Abstract—We propose a new approach to managing a rapidly evolving grid fabric. The paper begins with a discussion of current configuration systems and their applicability to grid fabrics, followed by a brief analysis of the approaches to fabric configuration of two major grids: TeraGrid and the European DataGrid. We then present our architecture for grid configuration management, and discuss our reference implementation.

I. INTRODUCTION

A. The Configuration Problem

Grids at present are primarily a collection of compute fabrics\(^1\) which have been enabled to share their workload. By pooling their resources, participating sites gain access to a much more powerful system than they would otherwise be able to use. The ability to load balance across the fabrics enables a grid to provide better utilisation of resources for all constituent sites.

When a fabric becomes a part of a grid, it is no longer just required to support the applications of the site to which it belongs. Central to the power of the grid vision is the ability of fabrics to provide similar capabilities in terms of application support. Not only must a fabric support the applications of partner sites at the time of its entry, it must also be able to adapt to support the applications of any new partner sites that subsequently join.

The range of applications a particular fabric may be required to support is vast and highly dynamic. This makes creating and maintaining a suitable configuration for each node in the fabric a significant challenge for its administrators.

As was noted at the inception of TeraGrid [1], attempting to fix a single configuration for all fabrics and for all time amounts to stagnation and atrophy. The configurations of the nodes present in a grid must evolve over time, but they must also remain mutually compatible, since it is vital that applications not be dependent upon which of the constituent fabrics they execute.

Different applications from different sites may also have conflicting configuration requirements. This may be for a variety of reasons. The most common clash in practice is between required versions of dynamically linked libraries. In an ideal world, minor library version differences would not change the semantics of a program, but this can be hard to guarantee. Application users are often unwilling to run their software against untried versions of a library.

It is also possible for an application to depend upon a particular executable. Whereas the shared library system was designed to allow multiple major library versions to coexist, executable packages rarely accommodate multiple installation in any standard fashion. These types of clashes can be very difficult to resolve.

The final type of configuration conflict is at the fundamental level. A conflict in required C library versions or in kernel set-up is very difficult to resolve. A particular application may require disk servers whose kernel configuration has been tuned to optimise their performance for a given workload. Conflicts in C library version can arise when using “bleeding edge” software linked against unstable (development) versions of the C library (typically an issue with glibc under Linux).

B. Configuration Systems

This section provides a brief overview of existing configuration systems as they relate to grid fabrics. Current configuration systems are on two basic levels, within which there are a number of broad categories, outlined below. Not all of these should be considered suitable for adoption in a grid fabric, however we believe all are presently in use.

1) Low-level Configuration: The first major type of configuration system deals with the low level configuration of a node. First amongst these is the network-level declarative tool. Here a configuration daemon takes a specification file detailing the required properties of a node and works on the node to attempt to realise those properties. LCFG\(^2\) [2] [3] and Quattor\(^3\) [4] are both examples of this kind of system.

Each node is identified with a single configuration file, called its profile which specifies what roles the node should play, what packages the node should have installed, and how those packages should be configured. The major strength of this approach is in the ability to share structured configuration information between nodes. By specifying a set of packages or configuration details for a given group, a node has but to be declared a member and all relevant configuration details will be taken care of. This also enables a separation of concerns, with different staff members or teams maintaining different roles.

---

\(^1\)We use the term fabric to mean a single computing element or cluster from henceforth.

\(^2\)Local Configuration System http://www.lcfg.org/

\(^3\)Quattor is an Administration Toolkit for Optimising Resources http://www.quattor.org/
Systems like LCFG and Quattor centralise all the configuration data for a network at a configuration server. They provide the ability to migrate node configurations and to restore failed nodes on new hardware, since all data relevant to the configuration of a given node is present in the configuration source. These systems do not require multiple installation of software packages, since they perform real software installation, however they do not support giving different users different views of the same system: any configuration change made to a node’s profile is a real configuration change seen by all users of that node. This is a key distinction between any low-level configuration system and the virtualisation systems discussed in section I-B-2.

In a grid context, LCFG has been used to maintain a shared “grid node” role exhibited by all fabric nodes at all sites. This has the advantage of guaranteeing a consistent configuration for all nodes in a fabric, however establishing the content of this file is an ongoing problem in such a system. This is discussed further in section II-A.

Imaging systems are popular large scale configuration tools, often used in the maintenance of clusters. They are most suited to large pools of identical hardware, where the differences between individual machines are very small and may be managed using simple scripts. In a grid, it is unlikely that all the constituent fabrics are so similar, making the sharing of images between sites very difficult. This forces each individual site to create its own images, placing a great burden on fabric administrators as such images are complex and rapidly evolving.

The other category of low level configuration system is characterised by a procedural approach, where the configuration of a system is described and enforced by a sequence of operations. These might be carried out by shell scripts, a tool like cfengine [5], manually or any combination of these. There is no declarative statement of the system’s configuration, and often a particular node’s configuration is difficult to even estimate.

Although a strategy like this is maintainable for small numbers of machines, or at sites where the rate of configuration change is low, it does not scale well to large numbers of machines nor does it deal well with highly dynamic configurations. When the number of machines is large, any broad change to the site configuration typically requires reworking a large number of scripts. Manual configuration changes are almost invariably unmanageable when there are a large number of nodes. On a computing element of more than a few tens of machines, no per-node changes should be made manually, as mistakes are almost certain to occur, and the time required to perform broad changes is likely to be very great.

In the case of a grid fabric, both these factors combine: there are often a large number of nodes and the rate of configuration change must be expected to be very high. For this reason, we suggest a procedural approach is fundamentally ill-suited to the management of a grid fabric.

2) Virtualisation Technologies: Virtualisation technologies separate the actual, low-level, configuration of a node from the

---

4User Mode Linux
hit.

A third class of virtualisation tools, requiring specialised network hardware, enables real machines to use filesystem images (just like virtual nodes do under UML or VMWare) without the normal security concerns. This is network virtualisation, as exemplified by Hewlett-Packard’s UDC [11]. Each node has a view of their local network, which may be modified in software. This gives native performance with many of the advantages (from a configuration perspective) of virtualising the execution environment.

The Chiba City cluster [12] at Argonne National Laboratories provides similar capabilities to those of network virtualisation. The principal difficulty experienced in the cluster’s operation is the recovery of failed nodes, since hardware resets will sometimes be required when exposing real hardware to arbitrary images. The solution devised for Chiba places the power supplies to nodes under network control, allowing controlled power cycling even when a node has failed. Some concerns were expressed, however, about the effect of these measures on the service life of nodes.

II. EXISTING SOLUTIONS

As part of our investigation of requirements for a dynamic reconfiguration system, we visited two existing grids, TeraGrid and the European DataGrid, to examine the techniques they use for configuration management. A brief overview of these findings is presented here to motivate our approach, which will be presented in section III.

A. The Uniform Approach in the EDG Testbed

The European DataGrid’s testbed used a single standard configuration technology at all sites. This was a modified version of LCFG [2] that came to be called EDG-LCFG. As discussed in section I-B.1, one advantage of having a single standard tool at a grid level is the ability to share configuration data between sites.

The EDG testbed provided standard configuration files specifying the key parts of a grid node’s configuration, similar to the TeraGrid’s configuration contract (see section II-B), but in machine readable form. These files, maintained centrally at CERN, were used by all participating sites, allowing a very high degree of consistency between the configurations of the sites, with little human intervention at the fabric level. The result was a highly homogeneous grid, one of the first key requirements identified in section I.

The EDG case study also highlights the limitations of using a system configuration tool on its own, however. Although EDG-LCFG was sufficient to keep the configurations of nodes in participating fabrics synchronised, it was not able to provide different configurations to different users, or to different jobs, based upon their requirements. EDG-LCFG (like LCFG) creates and maintains a single globally visible view at a node.

This becomes a significant problem when, as previously discussed, different applications have conflicting requirements. In these cases, significant negotiation between application developers and fabric maintainers is required to identify a single shared configuration within which all parties can coexist. The result is often a brittle and highly “dense” configuration, where each aspect is tied to a given value by a number of dependent applications. Supporting each new application requires incrementally more work on the part of both the developers and grid maintainers.

The EDG integration team (as part of work package 4 on fabric management [13]) showed considerable success in re-engineering existing applications to bring their dependencies into line with those of other grid applications. This approach, however, does not scale, does not lead to applications being portable across a range of grids and is also expensive in developer time.

A further problem with the uniform approach, as adopted by the EDG testbed, is the relatively slow turnaround on making new application versions available for grid use, as each application change must be integrated into the central configuration files. In the EDG testbed it could take up to 24 hours for a new application version to become live, too slow for application developers trying to test new versions. In addition, LCFG requires software it manages to be in RPM format, and creating RPMs is a relatively difficult procedure that many application developers found unnecessarily complicated. This lead to various subversions of the system, many of which were very wasteful of resources.

B. The Contract Approach in TeraGrid

In contrast to the uniform approach adopted by the EDG testbed, TeraGrid uses a much more powerful system based on the environment configuration system SoftEnv [6], discussed

<table>
<thead>
<tr>
<th>Method</th>
<th>None</th>
<th>Environment</th>
<th>System-level</th>
<th>Virtualised OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexibility of job-specific configuration</td>
<td>very low</td>
<td>low</td>
<td>high</td>
<td>very high</td>
</tr>
<tr>
<td>Simultaneous jobs with conflicting requirements</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Time to switch configurations</td>
<td>N/A</td>
<td>very low</td>
<td>potentially high</td>
<td>low</td>
</tr>
<tr>
<td>Repackaging needed</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Native job performance</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>

Fig. 1. A comparison of reconfiguration strategies
in section I-B.2. Unlike the EDG however, TeraGrid does not mandate a particular low-level configuration system, so that the low-level configuration consistency benefits reaped by the EDG are not available to the TeraGrid.

Instead of tool-enforced configuration consistency, TeraGrid sites are required to fulfill a configuration contract written in human readable form. This explicit, written contract is to be fulfilled by all TeraGrid nodes, and is published and maintained online. Each contributing TeraGrid site is responsible for ensuring that its nodes meet the contract at all times.

TeraGrid maintains a centralised configuration monitor for the entire grid by regular submission of test jobs to each fabric. These jobs simply check that given SoftEnv tags provide the software versions specified by the contract. All the results of the tests are reported back to a repository, where they are collated and made available for display.

Although all the information about a site’s compliance or non-compliance with the configuration contract is available centrally in machine-readable form, there is no further action that can be taken in the TeraGrid system without the intervention of a human operator. Because low-level configuration systems are left explicitly unspecified, and the high-level configuration system in use, SoftEnv, is not capable of modifying a node’s state, no automatic adaptation is possible. Hence TeraGrid provides automatic configuration monitoring, but not automatic configuration consistency; the burden of maintaining consistency falls entirely upon fabric management teams.

Furthermore, although use of SoftEnv reduces the workload of developers, by reducing the need to rework their applications to fit into a single configuration view, it greatly increases the burden on site administrators who must repackage and maintain multiple versions of each piece of software at each node in their fabric. This exacerbates the configuration problem further in the TeraGrid at present, we believe.

We note that although the contract system might be expected to better handle the reality of large, heterogeneous grids composed of many different fabrics, the replication of configuration effort at each site through the inability to share information places a burden upon administrators that is especially felt in a large grid. Given the difficulties we observed many sites having in meeting the contract, we suspect the manpower available to maintain such configurations is at present insufficient.

The contract approach would indeed be a better solution if a single configuration language could be used as an input to the tool of the site administrators choice. The contract could then be written in that language and greatly reduce the workload of site administrators. In the absence of such a language (all existing tools at the time of writing use their own individual language), this approach suffers from scalability problems, of a different kind but no less acute than those of the uniform approach adopted by the EDG testbed (section II-A), we attempt to adapt the contract approach of TeraGrid to be both more flexible and require less intervention from fabric administrators. We present first an overview of our approach, followed by a detailed examination of the architecture of our test system.

A. Overview

In the abstract, the approach taken by TeraGrid is to specify which configurations may be requested from a node in the grid, and exactly how they are to be requested. This is the essence of the contract approach when used environment reconfiguration as it is in TeraGrid. The major disadvantages of this approach, as noted in section II-B, are the difficulty in maintaining the underlying “real” node configurations without centralised support and the difficulty of installing multiple versions of a package for SoftEnv to provide.

Our approach is to generalise this architecture, so that instead of specifying that SoftEnv must be the system which carries out reconfiguration at the request of the user, any capable configuration system may be used. The user is not allowed to know or care how their requested configuration is arrived at.

We distinguish between the general system architecture which we have developed for dynamic reconfiguration, and the specific test system which is a proof of concept instantiation of the architecture. The test system uses three existing configuration systems, SoftEnv, LCFG and UML to perform reconfiguration. It is important to note that reconfiguration need only be from the user’s perspective. When using either SoftEnv or UML no low-level node reconfiguration takes place, however the user’s view of the node changes giving us effective reconfiguration.

1) The Reconfiguration Architecture: Our architecture for reconfiguration is based on users selecting the correct configuration for their job explicitly, using a configuration reference. A job should commence by switching to its desired configuration, using a reference previously negotiated on behalf of the owner. The underlying configuration technologies that perform the actual reconfiguration of the node are unspecified, but may be virtual or system level as defined earlier. Negotiation occurs through a web service interface, which accepts configuration requests and returns a reference which can be used on a node to request that configuration.

The most significant gains from using this architecture will be made when multiple configuration technologies are used to perform reconfiguration, including a standard centralised system-level tool to simplify sharing of configuration details between sites, or a standard virtual operating system tool whose filesystems may be shared between sites. More research is needed to develop a suitable language for general configuration requests, our prototype only handles package versions.

2) The Reference Implementation: The reference implementation consists of 4 main parts:

1) A configuration switcher which, when called on a node and given a reference, reconfigures the node accordingly.
Ours simply checks if a suitable UML filesystem exists, and if not asks both SoftEnv and LCFG to reconfigure to a pre-agreed schema.

2) A configuration negotiator web service which takes package requests and returns a reference to the user. Ours generates files for LCFG (and via LCFG, for SoftEnv) containing the given packages.

3) A “contextifier” which collates the negotiated package requests for LCFG into a single file, triggering LCFG to pass out the new configurations to all nodes in the fabric.

4) A client which enables users to participate in limited configuration negotiations.

B. Distribution and Selection

There are two ways of performing on-the-fly node reconfiguration for jobs. The model used by SoftEnv requires jobs to explicitly request their required configuration. This approach requires the user to know the configuration requirements of their job, either as a reference identifying those requirements (as per SoftEnv) or as something more detailed (for example, scripts that perform manual reconfiguration are not uncommon in the EDG).

An alternative approach we considered was to encapsulate the knowledge of the configuration required for an application within the application invocation itself. That is, by running a particular application, the user causes the necessary reconfiguration to support that application to occur. We term this implicit reconfiguration. It has the benefit of hiding the from the user the knowledge that any reconfiguration is taking place, allowing it to be used side by side with systems that are not performing any reconfiguration (i.e. statically configured machines, as in the EDG testbed described in section II-A).

The major drawback to implicit reconfiguration is that it cannot be used when users can provide their own applications as part of a job (for example, when a job commences by installing the user’s application). Explicit reconfiguration will always be necessary in this situation because the system cannot know about every possible user-submitted application. This was a use case we felt it was important to support, having seen its necessity for users of the EDG testbed.

Within explicit reconfiguration, there are two further subclasses of architecture. An architecture may be based upon the request of detailed configuration aspects by a particular job, for example, particular versions of each package it requires, along with some information relating to their configuration. This would then be translated at the node into a particular system configuration and the node modified accordingly. We term this a configuration fragment approach because individual reconfiguration requests are structured information in a configuration language.

The alternative is a configuration reference approach where requests are simply unique labels identifying some pre-agreed configuration. For example, an experiment might negotiate a particular reference which, when given to any particular node in the grid, causes an agreed, experiment-specific, configuration to be instantiated on that node.

Our decision to adopt a configuration reference approach comes primarily from the bias of existing configuration technologies towards centralisation. As noted in section I-B, many of the current configuration technologies are oriented around a central information repository, and the configurations at nodes within a fabric are expected to be derived from that repository (this is true for SoftEnv, LCFG, Quattor, and a host of others). Locally negotiated configuration changes are difficult to integrate into such a system. A configuration reference approach allows negotiation to be done once, centrally, and the results propagated to each site in a grid in a hierarchical fashion.

C. Configuration Negotiation

Any architecture based upon explicit reconfiguration must have some form of configuration negotiation, so that users’ configuration requirements can be recognised by the system. In TeraGrid, a set of SoftEnv tags and their meanings are negotiated manually by application developers and grid administrators. This works, but is not scalable nor is it cost effective, since much time is wasted by administrators and developers in face to face negotiation.

A better solution is to automate configuration negotiation so that human intervention is limited. This will become increasingly important as grids spread to take on more participating sites and experiments and the range of different configurations (or competing configuration requirements) grows significantly.

The CDDLM\(^5\) working group of the GGF\(^6\) is currently developing an approach to application configuration and deployment for grid fabrics based upon web services. The aim is to create a standard interface for describing and deploying applications across a grid fabric. However, much of the group’s emphasis is on deploying applications made up of grid services and persistent service applications like web-servers. This has lead to an unfortunately poor fit with the requirements of existing e-science grids like the TeraGrid. These grids do not use web services internally in any significant way, nor are the applications they wish to support exposed as web services. Rather than needing to deploy complex fault tolerant web servers, these grids are more concerned at present with making a large number of legacy applications with different configuration requirements co-exist within a single effective computing element.

Our approach to reconfiguration, for these reasons, does not follow the CDDLM model in any but the broadest sense. Users or developers submit a configuration request to a central configuration broker (built as a web service in standard fashion). This system identifies if the configuration requirements can be met by the software already available on the grid, and if it so, returns a unique identifier (reference) for that configuration, which may be used by any job executing at any node in the grid to change that node’s configuration to the one requested.

---

\(^5\)Configuration Description, Deployment and Lifecycle Management
\(^6\)Global Grid Forum http://www.ggf.org/
A configuration request, in our proof of concept implementation, only encompasses a list of packages and versions which the configuration must supply. More work is needed to broaden this interface so that more complex system level configurations can be performed. This is discussed in section IV-A. Automatically adding software to the fabric contract would be a valuable extension in this area, however the difficulties of maintaining grid level consistency were noted in section II-B.

D. Configuration Instantiation

As outlined in section I-B, there are a variety of configuration tools and systems suitable for use in a grid context. We chose three, one from each of the major categories, to provide the reconfiguration functionality for our proof of concept implementation. We discuss the way in which each is used in the following sections.

All reconfiguration, regardless of the mechanism by which it is provided, is initiated by running nodeswitch reference, where reference is the result of the configuration negotiation process discussed in section III-C. The results of this command can vary in principle from the user’s environment being rebuilt, to the node itself being rebuilt. In a production system, the likely cost of reconfiguring a given node would need to be taken into account. This potential enhancement is discussed in section IV-B.

1) Environment Reconfiguration with SoftEnv: SoftEnv provides a way to change a user’s view of the configuration of the system. Although the software present on the node does not change, those parts of that software that are visible to the user do. SoftEnv is intended to use a central database detailing the locations of each version of each application on each platform in the system. Various wildcard operators are available to improve the readability of the specification file.

Users select their required applications by creating a .soft file in their user area, containing entries for all the applications and versions they require. Administrators often create some dummy application versions to stand for the latest available version of a given package. When a user logs in, the SoftEnv system is run and generates a set of environment variables for the user based upon the combination of the user’s application list and the administrator database.

In our test implementation, each configuration reference that is negotiated leads to a .soft file being generated listing the applications and versions that were agreed for that reference during the negotiation. This file is copied to the job’s owner’s home directory, and resoft is invoked. If the reconfiguration does not involve SoftEnv an (essentially empty) file will be copied to the user’s home directory, but the logic is the same. The resoft alias causes the environment to be rebuilt as if login had just occurred.

2) System-level Reconfiguration with LCFG: LCFG is traditionally used for static configuration management, and our test fabric also uses it for this purpose. However, recent versions of LCFG also provide a limited ability to perform dynamic reconfiguration. This mechanism is called contexts and it provides a way to give configuration aspects different values dependent upon some runtime criteria.

We harness contexts as follows. The parts of a negotiated configuration which are to be met in system-space by LCFG are placed in a file with the same name as the reference to the configuration to which they belong. A relatively primitive tool named contextifier is then run upon the entire directory of negotiated configuration files, generating a single LCFG

![Fig. 2. The system architecture](image-url)
source file containing all the necessary context information.

The net result is that after a successfully negotiated configuration making use of LCFG, each node gains a new context to which it can switch, on-demand, via the context command. The propagation time between a configuration being successfully negotiated and available at the nodes in the fabric is typically small in our test cluster, although it would be expected to rise marginally in a large scale cluster. LCFG is known to scale to systems of the order of 1000 nodes, with only moderate propagation time for configuration changes (almost immediate unless the system is under heavy load or the network is partitioned, and invariably less than a few hours).

When nodeswitch is invoked the LCFG context command is run with the reference identifier as an argument. In the event that no LCFG configuration data has been identified with that reference, the node reverts to its default, baseline configuration. Otherwise the node will switch to the baseline configuration augmented by whatever parameters were specified in the LCFG source file from the negotiation process.

LCFG is capable of much more drastic reconfiguration than simply changing RPMs, however the lack of a suitable generic configuration language has prevented us from harnessing this functionality at present. Using LCFG to switch RPMs has big advantages in terms of security that in that users need not be given privileged access to a node. There are also significant bandwidth savings since LCFG is capable of performing caching.

We intend to investigate more complex reconfigurations based on the abilities of LCFG in the future.

3) Virtualisation with UML: UML is not typically thought of as a configuration system, however in a manner similar to SoftEnv, it can be used to allow a node to provide different views of itself according to the requirements of the users. Virtualisation approaches are most useful in allowing dramatic configuration changes that would be too difficult or expensive for a node to otherwise perform, although they may also be beneficial when a new configuration must be rapidly supported.

Our use of UML in providing dynamic reconfiguration amounts to selecting a suitable root filesystem based upon the reference passed to nodeswitch. Given a configuration reference, if a matching UML root filesystem exists, a UML instance is created using this filesystem, and both SoftEnv and LCFG reconfiguration are skipped.

In a production system, it would be preferrable to have more complex logic to determine which form of reconfiguration is used if there are multiple choices. Although LCFG and SoftEnv are generally orthogonal in terms of the configuration choices they provide for a given node (if you have multiply installable versions of packages, theres little benefit in being able to install them on the fly), UML provides an alternative to both. The correct choice between initiating a virtual instance (quick to start, but a performance penalty for the duration of the job) and performing reconfiguration (potentially slow to start, native speed thereafter) is dependent upon the length of time for which a job will execute.

As for LCFG and SoftEnv, the origin of the filesystem is expected to be the negotiation process. We offer one such process in the form of a simple web service capable of encoding different package version requirements into LCFG and SoftEnv source (in fact, it is all encoded into LCFG source, and some of that source is used to generate SoftEnv source, but theres no particular need to do it this way). However, our automated negotiation process is not capable of generating UML root filesystems on the fly, so if a UML instance is required to provide the reconfiguration for a given reference, manual intervention will be necessary to create this filesystem.

In order to insulate users of the system from the details of how reconfiguration is performed, a slightly modified UML root filesystem is needed. In particular, if a typical UML filesystem is used, the user will need to login (again) before their job can execute. To prevent this, the filesystem’s initialisation scripts (typically /etc/inittab) need to be modified so that a default user is automatically logged in rather than mingetty being run. This is a relatively simple change.

Using UML incurs space overheads since each root filesystem is often quite large. If the filesystems are downloaded on demand from the network to save disk space (the alternative is replication at each node, which is somewhat expensive) then there is an associated bandwidth cost when starting jobs in a virtualised configuration.

IV. CONCLUSIONS AND FURTHER WORK

A. Improving Configuration Negotiation

The proof of concept implementation of our architecture allows only the specification of a list of required packages and their versions. It is incapable of generating a UML filesystem containing the given configuration, although the back-end is capable of using such a filesystem if it exists (presumably due to its having been manually created in the present system).

The underlying approach and tools will scale to much more complex reconfiguration, including changing the configuration of particular packages or services on the node and changing fundamental operating system components like the C library and the kernel. Although there are minor technical obstacles to overcome in order to allow full-scale reconfiguration of a node to take place in response to a job configuration request, the primary difficulties rest in being able specify (in a system independent way) what configuration should take place.

There are three major configuration languages presently in widespread use. LCFG has its own high level specification language, a language called Pan was developed for the Quattor system, and SmartFrog (see [14]) contains a high level deployment language that is used for some types of reconfiguration. All three languages are closely tied to the underlying configuration systems they were written for, although the CDDLM working group are looking at ways to broaden the scope of SmartFrog as part of the standardisation effort.

One alternative to a true declarative language for configuration specification might be to use test code as in TeraGrid. In TeraGrid, such code provides the definition of a compliant node: any node successfully executing the tests is considered
to be compliant with the TeraGrid configuration contract, and as such the tests themselves become the contract. It is an open problem whether a system could automatically derive its configuration requirements from code like this.

The only obvious alternative to code analysis for a generalised negotiation framework is to have a single standardised configuration language which all participating sites can comprehend and manipulate. The present trend in configuration languages appears to be towards diversification, meaning that this is likely to force each site to use a common configuration system. This approach, used in the EDG testbed (see section II-A) has many technical advantages, but can be very problematic in the real world due to political difficulties.

Although Quattor is somewhat more tolerant, LCFG does not coexist well with other configuration systems, nor does it allow much scope for manual intervention in the configuration of aspects of nodes of which it is in control. All configuration information is typically forced to flow through the tool. This can create significant political obstacles, as sites are rarely keen to adopt a new configuration technology and hand it complete control of an entire fabric.

B. Integrating Reconfiguration Concerns into Scheduling

Reconfiguration can create efficiency concerns when used extensively within a grid or fabric. The time taken to reconfigure a node from one state to another may be significant, and this changes the strategy a scheduler should pursue to optimise fabric usage and job throughput. However, current schedulers accept only very limited kinds of information about the cost of executing a given job at a particular node.

The fundamental change to the scheduling environment when dynamic reconfiguration is taking place is that the cost of running a job on particular node becomes context dependent, with (at least) the previous job able to affect the time the next takes to complete. Depending on the style of reconfiguration used, unimportant configuration aspects may be allowed to remain between configuration switches, leading to the requirements of many previous jobs being involved in calculating the reconfiguration time of a current or future job.

Future work in this area could determine what degree of detailed information a node could be expected to provide regarding the cost of executing a given job after a given sequence of jobs, how it might be expected to calculate that cost, and how that information should be made available to schedulers in a system-independent way. Our proof-of-concept implementation does not provide any information for a scheduler to make use of.

C. Scaling the Architecture to the Grid Level

Our implementation provides a fabric level broker for configuration requests, rather than a grid level broker. In order to use a system like ours across a collection of fabrics, an additional component to perform grid level negotiation and dissemination to individual fabrics is required. This is necessary to make sure that the same identifiers can be used on nodes on each fabric, otherwise users must attempt to determine which fabric they are on, and use a fabric specific reference.

The implementation of such a grid level broker need only identify whether a request can be met by the software currently mandated to be available on its constituent fabrics, and return an identifier for the request if so. It can (and must) pass the details of what reconfiguration should be performed and how to the individual fabric brokers to allow fabrics to use different technologies to instantiate a configuration.

V. Summary

Dynamic reconfiguration provides a promising alternative approach to configuration management on complex and evolving clusters. Grid fabrics today present the foremost challenges in this area, and there is evidence that dynamic reconfiguration can help greatly.

VI. Acknowledgements

This work is part of the OGSAConfig project7 sponsored by JISC8. We would like to thank Remy Evard and Narayan Desai of Argonne National Laboratories and German Cancio of CERN for their help.

References


7http://groups.inf.ed.ac.uk/ogsaconfig
8The Joint Information Systems Committee http://www.jisc.ac.uk