ICENI II Architecture

A. Stephen McGough, William Lee, and John Darlington

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

Abstract. The Imperial College e-Science Networked Infrastructure (ICENI) has been developed by the London e-Science Centre for over four years. ICENI has prototyped many novel ideas for providing an end to end Grid middleware. This has included: Service-Oriented Architecture, component programming model, retaining and using metadata collected throughout the life-cycle of an application, and scheduling algorithms which are aware of workflow and performance data. In this paper we evaluate ICENI in terms of the projects where it has been deployed to determine the strengths and weaknesses of the ICENI system. We present our development of ICENI II, which maintains the good architectural design of the original ICENI and overcomes the weaknesses in the current implementation. Further, we outline the higher level services that are essential to Grid development.

1 Introduction

The ICENI (Imperial College e-Science Networked Infrastructure) [2, 10] service–oriented middleware concept originated from the research activities of Professor John Darlington and colleagues in the development and exploitation of functional languages. This culminated in the development of the ICENI I software implementation. Our focus within the ICENI concept has three major elements: prototyping the services and their interfaces necessary to build a service oriented Grid middleware; developing an augmented component programming model to support Grid applications; to explore the meta-data needed to enable effective decision-making; and the development of higher level Grid services.

In this paper, we discuss the uptake of the original ICENI implementation within the wider Grid community, and the views and comments from groups and individuals who have used ICENI within their projects (Section 2). From this, we determine a set of perceived weaknesses within the current implementation and discuss how they can be addressed (Section 3). From this we re-focus the ICENI concept more towards the high level services such as semantic coupling and implementation selection. We then present a new implementation architecture for ICENI II which follows the original ICENI concepts, taking the good and novel ideas and addresses the weaknesses previously noted (Section 4). We further detail this in sections 5,6,7,8, before commenting on the current development of ICEIN II and concluding (9).

2 Analysis of ICENI I

ICENI I has now had exposure in the wider Grid community through our involvement with projects and uptake through interested third parties: the GENIE project has used ICENI to Grid enable their unified Earth System Model, allowing them to vastly reduce their run time [14]; the e-Protein project uses ICENI to control gene annotation workflows [23]; the Immunology Grid project is using ICENI in areas of molecular medicine and immunology [27]; and visualisation and steering of LB3D simulations have been automated for use in the RealityGrid project [7]. Although these projects have demonstrated the correctness of the ICENI concept, we have identified several shortcomings of the ICENI I implementation.

Through feedback we have been assessing the usability of the current ICENI implementation. Although we still feel that the underlying concepts and architectural ideals developed are sound, it is apparent that the current ICENI implementation has become overburdened through the process of software decay. Suffering from a prolonged process of incremental design and development. As new features were incorporated into the implementation over time the original ideals of the ICENI concept were lost. This resulted in the current realisation being cumbersome to install as each feature requires a full installation of the whole ICENI architecture. However, none of these factors are intrinsic to the underlying ICENI architecture.

The Grid community has, over the last few years, been moving rapidly between different underlying middleware architectures, this has led to ICENI seeming somewhat outdated by its original selection of the JINI [22] architecture. Stability is now being approached within the Grid community with most developers moving towards Web Service oriented middlewares. The increased use of firewalls between sites on the Internet has caused significant problems when using JINI. As JINI was developed for use in a local environment without firewalls it uses a large range of ports when communicating between instances. This has caused major problems when deploying ICENI between administrative domains. This problem has been resolved by the development of JINI2. However, it was felt that Web Services provided a better solution.

3 Aims of ICENI II

From these observations we have come to the conclusion that a re-factoring of the ICENI implementation is required. The main aims of this re-factoring, to be known as ICENI II, are outlined below:

- Develop ICENI on top of Web Services. ICENI has always been architectured in a communications agnostic manner. It would now appear correct to develop ICENI on top of Web Services whilst still retaining this agnostic approach.
- Decompose the ICENI architecture into a number of separated composable toolkits, each of which can be used separately to perform tasks within the Grid. Alternatively these toolkits, and those from other Grid technology developers, can be used in an à-la-carte fashion with the sum functionality being greater than that of it's parts.
- Reduction of the footprint of ICENI on resources within the Grid. A barrier to the adoption of ICENI has been the amount of code that needs to be deployed onto a resource. By making most of ICENI optional, only those bits that are required need to be installed.
- Tightly defining the functionality of each of the toolkits. This should minimise software decay and allow each toolkit to focus on one goal.
- As the underlying fabric of the Grid becomes more stable it is now possible to develop higher level services. Thus enabling a full end to end, transparent, Grid pipeline from application concept through to execution. We seek to re-focus ICENI on these higher level services which are required to make the Grid useful for the application scientist.

3.1 The ICENI programming model

ICENI uses the component programming model in which an application is constructed from a number of self contained programming elements referred to as components. Each component has a clearly defined interface. Each component defined at the level of meaning (e.g. linear solver) that participates in the workflow may have multiple behaviors (e.g. pull data in, push data out) each of which has multiple implementations (e.g. Cholesky implemented in C) for different input parameters (such as matrix type for numerical method components) and also for different execution platform architectures (e.g. linux vs. MS Windows). Further information on the ICENI component model can be found in [19, 20].

4 Architecting ICENI II

The process of constructing the ICENI II architecture is currently in progress. Figure 1 shows the design diagram for ICENI II. Within this design we have determined that there are three main stages within an application's life-cycle:

- **Specification** of the application to be executed. This may be as simple a process as specifying a single task to be executed or as complex as defining an entire workflow that requires the orchestration of many components together to form the overall application. Without loss of generality we shall consider only workflows from this point on as a single task may be described in terms of a workflow. At this stage the workflow may be abstract in nature - the code that will be used to implement the tasks and the resources on which they run are not yet defined. The translation of an application scientist's ideas into these workflows is a complex task. It cannot be assumed that the application scientist will be an expert with workflow languages, we therefore see the need to develop application specific interfaces which allow the scientist to describe their problem in their own space - translating this down into a workflow.
- **Optimisation** of the workflow. Once the workflow has been generated it needs to be optimised before it can be deployed onto the Grid. The Optimisation stage selects the best use of resources and component implementations available on the Grid in an attempt to match the criteria specified by the application scientist. Most often this will be in terms of time constraints that the scientist may have. Although, other constraints may be used (or combined) such as cost of resources, reliability of execution or the level of trust required from the used resources and components. This stage will take the abstract workflow and resolve it into a concrete workflow in which implementations and resources will have been determined. The Optimisation stage of the life-cycle is decomposed further in Section 5.
- **Execution** of the workflow. In this stage the concrete workflow is enacted on the defined set of resources with implementations becoming active on resources at the appropriate point in time. If however the workflow does not proceed as planned then the optimisation stage can be re-entered so that the workflow may be altered in light of the changes. It is also possible to re-enter the optimisation stage in cases where parts of the original workflow couldn't be made concrete at the original pass or when new opportunities become available, such as availability of new resources. Further details of the Execution stage are given in Section 6.

Once a workflow enters the Optimisation stage an *Application Service* will be generated for it. The Application Service has two roles. The first is to provide a central point where the application scientist can find out about the progress of the application. The second function of the application service is to provide a constantly running entity within the Grid to represent the application. This can be used to prompt other services to perform activities as required and house the current state of the running application. See Section 7.

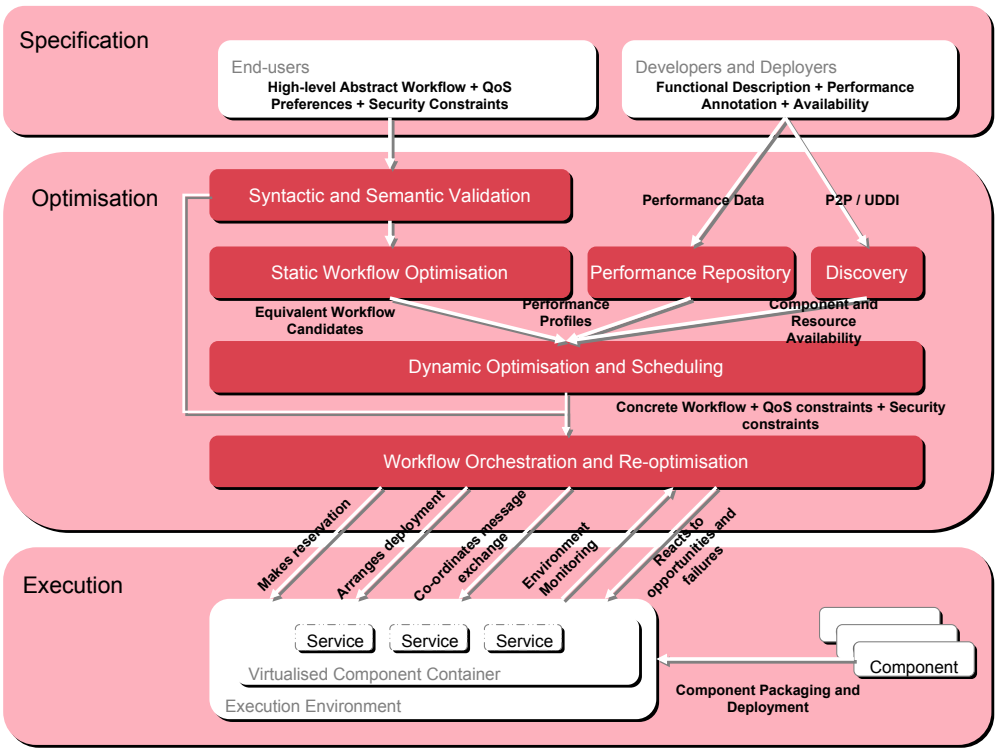

Fig. 1: The Design Diagram for ICENI II

4.1 Syntactic and Semantic validation

On entering the Optimisation stage the workflow is checked for Syntactic and Semantic validity. The first step in checking syntactic correctness ensures that all required inter-component connections have been made, otherwise the workflow will be unsuccessful at a later stage. Components have connection ports that either must be connected to operate correctly or are optional. The second step in checking syntactic correctness is to verify that the data to be exchanged through one of these connections is a valid transaction. At the most basic level this is checking such things as a component which outputs a matrix is connected to a component which is expecting a matrix. If a workflow fails at a syntactic level then it is returned to the specification stage.

The workflow may now be checked for semantic validation. At this stage scientific knowledge about what action a component performs, and what the meaning of its inputs and outputs, can be used to determine if the connection of two components makes sense. For example it may be syntactically correct to feed a matrix into a finite difference solver, though if this matrix is a diagonal solution to a set of linear equations this makes little semantic sense. If a workflow fails at a semantic level then it is returned to the specification stage.

5 Optimisation of Workflows

The Optimisation stage can be decomposed into several complementary services with most of these services being optional. The only non-optional element, referred to as the *Resolver*, matches abstract components with implementations and resources. The resolver provides simple matching of component implementations with resources, whilst pluggable optimisers can be used to improve the performance of the system. Detailed in the following subsections are the pluggable elements that we have defined so far.

5.1 Static Workflow Optimisation

The Workflow Static Optimisation Service is responsible for pruning and manipulation of the workflow in order to pre-optimise the workflow before any attempt is made to schedule. Using static information about the components, this service takes a workflow and produces a pre-optimised workflow that is expected to execute more efficiently on the Grid. This is achieved by manipulations to the workflow, including:

– **Re-ordering of components**: It may be possible to re-order some of the components within a workflow to improve efficiency.

– **Insertion of additional components**: This allows translation components to be added into the workflow to convert the output from one component into the desired format for the next component.
– **Workflow substitution**: A workflow may contain a subset of components which is semantically equivalent to an alternative subset which is known to be more efficient. This substitution can be made at this stage.
– **Pruning Redundant components**: Workflows, especially those that are composed of nested components, may contain components which serve no purpose towards the final result required by the user. These components can be identified and removed.
– **Component Substitution**: As components have multiple implementations, which may be suited better to different input types, it may be possible to use meta-data about the data that will arrive at the component to select an implementation set best suited for the particular component use.

It should be noted that the Workflow Static Optimisation Service does not consider the dynamic load on system's within the Grid.

5.2 Dynamic Optimisation and Scheduling

This section focuses on improving the efficiency of scheduling. Using complex scheduling algorithms to attempt to schedule several components over what may be millions of resources is in itself an NP-hard problem. It may be that by using simple general knowledge about the environment where the application will execute, we can prune the search space and simplify the scheduling process to a matter of seconds.

Many different techniques can be attempted at this stage. We take the approach that no dynamic resource information can be considered here. The techniques we propose for this stage are (listed in order):

– **Authorization**: If a user is not allowed to use a resource or software implementation it can quickly be removed from the potential search space.
– **Hardware / Software requirements**: Resources can be pruned from the tree if they don't match the minimum requirements. For example if the requirement is for a resource with an UltraSPARC processor and 2Gb of working memory, any resource not meeting this minimum requirement can be discarded.
– **Problem specific requirements**: In many cases it is known in advance that a particular problem has certain requirements. For example if a component is known to run for long periods of time with no ability to checkpoint then selection of unreliable resources may be pruned from the search space. Another example comes from Daily et al. [8] in which the 'closeness' of resources is taken into account when communication between components is significant.
– **Out of bounds selection**: Although a resource may match the minimum requirements for a component implementation it may be considered inappropriate for a particular use case. For example a Pentium processor running at 90Mhz may be capable of running a linear solver. However, if the problem size is significant (several thousand) it may be pruned at this stage – saving the expense of attempting to find a performance prediction for it later.

It should be noted that although we discuss at this stage the pruning of the search space this process is normally performed through the use of lazy evaluation.

5.3 Workflow-aware, performance-guided scheduling

The aim of most schedulers is to map the abstract workflow to a combination of resources and implementations that is both efficient in terms of execution time of the workflow and in terms of the time to generate the concrete workflow. Components need not all be deployed at the same time: lazy scheduling of components and the use of advanced reservations help to make more optimal use of the available resources for both the current and other users.

Schedulers need to be designed to be workflow aware. This has been implemented within ICENI [21]. Thus the scheduling of components depends not only on the performance of a component on a given resource, but also on the affect this will have on the other components in the workflow. Described below are the general steps taken to evaluate a suitable mapping of components onto resources.

As the components that make up the abstract workflow only describe the meaning of what should be carried out (we define this to include the dataflow between components) the first task of the scheduler is to match these component meanings with component implementations. There may be many component implementations matching any given component meaning. Once the implementations are known then selection can be performed as to which implementation should be used and on which resource.

The scheduler can speculatively match implementations with resources. The Scheduler can then interrogate performance information in order to obtain estimates on the execution times for these implementation / resource combinations. With this information and information gathered from the resources that have been discovered, the scheduler can determine an appropriate mapping of the components over the resources.

A number of equally optimal concrete workflows are selected using scheduling algorithms. Performance information is then utilised to predict both the duration of the entire application and the times at which each component is expected to begin execution. Performance data can be used to determine if the application will be

able to meet the *Quality of Service* (QoS) requirements set out by the user. The critical path of the application can also be determined. This will allow greater flexibility for selection of component implementations and resources for those components not on the critical path. The predicted component start times for each concrete workflow can then be passed to the reservation system, which responds with a single concrete workflow, including any reservations it was able to make.

Selection of the "best" concrete workflow – is defined by some user-defined criteria of the factors that are important to them. This could be based around quickest execution time, "cheapest" execution (where resources are priced) or some other metric, or combination of metrics. The techniques for combining these metrics and accurately modelling the users, resource owners and Grid managers requirements is an area of current research [35].

A number of scheduling algorithms have been developed for use in ICENI, these include random, best of n random, simulated annealing and game theory schedulers [35]. These schedulers can be made "workflow aware" so that they take the whole workflow into account when scheduling each component.

5.4 Lazy Scheduling / Deployment

In many workflows it may be beneficial not to map all the components to resources at the outset. This may be due to the fact that it is not possible to determine the execution time of a given component until further information is obtained by running previous components in the application. It may also be desirable to delay mapping until a later time in order to take advantage of resources and/or component implementations that may become available during the lifetime of the workflow.

The meta-data held about a component implementation indicates whether a component can benefit from lazy scheduling and / or advanced reservations. The scheduler may then decide to initially ignore these components. When the rest of the components are deployed to resources, all components that are not currently mapped to a resource (or to a future reservation on a resource) are instantiated in a virtual space (referred to as the "Green Room"). Components in the "Green Room" are able to communicate with other instantiated components, though only calls that add configuration data are valid, any call that requires computation will cause the component to be scheduled. Alternatively the application service, which is aware of the time when a component is required can pre-emptively start scheduling so that the component is made real (just) before it is required. Components which hold advanced reservations will remain in the "Green Room" until the start of their reservation slot. At this time the components will be deployed onto the resource which contains the reservation.

6 Execution of a Workflow

Due to the uncertainties of resource and network availability in a dynamic system such as the Grid, it is necessary to support advanced reservations to provide QoS guarantees. Reservations may be made on computational resources, storage resources, instruments or the underlying fabric of the Internet such as network links. The reservations may be made for exclusive use of the entity or, in some cases, some pre-agreed portionF of it.

The true vision of a computational Grid is that of a large number of resources, owned by many different organisations, available to execute code for anyone with access to the Grid. The motivating factor for many resource owners is likely to be the income that they can receive by selling cycles on their resources that would otherwise be left idle. However, it is not only resource owners that would benefit from this form of access to computational power. Small businesses and end-users could gain pay-by-use, metered access to large resources that they couldn't afford to purchase directly. Once the cost of resources is known to the consumers they can determine how they access the Grid in accordance with their willingness to pay. A trade off may be made between the speed of execution required and cost of access to a very high-performance resource.

In order for this vision to become a reality, an integrated, programmatically accessible payment framework needs to be integrated into the scheduling and execution stages of the workflow pipeline [15]. Given a mix of competing resource providers and the ability for users to negotiate requirements, possibly via a broker, for access to computational resources, a vast new market in execution power can be opened up.

The execution environment represents the virtualisation of the resource that manages the life-cycle of the parts of an application. The execution environment encapsulates the facilities available to the software component, such as inter-component communication, logging, monitoring, failure recovery, checkpointing and migration. These facilities are exposed to the software component through a set of abstract APIs. These abstractions allow the execution environments managing the parts of an application to co-operate and co-ordinate their runtime capabilities, such as network transport, co-location and shared file system. Software engineers developing the components are insulated from the implementation choice made by the optimisation stage by following the software patterns offered by the APIs. This is analogous to the MPI[12] abstraction for message-passing in parallel applications.

The software component instantiated in the execution environment is referred to as a service. We adopt Web Services as one *view* of the running software component. It is an ideal way for services on different physical resources to communicate with each other in an interoperable manner. The elements in the execution environment will be discussed in more detail.

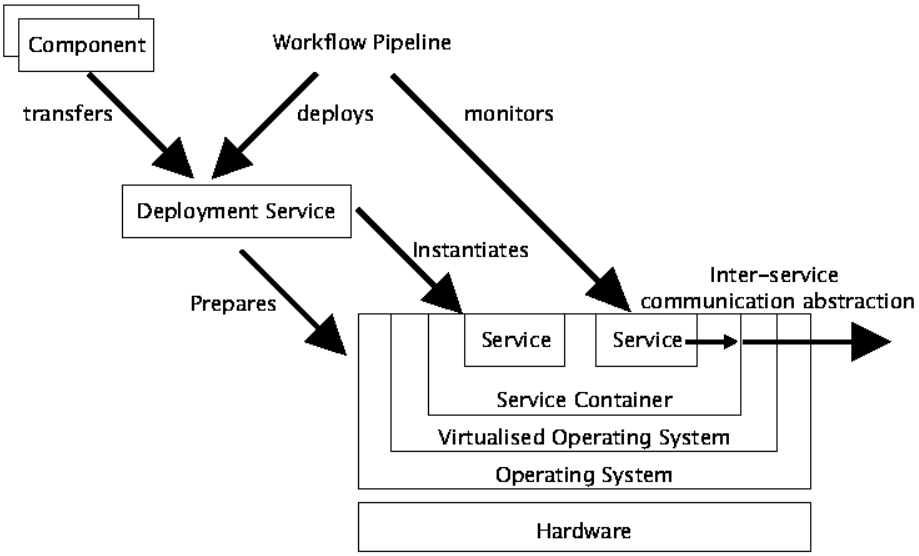

Fig. 2: Execution Environment and Multi-level Virtualisation

6.1 Component Deployment

A deployment service is the gateway to a computational resource. It is responsible for facilitating the provisioning and instantiation of a component assigned to a particular resource. Firstly, the deployment service prepares the execution environment. This might involve the preparation of a component container in a cluster resource. Recent advances in virtualisation technologies [3, 34] offer operating system-level virtualisation. Within the virtualised operating system, a component container provides the higher-level abstraction to the software component on top of the operating system facilities. The compartment model offers attractive features such as security and fault isolation. Multi-level virtualisation allows runtime facilities to be flexibly configured depending on the deployment requests [29]. Although virtualisation provides a sandbox environment for a component to execute seemingly exclusively, the cost in instantiating the container on-demand [16] may be too high for short-running components. Predictive instantiation might alleviate the setup cost by allocating resources in advance.

Once an execution environment is available, the deployment service will facilitate the provision of the software component onto the resource. This might involve the staging of software packages and their dependencies available remotely into the system. In order for this architecture to succeed across the Grid, a standardised interface for deployment and a language for software requirement description is essential. It reduces the need for users and software agents to understand a large number of description languages and deployment mechanisms to exploit a variety of Grid resources. The Job Submission Description Language (JSDL) [13] is being defined in the Global Grid Forum [9]. Although currently focused on describing deployment of traditional POSIX application, an extension has been proposed for describing software components for Java enterprise compliant containers and others. The Configuration Description, Deployment and Lifecycle Management (CDDLM) [4] is another standard effort focusing on the generic description and life-cycle control of components.

6.2 Checkpointing and Migration

Checkpointing is a technique for preserving the state of a process in order to reconstruct it at a later date. It is a crucial element for providing fault-recovery from a saved state. In scientific applications checkpointing provides a means for long-running simulations to be restarted at a previously examined parameter space [6]. This is also an important means for migrating the state of a process to another execution environment. This is often triggered by a re-scheduling decision as in some distributed resource managers such as Condor [30]. Migration might occur as a result of a recovery operation after a host failure [18]. Alternatively migration may be as the result of a desire to exploit new possibilities within the Grid, such as a new resource becoming available. In addition, migration can be initiated by a user wishing to steer an application according to performance and co-location concern typical in a simulation involving collaborative visualisation [26, 5]. Many checkpointing and migration systems exist including OpenMosix [32], OpenSSI [33] and Kerrighed [31].

In all cases, the application service prompts for a re-scheduling of the service process to a suitable resource so that QoS constraints are still respected. This might involve re-scheduling other services to different resources to achieve an optimal schedule. The stored checkpoints of services are transferred to the suitable resources through the deployment service and restarted in a reinstated execution environment. A component might receive messages during the time elapsed be-

tween its failure and restart. Such events are taken care of by the messaging abstraction in the initiating execution environment. The execution environment reports any anomalies like resource over-loading or network failure to the application service which in turn might trigger the migration process.

7 Application Service

The application service provides a real time service representing the running application. Once an application has started the application service will hold the current status of the running workflow and is able to monitor the progress of the workflow by interacting with all of the stages described above. It can use this information to determine if the application is running correctly and if it needs to trigger re-scheduling of the workflow and / or migration of some of the components. The application scientist can interact with the application service to find out the current status of the running application.

Figure 3 illustrates the stages of an application's life-cycle. The circular path represents the application service. Services can be interacted with through the blue interface lines with those services on the outside of the loop only being used once and those on the inside being used (potentially) more than once.

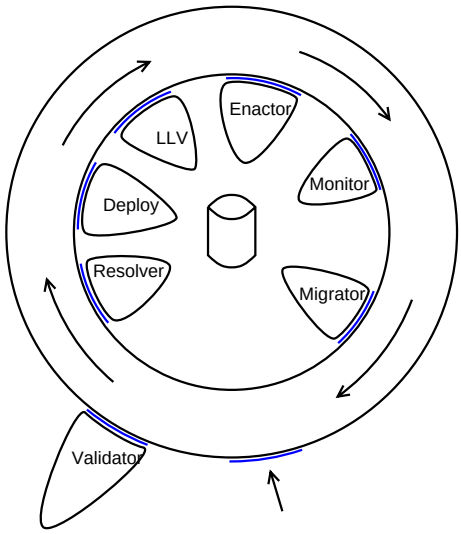

Fig. 3: The enacting of a workflow in ICENI II

Once the application has passed the validator it progresses to the resolver, which may contain optimisation stages. The appropriate components can then be deployed onto the resources and the Low Level Validator (LLV) can be used to determine if this has been performed correctly. The Enactor is used to start the components on the correct resources at the appropriate time, while the monitor evaluates whether the workflow is progressing to plan. The application service can use this information to determine if migration is required or if further interaction with the reslover/optimiser is required.

8 Application Interaction

Once execution of a workflow begins, output may be produced. This output needs to be managed according to the requirements of the application owner. It may be that the executing application simply writes out results to a file and, after execution completes, the file is returned to a location decided by the application owner.

8.1 Steering and Visualisation

The ability to steer complex computations simplifies a scientist's work by allowing the parameters of a computation to be changed during the execution process. Rather than start a long computation with a given set of parameters, then waiting till the execution completes before modifying the input parameters and running again, scientists can steer the computation by testing different input parameters in real time.

Visualisation allows scientists to view a visual representation of a computation. Techniques such as the Lattice Boltzmann method for modelling complex fluid flow problems benefit from the use of visualisation, as shown within the RealityGrid project [28]. Attempting to extract information from mathematical representations of such problems is much more difficult than seeing visual cues.

Work within RealityGrid has gone one step further than allowing scientists to steer and visualise data. Collaborative visualisation and steering is an important tool to allow scientists in different physical locations to work together efficiently. In a distributed environment such as the Grid, this is particularly useful. Components were developed within the ICENI framework to support the visulisation of a computation, concurrently, at several sites. Additionally, steering components could be executed at multiple sites to provide a way to steer the computation from multiple locations [7].

9 Developments towards ICENI II and Conclusion

Wherever possible, and appropriate, ICENI II will be developed using existing standards. To this end ICENI II will be developed as a set of Web Services and we are currently either using or investigating standard Grid Languages such as BPEL4WS and JSDL.

The first step towards ICENI II has been the development of a lightweight, Web Service based job submission and monitoring service (GridSAM [11]), based on an original prototype (WS-JDML [17]). GridSAM uses the upcoming Job Submission Description Language (JSDL [13]), evolved from JDML [1], and standardised through the Global Grid Forum [9]. This work will also comply with the up-coming OGSA-BES (Basic Execution Service [24]) and we are feeding our

work in to this effort. This work is being hardened through collaboration with the Open Middleware Infrastructure Institute [25].

We are now developing some of the higher level services on top of GridSAM for job deployment and migration. Along with developments in the other services for optimising workflows and deploying workflows across multiple resources.

References

1. A Common Job Description Markup Language written in XML. http://www.lesc.doc.ic.ac.uk/projects/jdml.pdf.
2. Anthony Mayer and Andrew Stephen McGough and Nathalie Furmento amd Jeremy Cohen and Murtaza Gulamalim and Laurie Young and Ali Afzal and Steven Newhouse and John Darlington. *Component Models and Systems for Grid Applications*, chapter ICENI: An intergrated Grid middleware to support e-Science, pages 109–124. 2004.
3. P. Barman, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP 2003*, September 2003.
4. CDDLM Working Group, GGF. https://forge.gridforum.org/projects/cddlm-wg.
5. J. Chin, P. V. Coveney, and J. Harting. The teragyroid project: Collaborative steering and visualisation in an hpc grid for modelling complex fluids. *UK All-hands e-Science Conference, 2004*, September 2004.
6. J. Chin, J. Harting, S. Jha, P.V. Coveney, A. R. Porter, and S. M. Pickles. Steering in computational science: mesoscale modelling and simulation. *Contemporary Physics*, 44:417–434, 2003.
7. J. Cohen, N. Furmento, G. Kong, A. Mayer, S. Newhouse, and J. Darlington. RealityGrid: An Integrated Approach to Middleware through ICENI. *Philisophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences (to appear)*, August 2005.
8. Holly Daily, Henri Casanovay, and Fran Berman. A Decoupled Scheduling Approach for the GrADS Program Development Environment. In *Proceedings of the Supercomputing 2002 conference*, Baltimore, November 2002.
9. Global Grid Forum. http://www.ggf.org.
10. N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. ICENI: Optimisation of Component Applications within a Grid Environment. *Journal of Parallel Computing*, 28(12):1753–1772, 2002.
11. Grid Submission and Monitoring service (GridSAM). http://www.lesc.imperial.ac.uk/gridsam.
12. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.
13. Job Submission Description Language Working Group. https://forge.gridforum.org/projects/jsdl-wg.
14. M. Y. Gulamali, A. S. McGough, R. J. Marsh, N. R. Edwards, T. M. Lenton, P. J. Valdes, S. J. Cox, S. J. Newhouse, J., and Darlington. Performance guided scheduling in genie through iceni. In *Proceedings of the UK e-Science All Hands Meeting 2004*, Nottingham, September 2004.
15. Jeremy Cohen and John Darlington and William Lee. Payment and Negotiation for the Next Generation Grid and Web. In *UK e-Science All Hands Meeting*, Nottingham, UK, sep 2005.
16. K. Keahey, K. Doering, and I. Foster. From Sandbox to Playground: Dynamic Virtual Environments in the Grid. In *5th IEEE/ACM International Workshop on Grid Computing*, November 2004.
17. W. Lee, A.S. McGough, S. Newhouse, and J. Darlington. A standards based approach to job submission through web services. In *Proceedings of the UK e-Science All Hands Meeting 2004*, Nottingham, September 2004.
18. M. Luo and C. Yang. Constructing zero-loss web services. *20th IEEE International Conference on Computer Communications*, June 2001.
19. A. Mayer, S. McGough, N. Furmento, J. Cohen, M. Gulamali, L. Young, A. Afzal, S. Newhouse, and J. Darlington. *Component Models and Systems for Grid Applications*, volume 1 of *CoreGRID series*, chapter ICENI: An Integrated Grid Middleware to Support e-Science, pages 109–124. Springer, June 2004.
20. A. Mayer, S. McGough, N. Furmento, W. Lee, S. Newhouse, and J. Darlington. ICENI Dataflow and Workflow: Composition and Scheduling in Space and Time. In *UK e-Science All Hands Meeting*, pages 627–634, Nottingham, UK, September 2003. ISBN 1-904425-11-9.
21. S. McGough, L. Young, A. Afzal, S. Newhouse, and J. Darlington. Workflow Enactment in ICENI. In *UK e-Science All Hands Meeting*, pages 894–900, Nottingham, UK, sep 2004.
22. Sun Microsystems. Jini(tm) Network Technology. http://java.sun.com/jini/.
23. A. O'Brien, S.J. Newhouse, and J. Darlington. Mapping of scientific workflow within the e-protein project to distributed resources. In *Proceedings of the UK e-Science All Hands Meeting 2004*, Nottingham, September 2004.
24. Open Grid Services Architecture - Basic Execution Service. https://forge.gridforum.org/projects/ogsa-bes-wg.
25. Open Middleware Infrastructure Institute (OMII). http://www.omii.ac.uk/.
26. S. M. Pickles, P. V. Coveney, and B. M. Boghosian. Transcontinental realitygrids for interactive collaborative exploration of parameter space (triceps). Winner of SC'03 HPC Challenge Competition (Most Innovative Data-Intensive Application), November 2003.
27. Immunology Grid Project. Immunology grid. http://www.immunologygrid.org.
28. RealityGrid Project. http://www.realitygrid.org/.
29. E. Smith and P. Anderson. Dynamic Reconfiguration for Grid Fabrics. In *5th IEEE/ACM International Workshop on Grid Computing*, November 2004.
30. Condor Team. Condor Project Homepage. http://www.cs.wisc.edu/condor.
31. The Kerrighed project. http://www.kerrighed.org/.
32. The open Mosix project. http://openmosix.sourceforge.net/.
33. The open SSI project. http://openssi.org/index.shtml.
34. User Mode Linux. http://user-mode-linux.sourceforge.net/.
35. Laurie Young. Scheduling componentised applications on a computational grid. MPhil Transfer Report, 2004.