:extension>
```

```
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="PreferenceIntType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicIntType">
            <xs:attribute name="match" type="RankList" default="=="/>
            <xs:attribute name="rank" type="xs:float" default="1"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="CharacteristicNonNegIntType">
    <xs:simpleContent>
        <xs:extension base="xs:nonNegativeInteger">
            <xs:attribute name="type" type="TypeList"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="RequirementNonNegIntType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicNonNegIntType">
            <xs:attribute name="match" type="MatchList" default="=="/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="PreferenceNonNegIntType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicNonNegIntType">
            <xs:attribute name="match" type="RankList" default="=="/>
            <xs:attribute name="rank" type="xs:float" default="1"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="CharacteristicFloatType">
    <xs:simpleContent>
        <xs:extension base="xs:float">
            <xs:attribute name="type" type="TypeList"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="RequirementFloatType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicFloatType">
            <xs:attribute name="match" type="MatchList" default="=="/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="PreferenceFloatType">
    <xs:simpleContent>
```

```
        <xs:extension base="CharacteristicFloatType">
            <xs:attribute name="match" type="RankList" default="=="/>
            <xs:attribute name="rank" type="xs:float" default="1"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="CharacteristicTimestampType">
    <xs:simpleContent>
        <xs:extension base="xs:dateTime">
            <xs:attribute name="type" type="TypeList"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="RequirementTimestampType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicTimestampType">
            <xs:attribute name="match" type="MatchList" default="=="/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="PreferenceTimestampType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicTimestampType">
            <xs:attribute name="match" type="RankList" default="=="/>
            <xs:attribute name="rank" type="xs:float" default="1"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="CharacteristicTimeType">
    <xs:simpleContent>
        <xs:extension base="xs:nonNegativeInteger">
            <xs:attribute name="type" type="TimeTypeList"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="RequirementTimeType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicTimeType">
            <xs:attribute name="match" type="MatchList" default="=="/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="PreferenceTimeType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicTimeType">
            <xs:attribute name="match" type="RankList" default="=="/>
            <xs:attribute name="rank" type="xs:float" default="1"/>
        </xs:extension>
    </xs:simpleContent>
```

```
</xs:complexType>

<xs:complexType name="CharacteristicMetricType">
    <xs:simpleContent>
        <xs:extension base="xs:nonNegativeInteger">
            <xs:attribute name="type" type="MetricTypeList"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="RequirementMetricType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicMetricType">
            <xs:attribute name="match" type="MatchList" default="=="/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="PreferenceMetricType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicMetricType">
            <xs:attribute name="match" type="RankList" default="=="/>
            <xs:attribute name="rank" type="xs:float" default="1"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="CharacteristicFrequencyType">
    <xs:simpleContent>
        <xs:extension base="xs:nonNegativeInteger">
            <xs:attribute name="type" type="FrequencyTypeList"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="RequirementFrequencyType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicFrequencyType">
            <xs:attribute name="match" type="MatchList" default="=="/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="PreferenceFrequencyType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicFrequencyType">
            <xs:attribute name="match" type="RankList" default="=="/>
            <xs:attribute name="rank" type="xs:float" default="1"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="CharacteristicBinaryType">
    <xs:simpleContent>
        <xs:extension base="xs:nonNegativeInteger">
```

```
            <xs:attribute name="type" type="BinaryTypeList"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="RequirementBinaryType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicBinaryType">
            <xs:attribute name="match" type="MatchList" default="=="/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="PreferenceBinaryType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicBinaryType">
            <xs:attribute name="match" type="RankList" default="=="/>
            <xs:attribute name="rank" type="xs:float" default="1"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="CharacteristicTransferType">
    <xs:simpleContent>
        <xs:extension base="xs:nonNegativeInteger">
            <xs:attribute name="type" type="TransferTypeList"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="RequirementTransferType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicTransferType">
            <xs:attribute name="match" type="MatchList" default="=="/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="PreferenceTransferType">
    <xs:simpleContent>
        <xs:extension base="CharacteristicTransferType">
            <xs:attribute name="match" type="RankList" default="=="/>
            <xs:attribute name="rank" type="xs:float" default="1"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>


<xs:simpleType name="TypeList">
    <xs:union memberTypes="TimeTypeList MetricTypeList
                           BinaryTypeList TransferTypeList
                           FrequencyTypeList TimestampTypeList"/>
</xs:simpleType>

<xs:simpleType name="TimestampTypeList">
```

```
    <xs:restriction base="xs:dateTime"/>
</xs:simpleType>

<xs:simpleType name="TimeTypeList">
    <xs:restriction base="xs:string">
        <xs:enumeration value="fs"/>
        <xs:enumeration value="ps"/>
        <xs:enumeration value="ns"/>
        <xs:enumeration value="us"/>
        <xs:enumeration value="ms"/>
        <xs:enumeration value="s"/>
        <xs:enumeration value="sec"/>
        <xs:enumeration value="second"/>
        <xs:enumeration value="min"/>
        <xs:enumeration value="minute"/>
        <xs:enumeration value="h"/>
        <xs:enumeration value="hr"/>
        <xs:enumeration value="hour"/>
        <xs:enumeration value="d"/>
        <xs:enumeration value="day"/>
        <xs:enumeration value="wk"/>
        <xs:enumeration value="week"/>
        <xs:enumeration value="mon"/>
        <xs:enumeration value="month"/>
        <xs:enumeration value="yr"/>
        <xs:enumeration value="year"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="MetricTypeList">
    <xs:restriction base="xs:string">
        <xs:enumeration value="f"/>
        <xs:enumeration value="p"/>
        <xs:enumeration value="n"/>
        <xs:enumeration value="u"/>
        <xs:enumeration value="m"/>
        <xs:enumeration value="K"/>
        <xs:enumeration value="M"/>
        <xs:enumeration value="G"/>
        <xs:enumeration value="T"/>
        <xs:enumeration value="P"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="FrequencyTypeList">
    <xs:restriction base="xs:string">
        <xs:enumeration value="Hz"/>
        <xs:enumeration value="KHz"/>
        <xs:enumeration value="MHz"/>
        <xs:enumeration value="GHz"/>
        <xs:enumeration value="THz"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="BinaryTypeList">
```

```
        <xs:restriction base="xs:string">
            <xs:enumeration value="B" />
            <xs:enumeration value="KB" />
            <xs:enumeration value="MB" />
            <xs:enumeration value="GB" />
            <xs:enumeration value="TB" />
            <xs:enumeration value="PB" />
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="TransferTypeList">
        <xs:restriction base="xs:string">
            <xs:enumeration value="Kb/s" />
            <xs:enumeration value="Mb/s" />
            <xs:enumeration value="Gb/s" />
            <xs:enumeration value="Tb/s" />
            <xs:enumeration value="Pb/s" />
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="MatchList">
        <xs:restriction base="xs:string">
            <xs:enumeration value="==" />
            <xs:enumeration value="!=" />
            <xs:enumeration value=">" />
            <xs:enumeration value="&lt;" />
            <xs:enumeration value=">=" />
            <xs:enumeration value="&lt;=" />
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="RankList">
        <xs:union memberTypes="MatchList RankDistanceList" />
    </xs:simpleType>
    <xs:simpleType name="RankDistanceList">
        <xs:restriction base="xs:string">
            <xs:enumeration value="&gt;&lt;" />
            <xs:enumeration value="&lt;&gt;" />
        </xs:restriction>
    </xs:simpleType>

</xs:schema>
```

B.2 GRDL Dimensions Schema

Listing B.2: *The no-namespaced GRDL schema GRDL-noNS-Dimensions-v1.0.xsd. This defines the dimensions which are understood. Every dimension has three definitions one for each of Characteristics Requirements and Preferences.*

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

```
<xs:include schemaLocation="GRDL-noNS-PropertyTypes-v1.0.xsd" />

<xs:complexType name="AllCharacteristics">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
        <!-- Selected GLUE Schema Properties -->
        <xs:element name="Latitude" type="CharacteristicFloatType" />
        <xs:element name="Longitude" type="CharacteristicFloatType" />
        <xs:element name="TotalCPUs" type="CharacteristicIntType" />
        <xs:element name="StartTime" type="CharacteristicTimeType" />
        <xs:element name="Status" type="CharacteristicType" />
        <xs:element name="Description" type="CharacteristicType" />
        <xs:element name="Contact" type="CharacteristicType" />
        <xs:element name="Load" type="CharacteristicFloatType" />
        <xs:element name="Benchmark" type="CharacteristicFloatType" />
        <xs:element name="AccessProtocol" type="CharacteristicType" />
        <!-- Selected JSDL Properties -->
        <xs:element name="Name" type="CharacteristicType" />
        <xs:element name="Project" type="CharacteristicType" />
        <xs:element name="Executable" type="CharacteristicType" />
        <xs:element name="Argument" type="CharacteristicType" />
        <xs:element name="WallTimeLimit" type="CharacteristicTimeType" />
        <xs:element name="CPUTimeLimit" type="CharacteristicTimeType" />
        <xs:element name="FileSizeLimit" type="CharacteristicBinaryType" />
        <xs:element name="VirtualMemoryLimit" type="CharacteristicBinaryType" />
        <xs:element name="CPUArchitecture" type="CharacteristicType" />
        <xs:element name="ExclusiveExecution" type="CharacteristicType" />
        <xs:element name="IndividualCPUCount" type="CharacteristicIntType" />
        <xs:element name="TotalCPUCount" type="CharacteristicIntType" />
        <xs:element name="IndividualCPUSpeed" type="CharacteristicFrequencyType" />
        <xs:element name="IndividualCPUTime" type="CharacteristicTimeType" />
        <xs:element name="TotalCPUTime" type="CharacteristicTimeType" />
        <xs:element name="IndividualDiskSpace" type="CharacteristicBinaryType" />
        <xs:element name="TotalDiskSpace" type="CharacteristicBinaryType" />
        <!-- Selected xRSL Properties -->
        <xs:element name="disk" type="CharacteristicBinaryType" />
        <xs:element name="cpuTime" type="CharacteristicTimeType" />
        <xs:element name="gridTime" type="CharacteristicTimeType" />
        <xs:element name="hostCount" type="CharacteristicIntType" />
        <xs:element name="count" type="CharacteristicIntType" />
        <xs:element name="maxWallTime" type="CharacteristicTimeType" />
        <xs:element name="maxCpuTime" type="CharacteristicTimeType" />
        <xs:element name="architecture" type="CharacteristicType" />
        <xs:element name="lifetime" type="CharacteristicTimeType" />
        <xs:element name="memory" type="CharacteristicBinaryType" />
        <xs:element name="maxMemory" type="CharacteristicBinaryType" />
        <xs:element name="minMemory" type="CharacteristicBinaryType" />
                    <!-- Selected JDL Properties -->
        <xs:element name="StdInput" type="CharacteristicType" />
        <xs:element name="StdOutput" type="CharacteristicType" />
        <xs:element name="JobType" type="CharacteristicType" />
        <xs:element name="InputSandbox" type="CharacteristicType" />
        <xs:element name="OutputSandbox" type="CharacteristicType" />
        <xs:element name="ExpiryTime" type="CharacteristicTimeType" />
        <xs:element name="Environment" type="CharacteristicType" />
        <xs:element name="DataAccessProtocol" type="CharacteristicType" />
```

```
        <xs:element name="OutputSE" type="CharacteristicType"/>
        <xs:element name="VirtualOrganisation" type="CharacteristicType"/>
    </xs:choice>
</xs:complexType>

<xs:complexType name="AllRequirements">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
        <!-- Selected GLUE Schema Properties -->
        <xs:element name="Latitude" type="RequirementFloatType"/>
        <xs:element name="Longitude" type="RequirementFloatType"/>
        <xs:element name="TotalCPUs" type="RequirementIntType"/>
        <xs:element name="StartTime" type="RequirementTimeType"/>
        <xs:element name="Status" type="RequirementType"/>
        <xs:element name="Description" type="RequirementType"/>
        <xs:element name="Contact" type="RequirementType"/>
        <xs:element name="Load" type="RequirementFloatType"/>
        <xs:element name="Benchmark" type="RequirementFloatType"/>
        <xs:element name="AccessProtocol" type="RequirementType"/>
        <!-- Selected JSDL Properties -->
        <xs:element name="Name" type="RequirementType"/>
        <xs:element name="Project" type="RequirementType"/>
        <xs:element name="Executable" type="RequirementType"/>
        <xs:element name="Argument" type="RequirementType"/>
        <xs:element name="WallTimeLimit" type="RequirementTimeType"/>
        <xs:element name="CPUTimeLimit" type="RequirementTimeType"/>
        <xs:element name="FileSizeLimit" type="RequirementBinaryType"/>
        <xs:element name="VirtualMemoryLimit" type="RequirementBinaryType"/>
        <xs:element name="CPUArchitecture" type="RequirementType"/>
        <xs:element name="ExclusiveExecution" type="RequirementType"/>
        <xs:element name="IndividualCPUCount" type="RequirementIntType"/>
        <xs:element name="TotalCPUCount" type="RequirementIntType"/>
        <xs:element name="IndividualCPUSpeed" type="RequirementFrequencyType"/>
        <xs:element name="IndividualCPUTime" type="RequirementTimeType"/>
        <xs:element name="TotalCPUTime" type="RequirementTimeType"/>
        <xs:element name="IndividualDiskSpace" type="RequirementBinaryType"/>
        <xs:element name="TotalDiskSpace" type="RequirementBinaryType"/>
        <!-- Selected xRSL Properties -->
        <xs:element name="disk" type="RequirementBinaryType"/>
        <xs:element name="cpuTime" type="RequirementTimeType"/>
        <xs:element name="gridTime" type="RequirementTimeType"/>
        <xs:element name="hostCount" type="RequirementIntType"/>
        <xs:element name="count" type="RequirementIntType"/>
        <xs:element name="maxWallTime" type="RequirementTimeType"/>
        <xs:element name="maxCpuTime" type="RequirementTimeType"/>
        <xs:element name="architecture" type="RequirementType"/>
        <xs:element name="lifetime" type="RequirementTimeType"/>
        <xs:element name="memory" type="RequirementBinaryType"/>
        <xs:element name="maxMemory" type="RequirementBinaryType"/>
        <xs:element name="minMemory" type="RequirementBinaryType"/>
                    <!-- Selected JDL Properties -->
        <xs:element name="StdInput" type="RequirementType"/>
        <xs:element name="StdOutput" type="RequirementType"/>
        <xs:element name="JobType" type="RequirementType"/>
        <xs:element name="InputSandbox" type="RequirementType"/>
        <xs:element name="OutputSandbox" type="RequirementType"/>
```

```
        <xs:element name="ExpiryTime" type="RequirementTimeType"/>
        <xs:element name="Environment" type="RequirementType"/>
        <xs:element name="DataAccessProtocol" type="RequirementType"/>
        <xs:element name="OutputSE" type="RequirementType"/>
        <xs:element name="VirtualOrganisation" type="RequirementType"/>
    </xs:choice>
</xs:complexType>

<xs:complexType name="AllPreferences">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
        <!-- Selected GLUE Schema Properties -->
        <xs:element name="Latitude" type="PreferenceFloatType"/>
        <xs:element name="Longitude" type="PreferenceFloatType"/>
        <xs:element name="TotalCPUs" type="PreferenceIntType"/>
        <xs:element name="StartTime" type="PreferenceTimeType"/>
        <xs:element name="Status" type="PreferenceType"/>
        <xs:element name="Description" type="PreferenceType"/>
        <xs:element name="Contact" type="PreferenceType"/>
        <xs:element name="Load" type="PreferenceFloatType"/>
        <xs:element name="Benchmark" type="PreferenceFloatType"/>
        <xs:element name="AccessProtocol" type="PreferenceType"/>
        <!-- Selected JSDL Properties -->
        <xs:element name="Name" type="PreferenceType"/>
        <xs:element name="Project" type="PreferenceType"/>
        <xs:element name="Executable" type="PreferenceType"/>
        <xs:element name="Argument" type="PreferenceType"/>
        <xs:element name="WallTimeLimit" type="PreferenceTimeType"/>
        <xs:element name="CPUTimeLimit" type="PreferenceTimeType"/>
        <xs:element name="FileSizeLimit" type="PreferenceBinaryType"/>
        <xs:element name="VirtualMemoryLimit" type="PreferenceBinaryType"/>
        <xs:element name="CPUArchitecture" type="PreferenceType"/>
        <xs:element name="ExclusiveExecution" type="PreferenceType"/>
        <xs:element name="IndividualCPUCount" type="PreferenceIntType"/>
        <xs:element name="TotalCPUCount" type="PreferenceIntType"/>
        <xs:element name="IndividualCPUSpeed" type="PreferenceFrequencyType"/>
        <xs:element name="IndividualCPUTime" type="PreferenceTimeType"/>
        <xs:element name="TotalCPUTime" type="PreferenceTimeType"/>
        <xs:element name="IndividualDiskSpace" type="PreferenceBinaryType"/>
        <xs:element name="TotalDiskSpace" type="PreferenceBinaryType"/>
        <!-- Selected xRSL Properties -->
        <xs:element name="disk" type="PreferenceBinaryType"/>
        <xs:element name="cpuTime" type="PreferenceTimeType"/>
        <xs:element name="gridTime" type="PreferenceTimeType"/>
        <xs:element name="hostCount" type="PreferenceIntType"/>
        <xs:element name="count" type="PreferenceIntType"/>
        <xs:element name="maxWallTime" type="PreferenceTimeType"/>
        <xs:element name="maxCpuTime" type="PreferenceTimeType"/>
        <xs:element name="architecture" type="PreferenceType"/>
        <xs:element name="lifetime" type="PreferenceTimeType"/>
        <xs:element name="memory" type="PreferenceBinaryType"/>
        <xs:element name="maxMemory" type="PreferenceBinaryType"/>
        <xs:element name="minMemory" type="PreferenceBinaryType"/>
                    <!-- Selected JDL Properties -->
        <xs:element name="StdInput" type="PreferenceType"/>
        <xs:element name="StdOutput" type="PreferenceType"/>
```

```
            <xs:element name="JobType" type="PreferenceType"/>
            <xs:element name="InputSandbox" type="PreferenceType"/>
            <xs:element name="OutputSandbox" type="PreferenceType"/>
            <xs:element name="ExpiryTime" type="PreferenceTimeType"/>
            <xs:element name="Environment" type="PreferenceType"/>
            <xs:element name="DataAccessProtocol" type="PreferenceType"/>
            <xs:element name="OutputSE" type="PreferenceType"/>
            <xs:element name="VirtualOrganisation" type="PreferenceType"/>
        </xs:choice>
    </xs:complexType>

</xs:schema>
```

B.3 GRDL Base Schema

Listing B.3: *The no-namespaced GRDL schema GRDL-noNS-Base-v1.0.xsd. This provides high level descriptions of resources.*

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:include schemaLocation="GRDL-noNS-Dimensions-v1.0.xsd"/>

    <xs:element name="resource" type="ResourceType"/>
    <xs:element name="task"     type="ResourceType"/>
    <xs:element name="executor" type="ResourceType"/>
    <xs:element name="storage"  type="ResourceType"/>

    <xs:complexType name="ResourceType">
        <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element name="chars" type="AllCharacteristics"/>
            <xs:element name="reqs"  type="AllRequirements"/>
            <xs:element name="prefs" type="AllPreferences"/>
        </xs:choice>
    </xs:complexType>

</xs:schema>
```

B.4 GRDL Namespace Wrapping Schema

Listing B.4: *The wrapping schema which adds a namespace to GRDL GRDL-Base-v1.0.xsd.*

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified"
    targetNamespace="urn:cern-ch:grid:GRDL:2006:1.0"
    xmlns="urn:cern-ch:grid:GRDL:2006:1.0"
    xmlns:grdl="urn:cern-ch:grid:GRDL:2006:1.0"
    version="2006:1.0" id="GRDL">

    <xs:include schemaLocation="GRDL-noNS-Base-v1.0.xsd"/>
```

```
</xs:schema>
```

B.5 GRDL DTD Schema

Listing B.5: *The GRDL-v1.0.dtd DTD for some of the JSDL dimensions expressed as GRDL XML.*

```
<!ELEMENT resource (chars*,reqs*,prefs*) >
<!ELEMENT task     (chars*,reqs*,prefs*) >
<!ELEMENT executor (chars*,reqs*,prefs*) >

<!ELEMENT chars (Executable*, Argument*, WallTimeLimit*, CPUTimeLimit*,
                              FileSizeLimit*, VirtualMemoryLimit*, CPUArchitecture*) >
<!ELEMENT prefs (Executable*, Argument*, WallTimeLimit*, CPUTimeLimit*,
                              FileSizeLimit*, VirtualMemoryLimit*, CPUArchitecture*) >
<!ELEMENT  reqs (Executable*, Argument*, WallTimeLimit*, CPUTimeLimit*,
                              FileSizeLimit*, VirtualMemoryLimit*, CPUArchitecture*) >

<!ELEMENT Executable (#PCDATA) >
<!ELEMENT Argument (#PCDATA) >
<!ELEMENT WallTimeLimit (#PCDATA) >
<!ELEMENT CPUTimeLimit (#PCDATA) >
<!ELEMENT FileSizeLimit (#PCDATA) >
<!ELEMENT VirtualMemoryLimit (#PCDATA) >
<!ELEMENT CPUArchitecture (#PCDATA) >

<!ATTLIST Executable
 type  CDATA #IMPLIED
 match CDATA #IMPLIED
 rank  CDATA #IMPLIED
  >

<!ATTLIST Argument
 type  CDATA #IMPLIED
 match CDATA #IMPLIED
 rank  CDATA #IMPLIED
  >

<!ATTLIST WallTimeLimit
 type  CDATA #IMPLIED
 match CDATA #IMPLIED
 rank  CDATA #IMPLIED
  >

<!ATTLIST CPUTimeLimit
 type  CDATA #IMPLIED
 match CDATA #IMPLIED
 rank  CDATA #IMPLIED
  >

<!ATTLIST FileSizeLimit
 type  CDATA #IMPLIED
```

```
  match CDATA #IMPLIED
  rank  CDATA #IMPLIED
   >

<!ATTLIST VirtualMemoryLimit
  type  CDATA #IMPLIED
  match CDATA #IMPLIED
  rank  CDATA #IMPLIED
   >

<!ATTLIST CPUArchitecture
  type  CDATA #IMPLIED
  match CDATA #IMPLIED
  rank  CDATA #IMPLIED
   >
```

Appendix C

Haskell Descriptions of the GRDL Model and Operations

C.1 Core GRDL Components

Listing C.1: *Characteristic Requirement Preference and Resource descriptions and their accessor functions.*

```
-----------------------------------------------------------------
-- Base Property Types
-----------------------------------------------------------------
type Dim        =   String
type Type       =   String  -- a separate mapping function maps
                            -- Type names to transform functions
type Value      =   Double  -- for illustrative purposes,
                            -- only "Doubles" allowed as Values
type Match      =   String  -- a separate mapping function maps
                            -- Match names to match functions
type Rank       =   Double
-----------------------------------------------------------------
-- Characteristic Type and Operations
-----------------------------------------------------------------
type Ch     = (Dim, Type, Value)

getChDim    :: Ch -> Dim
getChType   :: Ch -> Type
getChValue  :: Ch -> Value

getChDim    (d,t,v) = d
getChType   (d,t,v) = t
getChValue  (d,t,v) = v
-----------------------------------------------------------------
-- Requirement Type and Operations
-----------------------------------------------------------------
type Req    = (Dim, Type, Match, Value)
```

```
getReqDim   :: Req -> Dim
getReqType  :: Req -> Type
getReqMatch :: Req -> Match
getReqValue :: Req -> Value

getReqDim   (d,t,m,v) = d
getReqType  (d,t,m,v) = t
getReqMatch (d,t,m,v) = m
getReqValue (d,t,m,v) = v
```

```
-- Preference Type and Operations
```

```
type Pref  = (Dim, Type, Match, Value, Rank)

getPrefDim   :: Pref -> Dim
getPrefType  :: Pref -> Type
getPrefMatch :: Pref -> Match
getPrefValue :: Pref -> Value
getPrefRank  :: Pref -> Rank

getPrefDim   (d,t,m,v,r) = d
getPrefType  (d,t,m,v,r) = t
getPrefMatch (d,t,m,v,r) = m
getPrefValue (d,t,m,v,r) = v
getPrefRank  (d,t,m,v,r) = r
```

```
-- Resource Type and Operations
```

```
type Resource = ([Ch], [Req], [Pref])

getResChSet    :: Resource -> [Ch]
getResReqSet   :: Resource -> [Req]
getResPrefSet  :: Resource -> [Pref]

getResChSet    (cs,rs,ps) = cs
getResReqSet   (cs,rs,ps) = rs
getResPrefSet  (cs,rs,ps) = ps
```

C.2 Dimension Functions

Listing C.2: *Dimension operations.*

```
-- Dimension Functions
-----------------------------------------------------------------

getResDims      :: Resource -> Set Dim

getResDimsSet []          = empty
getResDimsSet (res:ress)  = unions [
                                     getResDims res,
                                     getResDimsSet ress
                                   ]

getResDims (chs, reqs, prefs) =    unions [
                                    getChSetDims   chs,
                                    getReqSetDims    reqs,
                                    getPrefSetDims   prefs
                                  ]

getChSetDims        :: [Ch]   -> Set Dim
getReqSetDims       :: [Req]  -> Set Dim
getPrefSetDims      :: [Pref] -> Set Dim

filterChByDimSet    :: [Ch]   -> Set Dim -> [Ch]
filterReqByDimSet   :: [Req]  -> Set Dim -> [Req]
filterPrefByDimSet  :: [Pref] -> Set Dim -> [Pref]

filterChResSubset       :: [Ch] -> Resource -> [Ch]

getChSetDims []         = empty
getChSetDims (c:cs)     = insert (getChDim c) (getChSetDims cs)

getReqSetDims  []       = empty
getReqSetDims  (r:rs)   = insert (getReqDim r) (getReqSetDims rs)

getPrefSetDims  []      = empty
getPrefSetDims  (p:ps)  = insert (getPrefDim p) (getPrefSetDims ps)

filterChByDimSet   cs dimSet = [c | c <- cs, member (getChDim   c) dimSet]
filterReqByDimSet  rs dimSet = [r | r <- rs, member (getReqDim  r) dimSet]
filterPrefByDimSet ps dimSet = [p | p <- ps, member (getPrefDim p) dimSet]

filterChResSubset  cs res    = filterChByDimSet cs
                               (getChSetDims (getResChSet res))
```

C.3 Type Definitions and Mapping Functions

Listing C.3: *Type definitions for GRDL Properties and mapping functions between type names and type functions.*

```
-----------------------------------------------------------------
-- Property Types and Comparability
-----------------------------------------------------------------
type TypeConv   =   Value -> Value
type TypeMap    =   [(Type,TypeConv)]


unit            ::  TypeConv
unit x          =   x

m_f             ::  TypeConv
m_p             ::  TypeConv
m_n             ::  TypeConv
m_u             ::  TypeConv
m_m             ::  TypeConv
m_K             ::  TypeConv
m_base          ::  TypeConv
m_M             ::  TypeConv
m_G             ::  TypeConv
m_T             ::  TypeConv
m_P             ::  TypeConv

metricUnits = [
            "f",    -- 1e-15,
            "p",    -- 1e-12,
            "n",    -- 1e-9,
            "u",    -- 1e-6,
            "m",    -- 1e-3,
            "base", -- 1e0,
            "K",    -- 1e3,
            "M",    -- 1e6,
            "G",    -- 1e9,
            "T",    -- 1e12,
            "P"     -- 1e15
          ]

metricMap :: TypeMap
metricMap = [
            ("f",    m_f),    -- 1e-15,
            ("p",    m_p),    -- 1e-12,
            ("n",    m_n),    -- 1e-9,
            ("u",    m_u),    -- 1e-6,
            ("m",    m_m),    -- 1e-3,
            ("base",m_base), -- 1e0,
            ("K",    m_K),    -- 1e3,
            ("M",    m_M),    -- 1e6,
            ("G",    m_G),    -- 1e9,
            ("T",    m_T),    -- 1e12,
            ("P",    m_P)     -- 1e15
          ]

m_f     x = x * 1e-15
m_p     x = x * 1e-12
m_n     x = x * 1e-9
m_u     x = x * 1e-6
```

```
m_m     x = x * 1e-3
m_base  x = x * 1e0
m_K     x = x * 1e3
m_M     x = x * 1e6
m_G     x = x * 1e9
m_T     x = x * 1e12
m_P     x = x * 1e15

b_B     :: TypeConv
b_KB    :: TypeConv
b_MB    :: TypeConv
b_GB    :: TypeConv
b_TB    :: TypeConv
b_PB    :: TypeConv

binaryUnits = [
            "B",  -- 2**0,
            "KB", -- 2**10,
            "MB", -- 2**20,
            "GB", -- 2**30,
            "TB", -- 2**40,
            "PB"  -- 2**50
        ]

binaryMap :: TypeMap
binaryMap = [
            ("B",  b_B),  -- 2**0,
            ("KB", b_KB), -- 2**10,
            ("MB", b_MB), -- 2**20,
            ("GB", b_GB), -- 2**30,
            ("TB", b_TB), -- 2**40,
            ("PB", b_PB)  -- 2**50
        ]

b_B  x = x * 2**0
b_KB x = x * 2**10
b_MB x = x * 2**20
b_GB x = x * 2**30
b_TB x = x * 2**40
b_PB x = x * 2**50

t_ns        :: TypeConv
t_us        :: TypeConv
t_ms        :: TypeConv
t_s         :: TypeConv
t_min       :: TypeConv
t_h         :: TypeConv
t_hr        :: TypeConv
t_hour      :: TypeConv
t_d         :: TypeConv
t_day       :: TypeConv
t_w         :: TypeConv
t_week      :: TypeConv
t_month     :: TypeConv
t_mon       :: TypeConv
```

```
t_yr        ::  TypeConv
t_year      ::  TypeConv

timeUnits = [
            "ns"     , -- 1e-9,
            "us"     , -- 1e-6,
            "ms"     , -- 1e-3,
            "s"      , -- 1,
            "min"    , -- 60,
            "h"      , -- 60*60,
            "hr"     , -- 60*60,
            "hour"   , -- 60*60,
            "d"      , -- 60*60*24,
            "day"    , -- 60*60*24,
            "w"      , -- 60*60*24*7,
            "week"   , -- 60*60*24*7,
            "month"  , -- 60*60*24*7*30,
            "mon"    , -- 60*60*24*7*30,
            "yr"     , -- 60*60*24*365,
            "year"     -- 60*60*24*365
        ]

timeMap :: TypeMap
timeMap = [
            ("ns",  t_ns)     , -- 1e-9,
            ("us",  t_us)     , -- 1e-6,
            ("ms",  t_ms)     , -- 1e-3,
            ("s",   t_s)      , -- 1,
            ("min", t_min)    , -- 60,
            ("h",   t_h)      , -- 60*60,
            ("hr",  t_hr)     , -- 60*60,
            ("hour",t_hour)   , -- 60*60,
            ("d",   t_d)      , -- 60*60*24,
            ("day", t_day)    , -- 60*60*24,
            ("w",   t_w)      , -- 60*60*24*7,
            ("week",t_week)   , -- 60*60*24*7,
            ("month",t_month), -- 60*60*24*7*30,
            ("mon", t_mon)    , -- 60*60*24*7*30,
            ("yr",  t_yr)     , -- 60*60*24*365,
            ("year",t_year)     -- 60*60*24*365
        ]

t_ns      x = x * 1e-9
t_us      x = x * 1e-6
t_ms      x = x * 1e-3
t_s       x = x * 1
t_min     x = x * 60
t_h       x = x * 60*60
t_hr      x = x * 60*60
t_hour    x = x * 60*60
t_d       x = x * 60*60*24
t_day     x = x * 60*60*24
t_w       x = x * 60*60*24*7
t_week    x = x * 60*60*24*7
t_month   x = x * 60*60*24*7*30
```

```
t_mon     x = x * 60*60*24*7*30
t_yr      x = x * 60*60*24*365
t_year    x = x * 60*60*24*365

s_Hz     ::  TypeConv
s_KHz    ::  TypeConv
s_MHz    ::  TypeConv
s_GHz    ::  TypeConv
s_THz    ::  TypeConv

speedUnits = [
             "Hz",  -- 1e0,
             "KHz", -- 1e3,
             "MHz", -- 1e6,
             "GHz", -- 1e9,
             "THz"  -- 1e12,
         ]

speedMap :: TypeMap
speedMap = [
             ("Hz", s_Hz),   -- 1e0,
             ("KHz", s_KHz), -- 1e3,
             ("MHz", s_MHz), -- 1e6,
             ("GHz", s_GHz), -- 1e9,
             ("THz", s_THz)  -- 1e12,
         ]

s_Hz  x = x * 1e0
s_KHz x = x * 1e3
s_MHz x = x * 1e6
s_GHz x = x * 1e9
s_THz x = x * 1e12

typeMaps = [binaryMap, speedMap, timeMap, metricMap]
```

C.4 Boolean Operations

Listing C.4: *Boolean operations.*

```
-----------------------------------------------------------------
-- Boolean Comparisons
-----------------------------------------------------------------

type Operation = Value -> Value -> Bool

lt       :: Operation
gt       :: Operation
lte      :: Operation
gte      :: Operation
eq       :: Operation
neq      :: Operation
true     :: Operation
false    :: Operation
```

```
lt    x y     = x <  y
gt    x y     = x >  y
lte   x y     = x <= y
gte   x y     = x >= y
eq    x y     = x == y
neq   x y     = x /= y
true  x y     = True
false x y     = False
```

C.5 Basic Pairwise Comparator

Listing C.5: *Basic Pairwise Comparator.*

```
-- Basic Pairwise Comparator
-----------------------------------------------------------------

type OperationMap = [(Match,Operation)]
opMap :: OperationMap
opMap = [
            ("LT",      lt),
            ("GT",      gt),
            ("LTE",     lte),
            ("GTE",     gte),
            ("EQ",      eq),
            ("NEQ",     neq),
            ("TRUE",    true),
            ("FALSE",   false)
        ]

getOp   :: Match        -> Operation
getOp m = head [op | (ma,op) <- opMap, ma == m]

bpc          :: Ch -> Operation -> Ch -> Bool

bpc ca op cb =      ((getChDim   ca) == (getChDim   cb)) &&
                    ((getChType  ca) == (getChType  cb)) &&
              op     (getChValue ca)    (getChValue cb)
```

C.6 Type Transforming Pairwise Comparator

Listing C.6: *Type Transforming Pairwise Comparator.*

```
-- Type Transforming Pairwise Comparator
-----------------------------------------------------------------

ttpc          :: Ch -> Operation -> Ch -> Bool

ttpc ca op cb =             (getChDim       ca) == (getChDim       cb)
       && isComparableType (getChType      ca)    (getChType      cb)
       &&                op (getChBaseValue ca)    (getChBaseValue cb)

getBaseType     :: Type -> Type
```

```
getBaseType t   | member t (fromList speedUnits)     = "speed"
                | member t (fromList timeUnits)      = "time"
                | member t (fromList binaryUnits)    = "binary"
                | member t (fromList metricUnits)    = "metric"
                | True                               = "unknown"

isComparableType        :: Type -> Type -> Bool

isComparableType t1 t2  |             t1  == t2               = True
                        |  getBaseType t1  == "unknown"       = False
                        | (getBaseType t1) == (getBaseType t2) = True
                        | True                               = False

getChBaseValue :: Ch -> Value
getChBaseValue ch = getChBaseValue2 ch typeMaps


getChBaseValue2 :: Ch -> [TypeMap] -> Value
getChBaseValue2 ch []           = getChValue ch
getChBaseValue2 ch (tm:tms)| member (getChType ch)
                           (fromList [t | (t,tc) <- tm]) =
                           (typeConvFromMap (getChType ch) tm) (getChValue ch)
                          | True = getChBaseValue2 ch tms

typeConvFromMap :: Type -> [(Type,TypeConv)] -> TypeConv

typeConvFromMap t [] = unit
typeConvFromMap t ((tm,tc):tms) | t == tm   = tc
                                | True      = typeConvFromMap t tms
```

C.7 Dimension and Type Transforming Pairwise Comparator

Listing C.7: *Dimension and Type Transforming Pairwise Comparator.*

```
-------------------------------------------------------------
-- Dimension and Type Transforming Pairwise Comparator
-------------------------------------------------------------
--      Does not convert types or values based on dimensions
--      (although could, in theory)
--      Checks if dimensions are comparable
--      Converts values to a common base type if possible

dttpc   :: Ch -> Operation -> Ch -> Bool
dttpc ca op cb = isComparableDim     (getChDim   ca)     (getChDim  cb)  &&
                 isComparableType    (getChType  ca)     (getChType cb)  &&
                                  op (getChBaseValue ca) (getChBaseValue cb)

-- The following are simply examples of comparable dimension sets
storageSet      = fromList ["perm_storage", "temp_storage", "disk_storage",
                            "tape_storage"]
benchmarkSet    = fromList ["SI2K", "SPECInt2000","BogoMIPS", "MIPS"]
timingSet       = fromList ["walltime", "cputime", "runtime"]
```

```
costSet          = fromList ["cost"]
speedSet         = fromList ["cpu_speed"]
ramSet           = fromList ["ram", "image_size"]

comparableDimSets =   [
                        storageSet,
                        benchmarkSet,
                        timingSet,
                        costSet,
                        speedSet,
                        ramSet
                      ]

isComparableDim :: Dim -> Dim -> Bool

isComparableDim d1 d2  =  (d1 == d2)
                          || isComparableDim2 d1 d2 comparableDimSets

isComparableDim2 :: Dim -> Dim -> [Set Dim] -> Bool

isComparableDim2 d1 d2 []              = False
isComparableDim2 d1 d2 (dset : dsets)  = (member d1 dset && member d2 dset)
                                         || isComparableDim2 d1 d2 dsets
```

C.8 Example Comparators

Listing C.8: *Example comparators "equivalent" and "ordered".*

```
-----------------------------------------------------------------
-- Equivalent
-----------------------------------------------------------------

equiv :: Ch -> Ch -> Bool
equiv ca cb = dttpc ca (==) cb

-----------------------------------------------------------------
-- Ordered
-----------------------------------------------------------------

ordered :: Ch -> Ch -> Bool
ordered ca cb = ca <= cb
```

C.9 Boolean Conversion

Listing C.9: *Boolean Conversion.*

```
-----------------------------------------------------------------
-- Boolean Conversion
-----------------------------------------------------------------
-- Interpret a set in a boolean context.
--      Empty Set       = False
--      Non-Empty Set   = True
```

```
bool :: [a] -> Bool
bool []       = False
bool (a:as)   = True
```

C.10 Characteristic Subsets

Listing C.10: *Characteristic subsets and related functions.*

```
-- Characteristic Subsets

chDimIntersect :: [Ch] -> [Ch] -> [Ch]
chDimIntersect chsA chsB = [ca | ca <- chsA, cb <- chsB,
                                 (getChDim ca) == (getChDim cb)]

chIntersect :: [Ch] -> [Ch] -> [Ch]
chIntersect chsA chsB = [ca | ca <- chsA, cb <- chsB, equiv ca cb]

isChSubset :: [Ch] -> [Ch] -> Bool
isChSubset chsA chsB = (fromList chsA) == (fromList (chIntersect chsA chsB))

subset :: [Ch] -> [Ch] -> Bool
subset = isChSubset

resChIntersect :: Resource -> Resource -> [Ch]
resChIntersect (chsA,_,_) (chsB,_,_) = chIntersect chsA chsB

part_subset :: [Ch] -> [Ch] -> Bool
part_subset chsA chsB =
        subset (filterChByDimSet chsA (getChSetDims chsB)) chsB
```

C.11 Requirement Subsets

Listing C.11: *Requirement subsets and related functions.*

```
-- Requirement Functions

reqToCh :: Req -> Ch
reqToCh (d,t,m,v) = (d,t,v)

r_equiv :: Req -> Req -> Bool
r_equiv ra rb = (equiv (reqToCh ra) (reqToCh rb))
                && ((getReqMatch ra) == (getReqMatch rb))

-- Requirement Subsets

reqDimIntersect :: [Req] -> [Req] -> [Req]
reqDimIntersect reqsA reqsB = [ra | ra <- reqsA, rb <- reqsB,
                                 (getReqDim rb) == (getReqDim rb)]
```

```
reqIntersect :: [Req] -> [Req] -> [Req]
reqIntersect reqsA reqsB = [ra | ra <- reqsA, rb <- reqsB, r_equiv ra rb]

isChReqSubset :: Ch -> Req -> Bool

isChReqSubset ca rb = dttpc ca (getOp (getReqMatch rb)) (reqToCh rb)

isReqSubset :: Req -> Req -> Bool
isReqSubset ra rb = (getReqMatch ra) == (getReqMatch rb)
                    && isChReqSubset (reqToCh ra) rb

isReqSetSubset :: [Req] -> [Req] -> Bool
isReqSetSubset reqsA reqsB =
           fromList [ra | ra <- reqsA, rb <- reqsB,
                    (getReqDim ra) == (getReqDim rb), isReqSubset ra rb]
        == fromList reqsA

isReqSetRSSubset :: [Req] -> [Req] -> Bool
isReqSetRSSubset reqsA reqsB =
           isSubsetOf (getReqSetDims reqsB) (getReqSetDims reqsA)
        && fromList [ra | ra <- reqsA, rb <- reqsB,
                    (getReqDim ra) == (getReqDim rb), isReqSubset ra rb]
        == fromList (filterReqByDimSet reqsA (getReqSetDims reqsB)))
```

C.12 Resource Matching

Listing C.12: *Resource matching and related functions.*

```
-----------------------------------------------------------------
-- Resource Matching
-----------------------------------------------------------------

getMatchSet :: Resource -> Resource -> Set Ch
getMatchSet (csA,rsA,psA) (csB,[]      ,psB) = empty
getMatchSet (csA,rsA,psA) (csB,(rB:rsB),psB) =
        union (fromList [c | c <- csA, isChReqSubset c rB])
              (getMatchSet (csA,rsA,psA) (csB,rsB,psB))

getSatSet :: Resource -> Resource -> Set Req
getSatSet (csA,rsA,psA) (csB,[]      ,psB) = empty
getSatSet (csA,rsA,psA) (csB,(rB:rsB),psB) =
        union (fromList [rB | c <- csA, isChReqSubset c rB])
              (getSatSet (csA,rsA,psA) (csB,rsB,psB))

getUnsatSet :: Resource -> Resource -> Set Req
getUnsatSet resA resB = difference (fromList (getResReqSet resB))
                                   (getSatSet resA resB)

asymatch :: Resource -> Resource -> Bool
asymatch    resA resB = (fromList (getResReqSet resB))
                        == (getSatSet resA resB)

match :: Resource -> Resource -> Bool
```

```
match resA resB = asymatch resA resB && asymatch resB resA

pmatch :: Resource -> Resource -> Bool
pmatch resA (csB,rsB,psB) =
        asymatch resA (csB, filterReqByDimSet rsB (getChSetDims csB), psB)

peermatch :: [Resource] -> Bool
peermatch ress = and [match resA resB | resA <- ress,
                                        resB <- ress,
                                        resA /= resB]
```

C.13 Sorting by Ranked Preferences

Listing C.13: *Sorting characteristic sets and resources by ranked preferences and related functions.*

```
-----------------------------------------------------------------
-- Preference Functions
-----------------------------------------------------------------
-- Quicksort to sort a set of Characteristics according to a sorted list of
-- preferences
-- NOTE: Doesn't consider Preference.Value, and doesn't behave properly for
--       Preference.Match which includes "equality" (e.g. >=, <=, ==).
--       A real system needs one sorting algorithm per "Match" operation.
chPrefSort :: [Ch] -> [Pref] -> [Ch]

chPrefSort []     []     = []
chPrefSort []     (p:ps) = []
chPrefSort (c:cs) []     = (c:cs)

chPrefSort (c:cs) (p:ps) =
           chPrefSort [c_first | c_first <- cs,
             dttpc c_first (getOp (getPrefMatch p)) c,
             isComparableDim (getChDim c_first) (getPrefDim p)] [p]

         ++ chPrefSort [c_tie   | c_tie <- (c:cs),
             equiv c_tie c,
             isComparableDim (getChDim c_tie) (getPrefDim p)] ps

         ++ chPrefSort [c_last | c_last <- cs,
             dttpc c       (getOp (getPrefMatch p)) c_last,
             isComparableDim (getChDim c_last) (getPrefDim p)] [p]

         ++ chPrefSort [c_other | c_other <- (c:cs),
             not (isComparableDim (getChDim c_other) (getPrefDim p))] ps

getResChSetByDim     :: Resource -> Dim -> [Ch]
getResReqSetByDim    :: Resource -> Dim -> [Req]
getResPrefSetByDim   :: Resource -> Dim -> [Pref]

getResChSetByDim     (cs,rs,ps) dim = [c | c <- cs, getChDim   c == dim]
getResReqSetByDim    (cs,rs,ps) dim = [r | r <- rs, getReqDim  r == dim]
getResPrefSetByDim   (cs,rs,ps) dim = [p | p <- ps, getPrefDim p == dim]
```

C.14 Preference Equivalence

Listing C.14: *Preference ordering equivalence and preference set normalisation.*

```
-- Preference Normalisation and Equivalence

prefToReq :: Pref -> Req
prefToReq (d,t,m,v,r) = (d,t,m,v)

p_equiv :: Pref -> Pref -> Bool
p_equiv pA pB = r_equiv (prefToReq pA) (prefToReq pB) &&
                getPrefRank pA == getPrefRank pB

porder :: [Pref] -> [Pref]
porder []   = []
porder (p:ps) =  porder [p_first | p_first <- ps,
                                   getPrefRank p_first <= getPrefRank p]
              ++ [p]
              ++ porder [p_last  | p_last <- ps,
                                   getPrefRank p < getPrefRank p_last]

pnorm :: [Pref] -> [Pref]
pnorm ps = pnorm2 (porder ps) 1.0

pnorm2 :: [Pref] -> Rank -> [Pref]
pnorm2 [] _ = []
pnorm2 ((d,t,m,v,r):ps) r_new = [(d,t,m,v,r_new)] ++ pnorm2 ps (r_new + 1.0)

prefSetEquiv :: [Pref] -> [Pref] -> Bool
prefSetEquiv psA psB = prefSetEquiv2 (pnorm psA) (pnorm psB)

prefSetEquiv2 :: [Pref] -> [Pref] -> Bool
prefSetEquiv2 []  []  = True
prefSetEquiv2 psA []  = True
prefSetEquiv2 []  psB = False
prefSetEquiv2 (pA:psA) (pB:psB) = p_equiv pA pB && prefSetEquiv2 psA psB
```

C.15 Resource Templates

Listing C.15: *Template equivalence function.*

```
-- Template Functions

tequiv :: Resource -> Resource -> Bool

tequiv resA resB =  isChSubset       (getResChSet  resB) (getResChSet  resA)
                 && isReqSetRSSubset (getResReqSet resA) (getResReqSet resB)
```

Bibliography

[1] I. Stokes-Rees, A. Tsaregorodtsev, V. Garonne, R. Graciani, M. Sanchez, M. Frank & J. Closier. "Developing LHCb grid software: experiences and advances", *Concurrency and Computation: Practice and Experience*, **19**, 2 (2007) 133–152. ISSN 1532-0626.

[2] I. Stokes-Rees, A. Tsaregorodtsev, A. Yu & V. Garonne. "DIRAC Lightweight Information and Monitoring Services using XML-RPC and Instant Messaging". In "Computing in High Energy Physics and Nuclear Physics 2004", (2004).

[3] V. Garonne, A. Tsaregorodtsev, A. Yu & I. Stokes-Rees. "DIRAC : a Scalable Lightweight Architecture for High Throughput Computing". In "Computing in High Energy Physics and Nuclear Physics 2004", (2004).

[4] V. Garonne, A. Tsaregorodtsev, A. Yu & I. Stokes-Rees. "DIRAC : Workload Management System". In "Computing in High Energy Physics and Nuclear Physics 2004", (2004).

[5] N. Brook. "LHCb Computing Model". Technical report, CERN (2004). CERN/LHCC-2004-036 LHCC-G-084.

[6] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach & T. Berners-Lee. "RFC 2616: Hypertext Transfer Protocol – HTTP/1.1". IETF (1999).

[7] T. Berners-Lee, R. Fielding & L. Masinter. "RFC 3986: Uniform Resource Identifier (URI): Generic Syntax". IETF (2005).

[8] R. M. Karp. "Reducibility among combinatorial problems". Plenum Press, New York (1972).

[9] Roy Fielding. "Architectural Styles and the Design of Network-based Software Architectures". Ph.D. thesis, University of California, Irvine (2000).

[10] Ben Carlyle. "The REST Triangle" (2006). `http://rest.blueoxen.net/cgi-bin/wiki.pl?RestTriangle`.

[11] P. Mockapetris. ""RFC 1034: Domain Names - Concepts and Facilities". IETF (1987).

[12] World Wide Web Consortium. `http://www.w3.org/`.

[13] Condor. `http://www.cs.wisc.edu/condor/`.

[14] M. Livny, J. Basney, R. Raman & T. Tannenbaum. "Mechanisms for High Throughput Computing". Speedup Journal (1997). 11(1).

[15] Ian Foster. "The Anatomy of the Grid: Enabling Scalable Virtual Organizations". volume 2150. Springer (2001).

[16] David S. Rosenblum & Alexander L. Wolf. "A Design Framework for Internet-Scale Event Observation and Notification". In M. Jazayeri & H. Schauer, editors, "Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)", pages 344–360. Springer–Verlag (1997).

[17] B. Carpenter. "RFC 1958: Architectural Principles of the Internet". IETF (1996).

[18] F. Carminati. "ALICE Computing Technical Design Report". Technical report, CERN (2005). CERN-LHCC-2005-018.

[19] D. Adams, D. Barberis, C. Bee, R. Hawkings, S. Jarp, R. Jones, D. Malon, L. Poggioli, G. Poulard, D. Quarrie & T. Wenaus. "The ATLAS Computing Model". Technical report, CERN (2005). CERN-LHCC-2004-037/G-085.

[20] C. Grandi, D. Stickland & L. Taylor. "The CMS Computing Model". Technical report, CERN (2004). CERN-LHCC-2004-035/G-083.

[21] Rajesh Raman, Miron Livny & Marvin H. Solomon. "Matchmaking: Distributed Resource Management for High Throughput Computing". In "Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing", Chicago IL (1998).

[22] Dewayne E. Perry & Alexander L. Wolf. "Foundations for the Study of Software Architecture", *ACM SIGSOFT Software Engineering Notes*, **17**, 4 (1992) 40–52.

[23] Jerome H. Saltzer, David P. Reed & David D. Clark. "End-To-End Arguments in System Design", *ACM Transactions on Computer Systems*, **2**, 4 (1984) 277–288.

[24] David D. Clark. "The design philosophy of the DARPA internet protocols". In "SIGCOMM", pages 106–114. Stanford, CA (1988).

[25] David W. Chadwick & Alexander Otenko. "The PERMIS X.509 role based privilege management infrastructure". In "SACMAT '02: Proceedings of the seventh ACM symposium on Access control models and technologies", pages 135–140. ACM Press, New York, NY, USA (2002). ISBN 1-58113-496-7.

[26] R. Housley, W. Polk, W. Ford & D. Solo. "RFC 3280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile". IETF (2002).

[27] J. Vollbrecht, P. Calhoun, S. Farrell, L. Gommans, G. Gross, B. de Bruijn, C. de Laat, M. Holdrege & D. Spence. "RFC 2904: AAA Authorization Framework". IETF (2000).

[28] S. Farrell & R. Housley. "RFC 3281: An Internet Attribute Certificate Profile for Authorization". IETF (2002).

[29] S. Tuecke, V. Welch, D. Engert, L. Pearlman & M. Thompson. "RFC 3820: Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile". IETF (2004).

[30] R. Alfieri, R. Cecchini, V. Ciaschini, L. dellrsquoAgnello, A. Frohner, A. Gianoli, K. Lorentey & F. Spataro. "VOMS, an Authorization System for Virtual Organizations". In "First European Across Grids Conference", Springer-Verlag (2003).

[31] J. Basney. "GFD-E.54: MyProxy Protocol". Global Grid Forum (2005).

[32] T. Ylonen & C. Lonvick. "RFC 4254: The Secure Shell (SSH) Connection Protocol". IETF (2006).

[33] M. Blaze, J. Ioannidis & A. Keromytis. "RFC 2792: DSA and RSA Key and Signature Encoding for the KeyNote Trust Management System". IETF (2000).

[34] J. Callas, L. Donnerhacke, H. Finney & R. Thayer. "RFC 2440: OpenPGP Message Format". IETF (1998).

[35] M. Roehrig, W. Ziegler & P. Wieder. "GFD-I.11: Grid Scheduling Dictionary of Terms and Keywords". Global Grid Forum (2002). Grid Scheduling Dictionary Working Group.

[36] I. Stokes-Rees, A. Tsaregorodtsev, V. Garonne, R. Graciani, M. Sanchez, P. Charpentier, N. Brook, M. Frank & J. Closier. "Live Performance Analysis of the LCG Computational Grid Environment During the LHCb Particle Physics Experiment Data Challenge 2004". In "Proceedings of the IEEE Symposium on Cluster Computing and the Grid (CCGrid2005)", (2005).

[37] S. Belforte *et al.* "CDF Run II Annual Computing Plan and Budget, FY-05". Technical report, FNAL (2004). CDF/DOC/COMP_UPG/PUBLIC/7290.

[38] The D0 Collaboration. "D0 Computing and Software Operations and Plan". Technical report, FNAL (2004).

[39] Standard Performance Evaluation Corporation. "SPEC CPU Integer 2000 Benchmark". `http://www.spec.org/`.

[40] MONARC Architecture Group. "Models of Networked Analysis at Regional Centres for LHC Experiments". `http://www.cern.ch/MONARC/`.

[41] MONARC Architecture Group. "Regional Centers for LHC computing" (1999).

[42] "Memorandum of Understanding for Collaboration in the Deployment and Exploitation of the Worldwide LHC Computing Grid". Technical report, CERN (2006). CERN-C-RRB-2005-01.

[43] "The EU DataGrid Project". `http://www.eu-datagrid.org`.

[44] F. Carminati *et al.* "Common Use Cases for a HEP Common Application Layer". Technical report, CERN (2004). HEPCAL RTAG Report (HEPCAL Prime).

[45] F. Carminati *et al.* "Common Use Cases for a HEP Common Application Layer for Analysis". Technical report, CERN (2003). HEPCAL II.

[46] CERN. "The LHC Computing Grid Project". `http://lcg.web.cern.ch/LCG/`.

[47] P. Buncic *et al.* "Architectural Roadmap Towards Distributed Analysis - Final Report". Technical report, CERN (2003). CERN-LCG-2003-033.

[48] E. Laure. "EGEE Middleware Architecture". In "EDMS 476451", CERN (2004).

[49] I Foster, C Kesselman, J Nick & S Tuecke. "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration". In "Open Grid Service Infrastructure WG", (2002).

[50] D. Adams, P. Charpentier, U. Egede, K. Harrison, R.W.L. Jones, J. Martyniak, P. Mato, J. Moscicki, A. Soroko & C.L. Tan. "The GANGA User Interface for Physics Analysis on Distributed Resources". In "Computing in High Energy Physics (CHEP 04)", (2004).

[51] AliEN. The Alice Experiment.

[52] T. Kosar & M. Livny. "Stork: Making Data Placement a First Class Citizen in the Grid". In "24th IEEE International Conference on Distributed Computing Systems (ICDCS2004), Tokyo, Japan", (2004).

[53] M. Wahl, T. Howes & S. Kille. "RFC 2251: Lightweight Directory Access Protocol (v3)". IETF (1997).

[54] Wolfgang Hoschek. "The Web Service Discovery Architecture". In "Proceedings of the 2002 ACM/IEEE conference on Supercomputing", pages 1–15. IEEE Computer Society Press (2002).

[55] "Universal Description, Discovery and Integration". `http://www.uddi.org`.

[56] Steven Fitzgerald. "Grid Information Services for Distributed Resource Sharing". In "Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01)", page 181. IEEE Computer Society (2001).

[57] B. Tierney, R. Aydt, D. Gunter, W. Smith, V. Taylor, R. Wolsky & M. Swany. "GWD-PERF.16-2: A Grid Monitoring Architecture". Global Grid Forum (2002).

[58] Andrew W. Cooke et al. "Relational Grid Monitoring Architecture (R-GMA)". In "UK e-Science All Hands Conference 2003", (2003).

[59] SETI@Home. `http://setiathome.ssl.berkeley.edu/`.

[60] BOINC. `http://boinc.berkeley.edu.`

[61] distributed.net. `http://www.distributed.net`.

[62] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling & P. Vanderbilt. "Open Grid Services Infrastructure". In "Open Grid Service Infrastructure WG", (2003).

[63] Globus & IBM (2004). `http://www.globus.org/wsrf/#announcement`.

[64] Guy Rixon. AstroGrid Project (2003). `http://wiki.astrogrid.org/bin/view/Astrogrid/GlobusToolkit3Problems`.

[65] Savas Parastatidis, Paul Watson & Jim Webber. "A Grid Application Framework based on Web Services Specifications and Practices and Grid Resource Specification". Technical report, School of Computing Science, University of Newcastle upon Tyne (2003).

[66] Paul Brebner. "Evaluation of Globus Toolkit 3.2 (GT3.2) Installation". Technical report, Software Systems Engineering Group, Department of Computer Science, University College of London (2004).

[67] Paul Brebner, Jake Wu & Oliver Malham. "Evaluating OGSA across organisational boundaries". Technical report, Software Systems Engineering Group, Department of Computer Science, University College of London (2005).

[68] Wolfgang Emmerich, John Darlington, Malcolm Atkinson, Dave Berry & Savas Parastatidis. "Establishment of an Experimental OGSA Grid". Technical report, Software Systems Engineering Group, Department of Computer Science, University College of London (2005).

[69] P. Wilson, W. Emmerich & J. Brodholt. "Leveraging HTC for UK eScience with Very Large Condor pools: Demand for transforming untapped power into results." In "UK e-Science All Hands Conference 2004", (2004).

[70] M. Calleja et al. "Grid tool integration within the eMinerals project". In "UK e-Science All Hands Conference 2004", (2004).

[71] M. Calleja, B. Beckles, M. Keegan, M. A. Hayes, A. Parker & M. T. Dove. "CamGrid: Experiences in constructing a university-wide, Condor-based grid at the University of Cambridge". In "UK e-Science All Hands Conference 2004", (2004).

[72] B. Beckles. "Implementing privilege separation in the Condor system ". In "UK e-Science All Hands Conference 2005", (2005).

[73] B. Beckles. "Building a secure Condor pool in an open academic environment". In "UK e-Science All Hands Conference 2005", (2005).

[74] Dror G. Feitelson & Ahuva Mu'alem Weil. "Utilization and Predictability in Scheduling the IBM SP2 with Backfilling". In "12th International Parallel Processing Symposium", pages 542–546 (1998).

[75] IETF. "Internet Engineering Task Force". `http://www.ietf.org/`.

[76] Jabber Software Foundation. `http://www.jabber.org/`.

[77] Ian Paterson, Peter Saint-Andre & Dave Smith. "JEP-0116: Encrypted Sessions". Technical report, Jabber Software Foundation (2006).

[78] G. Pape. "runit Service Supervision Toolkit". `http://smarden.org/runit/`.

[79] Andrei Tsaregorodsev et al. "DIRAC - Distributed Implementation with Remote Agent Control". In "Proceedings of Computing in High Energy and Nuclear Physics (CHEP)", (2003).

[80] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny & Steve Tuecke. "Condor-G: A Computation Management Agent for Multi-Institutional Grids". In "Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC)", pages 7–9. San Francisco, California (2001).

[81] A. T. Doyle, S. L. Lloyd & A. McNab. "GridSite, GACL and SlashGrid: Giving Grid Security to Web and File Applications". In "Proceedings of UK e-Science All Hands Conference 2002", (2002).

[82] Stuart Patterson. "LHCb Distributed Data Analysis on the Computing Grid". Ph.D. thesis, University of Glasgow (2006).

[83] Vincent Garonne. "Étude, définition, et modélisation d'un Système Distribué à Grande Échelle". Ph.D. thesis, Université de la Méditeranée, Aix-Marseille II (2005).

[84] A. Tanenbaum. "Modern Operating Systems". Prentice Hall (2001).

[85] W. Stallings. "Operating Systems". Prentice Hall, 5th edition (2004).

[86] A. Silberschatz, P. Galvin & G. Gagne. "Operating System Concepts". Wiley, 7th edition (2005).

[87] Richard W. Conway, William L. Maxwell & Louis W. Miller. "Theory of Scheduling". Addison-Wesley (1967).

[88] T.L. Casavant & J.G. Kuhl. "A taxonomy of scheduling in general-purpose distributed computing systems", *IEEE Transactions on Software Engineering*, **14**, 2 (1988) 141–154. ISSN 0098-5589.

[89] D. Lifka. "The ANL/IBM SP Scheduling System". Technical report, Argonne National Laboratory (1995).

[90] Philippe Baptiste, Peter Brucker, Marek Chrobak, Christoph Durr, Svetlana A. Kravchenko & Francis Sourd. "The Complexity of Mean Flow Time Scheduling Problems with Release Times" (2006). `http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0605078`.

[91] David Jackson, Quinn Snell & Mark Clement. "Core Algorithms of the Maui Scheduler", *Lecture Notes in Computer Science*, **2221** (2001) 87.

[92] Platform LSF. `http://www.platform.com/`.

[93] V.Hamscher, U.Schwiegelshohn, A.Streit & R.Yahyapour. "Evaluation Of Job Scheduling Strategies for Grid Computing". In "7th International Conference on High Performance Computing", (2000).

[94] Klaus Krauter, Rajkumar Buyya & Muthucumaru Maheswaran. "A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing", *International Journal of Software: Practice and Experience (SPE)*, **32**, 2.

[95] T. Berners-Lee. "Web Architecture from 50,000 feet". Technical report, W3C (2002). `http://www.w3.org/DesignIssues/Architecture`.

[96] M. Sgaravatto et al. "Pratical approaches to Grid workload and resource management in the EGEE project". In "Computing in High Energy Physics and Nuclear Physics 2004", (2004).

[97] P. Andreetto et al. "CREAM: A Simple, Grid-Accessible, Job Management System for Local Computational Resources ". In "Computing in High Energy Physics and Nuclear Physics 2006", (2006).

[98] F. Carminati & J. Templon. "Preliminary Observations on LCG-2 Based on the 2004 Data Challenges". Technical report, CERN (2004). CERN-LCG-GAG-DC04.

[99] "The Globus Resource Specification Language RSL v1.0". Technical report, The Globus Project (2005).

[100] G. Brown, editor. "System 390 JCL". John Wiley & Sons, 4th edition (1998).

[101] Fabrizio Pacini. "JDL Attributes Specification". Technical report, EGEE (2005).

[102] Ali Anjomshoaa, Fred Brisard, Michel Drescher, Donal Fellows, An Ly, Stephen McGough, Darren Pulsipher & Andreas Savva. "Job Submission Description Language (JSDL) Specification, Version 1.0". Technical report, Global Grid Forum (2005).

[103] D. Bell, T. Kojo, P. Goldsack, S. Loughran, D. Milojicic, S. Schaefer, J. Tatemura, & P. Toft. "Configuration Description, Deployment, and Lifecycle Management" (2003).

[104] Henderson & H. Tweten. "Portable Batch System : External reference specification". Technical report, NASA Ames Research Center (1996).

[105] R. Raman. "Matchmaking Frameworks for Distributed Resource Management". Ph.D. thesis, Dept. of Computer Science, University of Wisconsin, Madison (2000).

[106] C. Liu & I. Foster. "A Constraint Language Approach to Matchmaking". In "14th International Workshop on Research Issues on Data Engineering: Web Services for E-Commerce and E-Government Applications (RIDE'04)", (2004).

[107] Rajesh Raman, Miron Livny & Marvin Solomon. "Resource Management through Multilateral Matchmaking". In "Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)", pages 290–291. Pittsburgh, PA (2000).

[108] INFN/DataTag. "The GLUE Schema Effort". `http://infnforge.cnaf.infn.it/glueinfomodel`.

[109] Fabrizio Pacini. "Job Description Language How-To". Technical report, DataGRID Project (2001).

[110] "Extended Resource Specification Language". Technical report, NorduGrid (2005).

[111] Kenneth Rosen. "Discrete Mathematics and its Application". McGraw-Hill, 5th edition (2003).

[112] Subramanian Kannan, Mark Roberts, Peter Mayes, Dave Brelsford & Joseph Skovira. "Workload Management with LoadLeveler". Technical report, IBM (2001).

[113] H. Casanova. "SimGrid: A toolkit for the simulation of application scheduling". In "Proceedings of the IEEE Symposium on Cluster Computing and the Grid (CCGrid2001)", (2001).