Implementing a Secure, Service Oriented Accounting System for Computational Economies

John D. Ainsworth, Jon MacLaren, John M. Brooke
Manchester Computing, The University of Manchester
{john.ainsworth, jon.maclaren, john.brooke }@manchester.ac.uk

Abstract

The ability to record and account for the usage of computational resources in a standardised way across different systems from multiple administrative domains is a precursor to widespread Grid deployment and availability. We describe the implementation of a Resource Usage Service (RUS), which is based on two emerging Global Grid Forum (GGF) standards, and uses Web Services technology. We address the issue of security, which is essential since usage information is potentially sensitive and must be verifiably accurate. We show that useful robust functionality can be provided using current Web Services standards and we thus avoid over-dependency on evolving Grid frameworks and middleware. This work has been undertaken within the UK e-Science project Markets for Computational Services, and builds on scenarios contributed to the project by organisations that manage and sell computing resource as a business.

1. Introduction

The most popular current approach in Grid Computing is to build grids using Service Oriented Architectures, typically based on existing and emerging Web Services standards. So a Grid can be seen as a collection of interacting services some of which virtualize the underlying resources, which can then be thought of as commodities. These include, but are not limited to, compute cycles, data storage, database access, information streams from sensors, and the use of experimental apparatus. In the future, it is likely that these resources will be traded, as well as consumed.

The need for a standardised way of recording usage, which will enable the above, has been recognised by the Grid community [1]. However, this is an area that has not received the attention it warrants, despite some initial efforts [2][3][4]. Resources on the Grid tend to be allocated and charged for through traditional (non-Grid) mechanisms, with the use of resources only being permitted through the prior negotiation of access, again through traditional mechanisms. For example, although it is possible to access resources on the Grid through Grid middleware, such as the Globus Toolkit [5] or UNICORE [6] you must first apply for an account on the underlying machines through non-Grid mechanisms. Similarly, usage can only be accounted for at the local system level rather than Grid-wide, as the mechanisms available tend to be designed for large multi-user computers or clusters operating within a single administrative domain, where the logging of usage is under the control of the resource management system that controls the resources, such as Load Sharing Facility (LSF) [7] or Portable Batch System (PBS) [8]. Indeed, on large specialist supercomputers proprietary mechanisms for accounting came as part of the management software. Grid computing is generally accepted to work across multiple-domains and strives to provide abstractions that hide the differences between management systems.

In order for Grid computing to realise its goal where users (or software operating on a users behalf) can dynamically discover and negotiate the use of previously unknown resources across multiple administrative domains, abstractions that permit the trading or exchange of resources between the different providers need to be established. The Market for Computational Services (MCS) project [11], funded as part of the UK e-Science programme, aims to help bring this goal nearer, by using the model of a utilities market to provide some of the core components that are required for such an infrastructure. Only when Grid computing can provide standards-compliant services for accounting, measuring, and charging for resource usage, can the use of resources on the Grid become decoupled from traditional accounting and charging solutions.

We also believe that enabling resources to be re-sold is vital to the commercial development of the Grid. By introducing resellers into the Grid, you allow third-party companies—who might not own any Grid-enabled resources—to contribute other areas of expertise, while allowing them to make a profit. One of the use cases produced by the MCS project discussed a Computational Chemistry reseller, who targeted computational chemists, providing—for a monthly fee—some supercomputer CPU hours with a number of “free” runs of the Gaussian04 software; subsequent usage would be charged at a set tariff [12]. In short, we have to enable new business models if commercial participation in the Grid is to take off. Only then can we realise the goal of grid computing as a utility, like electricity, water or gas.

2. The Market for Computational Services Project

The Markets for Computational Services (MCS) Project started in May 2003, and initial plans for the project were based upon the emerging Open Grid Services Infrastructure (OGSI) [13]. The aim was to produce a basic infrastructure during the first year of the project, covering the recording of resource usage and interfaces to charging mechanisms; and to build upon this in the second year, developing methods for aggregating and auctioning resources, while also refining the infrastructure. Descriptions of this planned work can be found in [9][10].

The abandonment of OGSI by those proposing it caused the goals of the MCS Project to be adjusted. Much of the second year of the project had to focus on refactoring the infrastructure to fit around Web Services. This task was complicated by the fact that the proposed architecture relied heavily upon those features of OGSI that deviated from standard Web Services.

As part of the second year work of the MCS Project, a prototype solution was developed for the user-guided discovery of computational resources suitable for the execution of a specified computational job. In addition to the Resource Usage Service, described in this paper, this end-to-end solution included a Resource Broker and a Quotation Service, as shown in the architecture diagram in Figure 1. From this prototype solution, only the Resource Usage Service has been developed to a standard suitable for deployment in production services, whilst the other components remain as open areas of research. This reflects the current priorities of those deploying Grids. An overview of this work has yet to be published.

Other publications and documentation relating to the MCS project can be found on the project website [11].

3. Architecture

The GGF Resource Usage Service working group has produced a specification, which defines a Resource Usage Service (RUS) [15], whose purpose is to store accounting information. As this is only a draft, it is incomplete; some areas are underspecified, ambiguous and open to interpretation. The RUS specification itself depends upon the Usage Record format specification [14], which is another draft GGF standard, albeit one which is nearer completion. This standard provides a common way of representing usage information for batch and interactive compute jobs in XML, which is independent of the system it was generated from. This enables the recording of usage in a homogeneous way from a set of heterogeneous resources. The Usage Record format has also been used in the accounting system developed as part of GridBank [4], but this does not use the RUS specification.

The most recent version of the RUS specification is based on the Open Grid Services Infrastructure or OGSI, which is now obsolete. Consequently, we chose to implement a plain web service version of the Resource Usage Service and so the parts of the GGF specification relating to OGSI, e.g. the use of Service Data Elements, were re-interpreted within our implementation paradigm. We encountered no difficulties with this re-interpretation. The RUS is a persistent single instance service, whose job is to provide persistent storage of usage data. All the operations the RUS exposes to clients are atomic, there is no dependency between them; consequently interactions with the RUS are stateless.

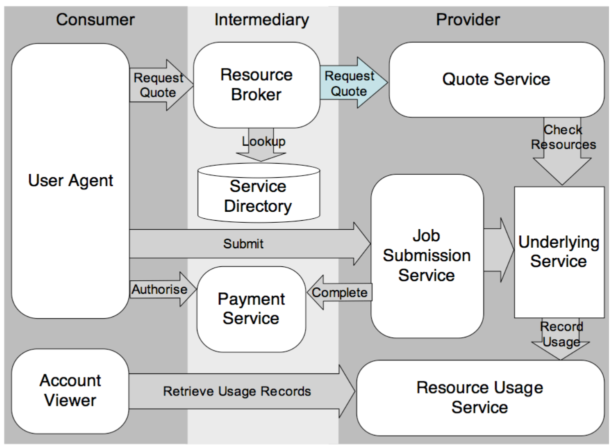

Figure 1. End-to-End Economic Architecture Implemented in the MCS Project

The three fundamental requirements, which drive the architecture of the Resource Usage Service, are as follows. Firstly, it must provide persistent storage and retrieval of XML Usage Records, conforming to the format defined by the GGF Usage Record Working Group [14]. Secondly, it must implement the WSDL port-types defined in the GGF Resource Usage Service Specification [15]. The final requirement is that read and write access to the data stored must be restricted to ensure privacy and prevent fraud.

As the RUS is a system on which financial transactions ultimately depend, security is of the utmost importance. Ensuring the privacy of data requires read access to be restricted on a per record basis. For example, a resource consumer should be able to retrieve their usage information, but no one else's. It also requires that the messages between the RUS client and the RUS are encrypted. Preventing fraud requires a combination of measures to be taken. Firstly, write access must be restricted to those entities that own the metered resources. This means that service consumers trust service providers to record the usage correctly. This is the usual trust model employed when consumption is metered. The data stored in the RUS must also be non-repudiatable. This can be achieved by ensuring message integrity between RUS client and RUS, and by maintaining an audit trail of each write operation on the RUS, where the identity of the entity performing the write operation and the date and time of the operation is recorded.

Underpinning all of this is the ability to authenticate users and authorize their individual operations. In accordance with the RUS specification we have implemented role-based access control, with two categories of users, namely administrators and resource managers. The administrators have full read and write access to the RUS. The resource managers have their read and write permission restricted to only the records that relate to the resources for which they have managerial responsibilities.

4. Implementation

Our implementation consists of a web service that implements the RUS interface, and performs access control, and an underlying database which provides the storage and retrieval functionality, as is shown in Figure 2. This is a natural and obvious implementation for the Resource Usage Service.

The Markets implementation of the RUS is in plain web services, and has been deployed in the Sun Application Server container, which is part of J2EE. It does not depend on any Grid middleware, but does need WS-Security to be supported in the web service container, which we had to implement in J2EE.

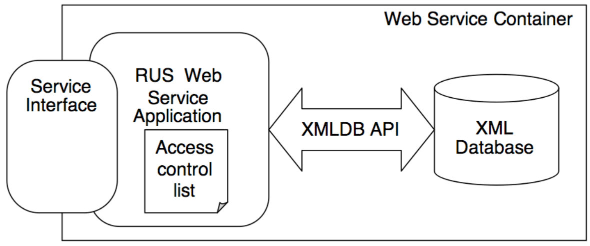

Figure 2. Resource Usage Service Architecture

We have chosen the Apache Xindice [16] XML database as our underlying database, which implements the XML:DB API [17]. An XML database was used as opposed to a conventional relational database, because we are storing usage records that are XML documents, and so we can store them directly in XML format without the need to map to a different database schema. The XML:DB API supports queries using XPath [18] and record modification using XUpdate [19]. These are both native XML technologies and eliminate the translation required with non-XML databases. The Xindice XML database is itself implemented as a web service and is hosted in the same container instance as the RUS web service application.

The Resource Usage Service Specification states that Usage Records should be stored in a single XML document whose root element is of the type `rus:RUSUsageRecords`, which contains multiple `rus:RUSUsageRecord` elements. This has not been implemented as described for the following reason. The Xindice database is optimized for storing large numbers of small to medium sized XML documents. It does not handle single large documents well, and indeed, its maximum document size is constrained to 5 MB. Therefore we have stored each `rus:RUSUsageRecord` as a separate document. This is viable with the Xindice database, as it allows XPath queries to span all documents in a collection.

The GGF RUS working group has not yet produced a WSDL description of the service interface, and so we have defined one based upon the Service Interface Definition given in the RUS specification. We have used the document-literal encoding for the SOAP messages. There are three fault types defined, namely `RUSInputFault`, `RUSProcessingFault` and `RUSUserNotAuthorisedFault`. The first is used when an operation is invoked with invalid input, for example a usage record that does not conform to the schema. The second is returned if the RUS encounters an internal error. The third is used when a user has no authorisation to access any of the operations of the RUS instance. For each operation we have tried to use the types defined in existing schemas where possible. For responses, we have defined our own types for `OperationalResult` and `RUSIdList` in accordance with the RUS specification. We have used JAX-RPC (1.1.1) [20] for compiling our WSDL to Java, with the databinding option switched on. However due to the complexity of the `urwg:UsageRecord` format, translation completely to Java classes is not possible, and so those operations which pass `urwg:UsageRecords` pass them as a `SOAPElement`. The RUS then parses the `SOAPElement` and verifies it against the schema to ensure that the usage record is valid.

The URWG schema specifies only two mandatory elements in a usage record, namely `urwg:RecordIdentity` and `urwg:Status`. Our implementation requires that usage records must also contain the elements `urwg:MachineName`, `urwg:SubmitHost`, `urwg:GlobalJobId` and `ds:X509SubjectName` from `urwg:UserIdentity`. If these elements are not present then the record is rejected. The rationale for this is that without these four fields, there would be no way to trace which user or resource the record relates to. In future the elements that are mandatory in this way will be a configurable option.

The RUS specification defines operations for inserting, modifying, replacing and deleting records. As the RUS is recording usage information that is ultimately used to determine how much a consumer owes a resource provider, it is requirement that an audit trail exists for every record in the database, as it is can be effective in resolving disputes. An audit trail records every modification to a Usage Record. The RUS specification provides for the beginning of the audit trail, by mandating in the `rus:RUSUsageRecord` format that when a record is inserted into the RUS, the distinguished name of the entity which is storing the record and a timestamp. We have extended this such that when a record is modified, we also record the identity of the modifier and the time of the modification, as part of the `rus:RUSUsageRecord`. When a record is deleted, only the `urwg:UsageRecord` element of the `rus:RUSUsageRecord` is removed, and again we record, as part of the `rus:RUSUsageRecord`, the details of who deleted the record and when they did it.

Our implementation of the RUS uses an access control list, which is an XML document stored in a configuration file, to define the roles of users and their permissions. We have defined an XML schema for the format of this access control list. Each entry in this file defines either an administrator or a resource manager, and associates it with the distinguished name from an X.509 [21] digital certificate. If a resource manager is defined, then the permitted resources are also listed. In our implementation we have used `urwg:ProjectName`, `urwg:MachineName`, `urwg:SubmitHost` as the three types of resources, although this could be made a configurable option. A resource manager may have zero or more entries for each resource type. For an individual usage record, the resource manager is authorized to read or write it if for each one of the non-empty resource types defined in the access control list there exists a entry whose value is equal to the value of the corresponding element in the usage record.

We used WS-Security [22] using X.509 digital certificates both for guaranteeing message integrity and for authentication of the entity invoking a RUS operation. This has been implemented into the JAX-RPC handler mechanism. When the web-services container invokes the application code for an operation, the application code can retrieve the verified X.509 distinguished name of the client entity and also be sure that the message has not be tampered with. The next step is to authorise the client entity. For all operations, the client entity is categorised as either an administrator or a resource manager. If the client entity has no entry for its distinguished name in the access control list then a `RUSUserNotAuthorisedFault` message is returned. If the client is determined to be an administrator then no further authorisation is required. For resource managers, per record authorisation is required and this is implemented in one of two ways depending on whether the operation is based on a query or not. For query based operations, such as `RUS::deleteRecords`, the XPath query is extended with extra predicates derived from the permitted resources list. This ensures that the query only returns records for which the resource manager is authorised. For non-query operations, such as `RUS::insertUsageRecords`, the resource manager must be authorised for each record in turn. This is implemented in the RUS by first determining the set of records affected by the operation, and then determining whether the resource manager is authorised by checking the values of the elements in the usage record against the permitted resources defined in the entry in the access control list. If a resource manager is not permitted to perform an operation on a specific record, for example using the `RUS::deleteSpecific` operation, then `rus:permissionDenied` is returned in the `rus:operationalResult` element.

Only those defined as administrators and resource managers are allowed to access the RUS. However, allowing individual users to check their personal resource usage is desirable feature. We have devised an architecture that permits this and does not require the configuration of individual users in the access control list of the RUS. This system is shown in Figure 3.

Users are able to access the query system through a web browser, using the SSL protocol for client authentication. A query form is presented to the user, which allows them to select a pre-defined query or enter their own XPath query. The form is sent to the Java Servlet and it appends an extra predicate to the XPath query, which ensures that only the records corresponding to the authenticated user's usage can be returned. The Servlet sends the query to the RUS web service, and returns the results to the users as a HTML page. The RUS Query Servlet must have permission to access the RUS as either an administrator or a resource manager. The alternative to this mutli-tiered approach would be to append the extra predicate at the RUS itself when a query operation is invoked and the requesting entity is not know to the RUS through the access control list. This approach has several disadvantages - the user must have a bespoke application capable of querying the RUS; intermediate processing of the query and responses is not possible; a denial of service attack exists as anyone can make the RUS execute a query. We have successfully used this approach to provide a web-based administration interface, which exposes all operations of the RUS.

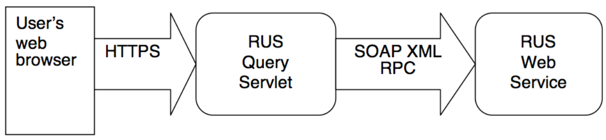

Figure 3. Multi-Tiered RUS Query Architecture

5. Performance Characterisation

The RUS is expected to be deployed on high job throughput clusters and grids. This requires that the RUS must have high storage capacity, low processing overhead for read and write operations, and is able to handle with a large number of transactions per second. The performance characterisation and subsequent tuning is necessary before deployment.

The performance of the RUS is dependent in the main on the underlying database used, which in our case is Xindice. The RUS web service and the container that it executes within incur a constant processing overhead that is independent of the size of the database. For example with an insert operation, the RUS web service checks that a Usage Record is valid and that the user has authorisation to insert this record. Then it queries the database to see if the record already exists, and if it does not then it inserts it. The time taken for the query is dependent on the size of the database, whilst the actual insert operation is not.

In our performance tests we have used a dual CPU Intel 3.06GHz box with 4Gb RAM with Red Hat Enterprise 3.0 and the Sun Application Server 8.0 Update 1 as the container. Xindice permits the user to choose the elements that will be indexed. In our tests we restricted this to the four elements that are used to determine if a record already exists when inserting, namely `urwg:MachineName`, `urwg:SubmitHost`, `urwg:GlobalJobId` and `ds:X509SubjectName` from `urwg:UserIdentity`. This ensures that the insert operation is as fast as possible. If a query is made across an element that is not indexed then Xindice must perform a sequential search through all the elements. This has implications for the RUS Query Service and deployment of the RUS, if arbitrary queries are to be permitted, then Xindice must index every element of the Usage Record, otherwise queries must be restricted to the elements that are indexed.

In our testing we have used the Usage Records produced by the CSAR HPC service. Since there are not enough unique records available to populate the database to the levels we require, then our spooler that inserts the records, replaces the `urwg:UserIdentity` with a random string value and increments the `urwg:GlobalJobId`.

We have populated a database with 1.5 million records. The sizes of the underlying database files on disk are shown in Table 1.

Number of Records	1.5 million
Average Record Size	1.2 Kb
Size of database file	9.3 Gb
Average size of each index file	1.9 Gb
Average insertion time for a record	0.3 s

Table 1 Statistics for a heavily populated database

We have tested the RUS with multiple clients operating simultaneously. We have used ten inserters, with two deletion clients and two query clients. The deletion client request the deletion of ten records chosen at random, whilst the query client requests the retrieval of ten records chosen at random. When multiple inserters only are run, we have seen a small (<0.01s) increase in the average insertion time on our 1.5 million record database, which reflects then extra processing overhead for the container handling multiple requests. Using the deletion and query clients simultaneously with the inserters predictably increases the insertion time.

6. Lessons Learned

The specification mandates that usage records be stored as a single XML document, but we chose to store them as separate documents for performance reasons. This raises the question of how much a specification should dictate an implementation. In this case, does it matter how the information is stored in the RUS, providing it conforms to the service interface? A case can be made for defining a single document format that is to be used when exporting the contents of the database, presumably so that it can be imported into another RUS implementation, but this would still not mandate the internal representation of data. This is a recurring issue with the RUS specification. Another example is the definition of the role-based based security in the specification. Standardising this does nothing to foster interoperable implementations, which is the purpose of a standard. Indeed, this is an area where implementations can differentiate themselves from one another, which is vitally important if we are ever want to be able to chose between commercial implementations. Whilst we acknowledge that it is only a draft standard, we would argue strongly that it should be restricted to the defining the service interface and the format of the `rus:UsageRecord`.

7. Conclusions and Future Work

In our implementation WS-Security has been used to provide authentication and message integrity. An alternative approach would be to use Transport Layer Security [23]. The advantage of TLS is that it is widely available, and it is mature. In addition, available TLS implementations would appear to be much faster than using Java-based WS-Security. TLS would also provide a means to encrypt the messages sent between a client and the RUS, which is not yet available in our WS-Security implementation.

We expect large volumes of usage data to be produced by future Grid services, and this data must be stored for an indefinite period. This will require the RUS to provide an archiving capability, where records over a certain age can be moved out of the RUS into archive storage. We currently use Xindice's backup functionality to keep copies of the data, but there is no mechanism for pruning old records from the database.

We are working towards deploying the RUS on the CSAR HPC facility at the University of Manchester [24]. The XML usage records are produced by the local batch scheduler on job completion and are stored in a spool directory on the local file system. Periodically, the Resource Usage Spooler, which is effectively a RUS client, collates all the usage records in the directory, which are then sent to the RUS by invoking the `RUS::insertUsageRecords` operation. The records are then deleted from the local file system. We are also working to deploy the RUS on the UK National Grid Service [25].

We have not considered how the Resource Usage Service might be employed in the envisaged Market for Computational Services. For example, when charging for the use of a resource, the RUS would be queried before a banking service was contacted to decrement a users account according to their usage. Alternatively, a billing service might aggregate a client's usage over a period of time, e.g. a calendar month, and present this to the client for payment. Indeed, there are many higher-level models that can be conceived, from the current situation up to the trading of Grid resources by brokers as in the electricity spot market. However, in all these scenarios, there is a need to access usage information in a uniform manner. We believe that the provision of this fundamental Grid component as a web service is an appropriate way forward.

This work was funded as part of the DTI/EPSRC e-Science Core Technology Programme through the Market for Computational Services project.

References

[1] Global Grid Forum Grid User Service Research Group "Grid Constitiution", draft version available online at http://forge.gridforum.org/projects/gus-rg/document/Grid_Constitution/en/1

[2] M. Koo, "IPG Distributed Accounting System", NASA Ames Research Centre, 2001, available online at http://www.nas.nasa.gov/Groups/Database/ipgacct.pdf

[3] A. Saleem, M. Krznaric, J. Cohen, S. Newhouse, and J. Darlington, "Using the VOM portal to manage policy

within Globus Toolkit, Community Authorisation Service & ICENI resources", in *UK e-Science All Hands Meeting*, p. 418--423, Nottingham, UK, Sep. 2004 ISBN 1-904425-21-6

[4] A. Barmouta and R. Buyya,, "GridBank: A Grid Accounting Services Architecture (GASA) for Distributed Systems Sharing and Integration", in *Workshop on Internet Computing and E-Commerce, Proceedings of the 17th Annual International Parallel and Distributed Processing Symposium (IPDPS 2003)*, IEEE Computer Society Press, USA, April 22-26, 2003, Nice, France.

[5] Globus Project website. http://www.globus.org/.

[6] Unicore Project website. http://www.unicore.org/.

[7] Load Sharing Facility, Platform Computing Products. http://www.platform.com/products/LSF/

[8] Portable Batch System. http://www.openpbs.org/

[9] S. Newhouse, J. Darlington, M. Asaria *et al*, "Trading Grid Services Within the UK e-Science Grid", in *Proceedings of the UK e-Science All Hands Meeting*, Nottingham, UK, Sep. 2003. ISBN 1-904425-11-9, pp.13–20.

[10] S. Newhouse, J. MacLaren, K. Keahey, "Trading Grid Services within the UK e-Science Grid", in *Grid Resource Management: State of the Art and Future Trends* (Eds. J Nabrzyski *et al*), Kluwer Publishing, Sep. 2003, ISBN 1402075758 pp. 275 – 285.

[11] Markets for Computation Service Project website. http://www.lesc.ic.ac.uk/markets/

[12] S Newhouse, J MacLaren, K Keahey, "GESA Usecases", Global Grid Forum Grid Economic Services Architecture Working Group, available online at http://www.lesc.ic.ac.uk/markets/draft-ggf-gesa-use-cases-01-7

[13] S Tuecke, K Czajkowski, I Foster *et al*, "Open Grid Services Infrastructure", Global Grid Forum Recommendation, GFD.15. Available online at http://www.ggf.org/documents/GWD-R/GFD-R.015.pdf

[14] R. Mach *et al*, "Usage Record – XML Format", Global Grid Forum Usage Record Working Group draft version 12. Available online at http://www.psc.edu/~lfm/Grid/UR-WG/URWG-Schema.12.doc

[15] S. Newhouse and J. MacLaren, "Resource Usage Service RUS" Global Grid Forum Resource Usage Service Working Group draft-ggf-rus-service-4. Available online at https://forge.gridforum.org/projects/rus-wg/document/draft-ggf-rus-service-4-public/en/1

[16] Apache Xindice XML database. http://xml.apache.org/xindice/

[17] "XML:DB API" draft specification September 2001, available online at http://xmldb-org.sourceforge.net/xapi/xapi-draft.html

[18] "XML Path Language (XPath) Version 1.0", W3C Recommendation 16 November 1999. Available online at http://www.w3.org/TR/xpath

[19] "XUpdate" working draft September 2000, available online at http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html

[20] "JSR-000101 Java API for XML-Based RPC Specification 1.1.", Available online at http://java.sun.com/xml/jaxrpc/index.jsp

[21] ITU-T Rec. X.509|ISO/IEC 9594-8, The Directory: Authentication Framework, 2000.

[22] "OASIS Web Services Security 1.0 (WS-Security 2004)", Organization for the Advancement of Structured Information Standards. Available online at http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss

[23] "The TLS Protocol Version 1.0", IETF RFC2246. Available online at http://www.ietf.org/rfc/rfc2246.txt.

[24] Computational Services for Academic Research website. http://www.csar.cfs.ac.uk

[25] The UK National Grid Service website. http://www.ngs.ac.uk