UK OGSA Evaluation Project

Report 2.0 (Final) February 2005

Evaluating OGSA across organisational boundaries

An EPSRC funded project

Document URL: http://sse.cs.ucl.ac.uk/UK-OGSA/Report2.doc

Report Editor: Paul Brebner, UCL/CSIRO[1]

Report Contributors: Paul Brebner, Jake Wu, Oliver Malham

Project Members:

Paul Brebner[2], Wolfgang Emmerich[3], Jake Wu[4], Savas Parastatidis[5], Mark Hewitt[6], Oliver Malham[7], Dave Berry[8], David McBride[9], Steven Newhouse[10]

Abstract

The goal of the UK OGSA Evaluation Project was to establish a Globus Toolkit Version 3.2 (GT3.2) test-bed across four organisations, and then conduct experiments to learn about the cross-organisational software engineering aspects of using it. Report 1.0 details our initial experiences installing, securing and testing GT3.2. This report covers the second phase activities, including the development of a GT3 benchmark, performance results, experiments and analysis related to deployment, index and data movement services, and further insights into security and debugging. We conclude that there is a need for better research and infrastructure/tool support for cross-cutting non-functional concerns across organisations (some of which are grid-specific, others are common to SOAs in general), and that there are substantial differences between legacy (e.g. GT2) and Service Oriented (E.g. OGSA/GT3) approaches to Grid infrastructure and applications which require further elicitation.

---

[1] Email: Paul.Brebner@csiro.au
[2] CSIRO ICT Centre, ANU, Canberra, Australia
[3] Computer Science Department, University College London
[4] School of Computing Science, University of Newcastle upon Tyne
[5] School of Computing Science, University of Newcastle upon Tyne
[6] School of Computing Science, University of Newcastle upon Tyne
[7] National e-Science Centre, University of Edinburgh
[8] National e-Science Centre, University of Edinburgh
[9] London e-Science Centre, Imperial College London
[10] OMII, University of Southampton

Contents

Note

This report is the second and final report for the project, and is designed to be read in conjunction with Report 1.0 [13].

1 Project Plan Review

The UK-OGSA evaluation project [1] started on 15 December 2004, and ran for 12 months. Four organisations (Department of Computer Science - UCL, School of Computer Science - University of Newcastle, National e-Science Centre - Edinburgh University, London e-Science Centre - Imperial College), and nine project members were involved, of which four were employed on the project (equivalent to 2.5 FTEs). A system administrator at UCL was involved part-time for the first few months for the initial installation of GT3, and a systems administrator at Newcastle was also associated with the project. A project web site was maintained during the project [1], and project issues, records and outputs are hosted in a NeSCForge site [2]. The project was managed from UCL, and regular meetings via Access Grid were conducted.

A minor hiccup occurred early in the project when Globus announced that the Open Grid Services Infrastructure (OGSI, [15]) was "dead" and that they would terminate development of GT3, and start work on GT4, which would instead conform to WS-RF [3]. We produced a position report explaining our reaction to this event, and the expected impact/changes to the project goal and plans, which was sent to Tony Hey [4].

In practice, no significant change in the project plan/goals was required, as we were interested in GT3 as an exemplar of a Grid Service Oriented Architecture (SOA) – the Open Grid Services Architecture (OGSA) [39] - a role which it was still able to fill even though it would not be developed further, documentation/support were incomplete and sporadic, and the OGSA specification itself was incomplete and underspecified.

The novel focus of this project was "the evaluation of OGSA across organisational boundaries". We desired to gain insight into the issues related to deploying OGSA across organisational boundaries, from software engineering, middleware, and architectural perspectives.

To summarise the project goals:

- Cross-organisational
- Grid SOA (i.e. not GT3 specific), and
- Evaluated from s/w engineering, middleware and architecture perspectives (not end-user science oriented).

The plan for stage one involved the installation, configuration, testing, securing, and re-testing of basic (core) GT3.2 infrastructure across the 4 test-bed sites. The results for the first phase were reported in Report 1 [13].

The plan for the second stage included:

- The development of a synthetic benchmark for evaluating the reliability, performance and scalability of GT3, and for comparing different architectural and implementation approaches.
- Installation, configuration and testing of extra services such as index and data movement services.
- Experiments with automated deployment of infrastructure and services on the test-bed.

A review of these activities follows.

1.1. GTMark – A GT3.2 Benchmark

In order to evaluate GT3 it was apparent that a realistic (in terms of computational demands and usage) test application was needed. Given the absence of any existing GT3.2 specific applications, and the effort required to port an existing scientific application, or older applications/benchmarks, it made sense to develop our own benchmark: GTMark (a pure Java Globus Toolkit Benchmark).

Performance, scalability and reliability experiments for one version of the benchmark were conducted across the four sites, and the results are reported in section 3.

We also intended to use the benchmark to evaluate different architectural and design choices, particularly the differences between exposing science as "1st order" services (which can participate in all phases of a SOA model including registration, discovery, and use), and "2nd order" services (which can only be accessed indirectly via "generic" 1st order grid specific services such as the Master Managed Job Factory Service (MMJFS

[22]). Figure 1 pictures the difference between the "science as 1st or 2nd order services" approaches (Green triangles represent external service interfaces, the large grey circle is the container which hosts the implementation (white circles) of some of the services):

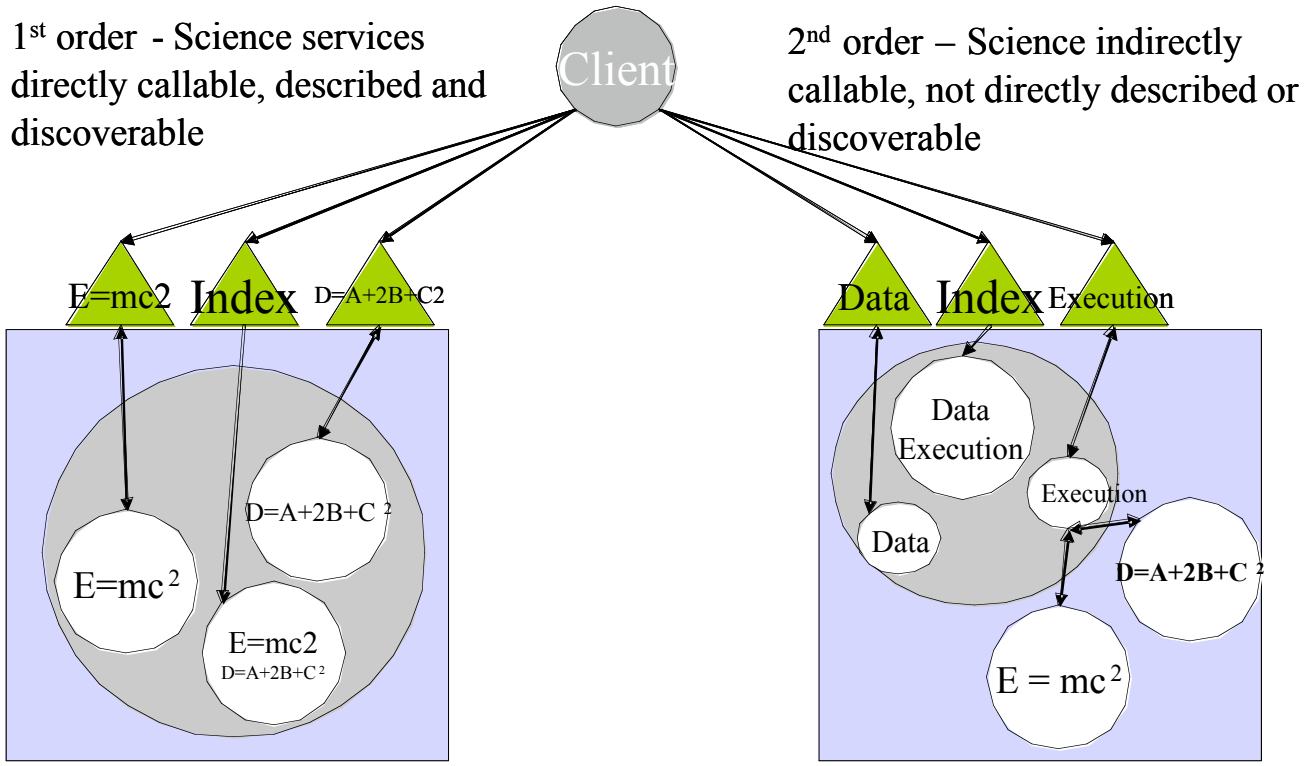

Figure 1 - Science as 1st or 2nd order services

We also wanted to evaluate various non-functional characteristics against each of these approaches (e.g. deployment). However, we determined that the use of the benchmark introduced some artificial artefacts which would inevitably skew the results. For example, because the benchmark is pure Java code, scalability and response times of the 2nd order implementation is bound to be poor due to having to start a JVM for each service invocation, compared with 1st order approach which enables multiple service instances to be hosted in the one JVM. Running a few JVMs should not be an intrinsic problem, but the total resource usage (such as memory) will be higher, and the start-up time for each new JVM will be slower (the server JVM engine is not optimised for start-up time, but is needed to ensure adequate performance). However, running a larger number of JVMs would be substantially more un-scalable [E.g. 40].

Also, due to bugs in various parts of the infrastructure some features that were critical for the comparison of approaches were known to be defective. For example, bugs in both Axis and GT3 meant that SOAP attachments do not work correctly. Finally, the implementation of the two versions of the benchmark was not completed in the time-frame and with the resources available. The intrinsic difficulties of coordinating and managing distributed activities, such as development and deployment, across remote

project member sites, contributed to this. Section 2 contains more details of the benchmark.

1.2 Testing other services

Good progress was made testing index services and data transfer. However, in the long run these proved to be problematic to set up in the time-frame available, and an exhaustive evaluation of these services using the benchmark to compare their utility for each of the architectural choices was not conducted. Instead, an analytic evaluation based on our experiences is reported in sections 5 and 6.

1.3 Deployment Experiments

Specification, development, deployment and evaluation of a scalable automated approach for deploying GT3 infrastructure and services was planned. A UCL MSc student project resulted in the development and demonstration of an automated framework for GT3.2, Tomcat, and Grid services deployment, in a laboratory setting. However, transferring the solution out of the laboratory to cross-organisational deployment with firewalls, security, and deploying secured infrastructure and services, encountered a number of problems which could not be resolved in the time-frame of the project. More details can be found in section 4.

1.4 Other activities

Although not explicitly planned, we also expect to publish papers comparing the use of 1st and 2nd order services in GT3 (analytically, using a "mechanisms" or "quality attribute design primitives" approach, for example see [11]), and an investigation into debugging service oriented architectures (including a novel approach resulting from combining deployment context and infrastructure with exception detection, diagnosis and rectification – see [12]). Sections 7 and 8 introduce some of these areas, including further thoughts on security.

2 GTMark – GT3.2 benchmark details

Discussions were held with NASA about using their GT2 benchmark, but the effort to port it to GT3.2 and support it exceeded our resources [5]. SciMark2.0 [6] was chosen as the server side load for a new GT3.2 benchmark, as it could be parameterised to run different algorithms, and use more or less CPU cycles and memory. It was modified to provide a suitable load for measuring throughput, and implemented as a "1st order" GT3.2 Grid Service – that is, a callable service which has a unique WSDL/GWSDL description. The client side work load and metrics gathering capability was based on an

Java 2 Platform Enterprise Edition (J2EE) benchmark, Stock-Online [7], which was extended to model the execution of concurrent work-flow instances utilising different tasks.

The benchmark is customisable at run time in a number of different ways. The nodes that the benchmark service is deployed on to can be enumerated, or discovered via a directory/index service. Tests can be run for a fixed period of time, or until a steady state for a minimum period of time has been achieved for a specified metric, and can be run with a given concurrency (equivalent to the number of concurrent work-flow instances), or a minimum, maximum and increment to repeat the test for different concurrencies. It can be configured to measure task throughput, or work-flow response times/throughput (where a complete sequence of work flow tasks must be completed). Fixed, or variable task loads can also be chosen to measure the impact on load balancing of heterogeneous tasks and servers, and the number of tasks per work-flow instance specified. For simplicity (and because we wanted to measure scalability in the simplest possible way) the current version assumes a strict pipeline work-flow, but more complex flows are possible within the same framework (e.g. allowing synchronisation points, and with variable topologies and fan-outs/ins etc).

A number of different load-balancing strategies can be selected, but the basic one is round-robin, which needs to be initialised with the relative weights for the capacity of each server (the relative number of service instances per server is proportional to the server weight). Approaches to provide dynamic load-balancing and server resource monitoring options were also investigated. Different data movement (e.g. file size, frequency and source/targets) and service state behaviour scenarios can be selected, including: stateless service invocation (data must be sent to and from the client or file server machine to the server where each service instance will invoked each time a new service is called); stateful (data only needs to be moved from one server to another when a new server is used for the next service instance invocation, and for first and last service instance). Fine grain service instance lifetime management is also possible allowing the benchmark to be used to test scalability for different service usage/lifetime and resource usage scenarios (e.g. when instances are created/destroyed, reuse of stateful service instances in same work-flow, etc).

Alternative security requirements and models can be set, and the way that exceptions (due to security violations or other causes) are handled can be varied. In the current version there is limited monitoring of the state/progress of the services/tasks (polling only at present, polling interval can be set). Metrics gathered during benchmark runs include average/min/max per call and per work-flow response times, average/min/max throughput (calls per unit time), reliability (number of service calls failures and exceptions), and time take for instance creation, work flow execution, and instance destruction.

Further work would be required to the benchmark to ensure that it the benchmark is robust, all options (and combinations of options) adequately tested, and documented

sufficiently to enable it to be used outside the test-bed context, and to extend it to target GT4 or other Grid/Web Service infrastructures.

Some other Grid or OGSA benchmarking efforts that we are aware of include [8-10].

3 Performance, scalability, and reliability

3.1 Performance and Scalability

Test background

Since publication of Report 1, we replaced the "test" container on all sites with Tomcat, and conducted performance tests. Some of these results have been reported in [14]. The installation of Tomcat was relatively straightforward, except at one site where a number of attempts were made before it was reliable enough for repeated load testing.

A basic version of the benchmark was used, with computational load only (no data transfer apart from parameter values used for each call, and the results sent back to the client amounting to a few bytes of data only). The tests were carried out across the four test-bed machines, all with the Tomcat container and the JVM server engine with _ GB Ram available for the heap. A fifth machine was used to run the client program. The benchmark service was a "1st order" service, and was manually deployed to each machine. Each service call had exactly the same computational load, and each client thread made multiple calls to simulate the execution of different work-flow instances.

Load-balancing was handled on the client-side using a round-robin algorithm based on a weighted measured throughput for each server, by initially creating the weighted number of factories per server and using a different one for each new call (otherwise it is likely that a given work-flow/thread will be constantly bound to service instances on the slowest machine which substantially reduces the throughput).

In practice (with data movement, and stateful services) it is desirable to ensure that invocations use the same server ,and rely on service state or local data caching/movement.

A round-robin static load balancer is relatively primitive, and could be improved on with a dynamic load-balancing algorithm, based on measured response time for each factory (or node) as, in practice, calls may take different times resulting in some machines being more heavily loaded than others, exacerbated by other loads running on the same machines, and differences in speed and capacity of machines, saturation of subnets, etc. However, more thought needed to be given to the problem of dynamically and accurately measuring current load, and predicting future capacity.

Moreover, load balancing should really be done server side, for each "cluster" of machines (although in our case we did not have clusters available), and providing a single entry point to each cluster. This still leaves the problem of how to correctly load-balance across multiple "clusters" (i.e. global load-balancing). There does not seem to be a default 3rd party GT3 load-balancer. Condor does resource and job scheduling which is a different problem (and reveals something of the difference between Grid and SOA approaches: Grid is application/job oriented, and SOA is service oriented). There are also problems trying to schedule/balance jobs of different types on the same machines (e.g. long running batch jobs, and short running services).

Load-balancing and retry/redirection for failures is also an issue. The client should ideally not have to attempt to interpret and handle errors, or load balancing strategies, related to resource saturation or the unavailability of containers/machines.

Test results

During benchmark runs we measured throughput (scalability), performance (average response time for service calls), and reliability (number and type of exceptions).

The impact of security was minimal (unmeasurable) for the two major approaches provided by GT3.2 (encryption and signing). This is somewhat different to what we noticed initially using the test containers on all sites. It is likely that the test-container imposes a higher overhead for the security mechanism.

The fastest response time was on the test-bed machine at Edinburgh (3.6s) and the slowest was UCL/Newcastle (26s). As server loads, and therefore response times, increased, client-side timeouts occurred, causing problems as the default client side exception handling naively retried the call – depending on what the problem is this isn't always the best thing to do with stateful services. In fact, calling the same instance a second time was found to kill the container. It immediately started consuming 100% of the CPU and the service instance could not be deleted. The only cure was to restart the container.

A maximum of 95% of predicted maximum throughput was achieved on four machines. This, along with the slope of the throughput graphs, indicates good scalability. The reason that 100% capacity wasn't reached could be due to imperfect load-balancing.

In the original experiments we achieved comparable throughput by excluding the slowest machine. Basically the slowest machines do little to increase the throughput, and in fact are likely to impose a penalty due to the substantially longer per task response times they introduce. We did not try removing them for the final tests however.

In 3 out of 4 cases, the Tomcat and Test containers gave similar performance. However, on the slowest node (Newcastle), Tomcat was 67% faster than the test container. This was a peculiar result and even after spending considerable time reinstalling parts of the infrastructure, and examining and comparing configurations, JVM settings, etc, no

satisfactory explanation was obtained (although one suspicion is that the test container experiences a bottleneck due to shortage of a critical resource on the relatively powerless machine).

On long runs consistently reliable throughput was achieved over a period of hours. For example approx 4,080 calls/hour over 12 hours, delivering a total of 48,000 calls (or 98,000 calls/day).

By comparison, there was minimal (unmeasurable) overhead of using the GT3/Java version of the benchmark compared to a comparable standalone pure-Java version of the benchmark using local calls only.

One of the machines was a two CPU, hyperthread machine, and enabled the benchmark to perform close to the capacity of a four CPU machine, although the scalability was not perfect (both in terms of increasing load, and in the saturated part of the curve). One of the other machines also had hyperthreading, but this had not been enabled due to the inability of the Grid job scheduler associated with another installation of Globus (used by another project on the same machine) to utilise hyperthreads.

The following graphs show average response time (ART) in seconds (s) for increasing load (number of client side threads, equivalent to number of server side service instances) (Figure 2), and Throughput (Calls Per Minute – CPM) for increasing load (Figure 3).

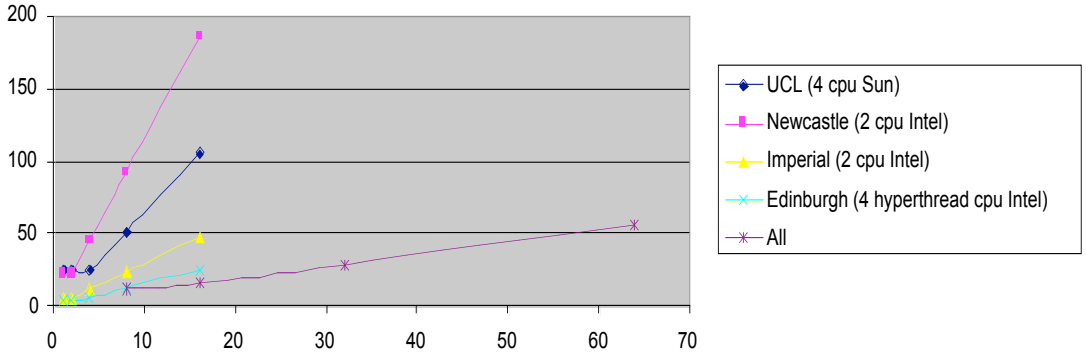

Figure 2 Average Response Time (ART) with increasing load

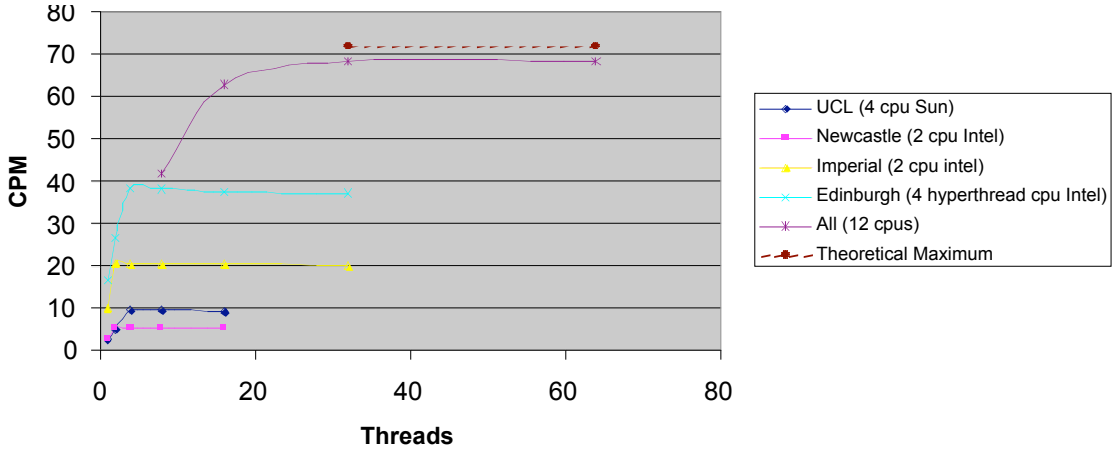

Figure 3 Throughput (CPM) with increasing load

Instance creation overhead

We also observed that the time taken to create instances using a factory service, and the first call to a new service instance, is slow by comparison to subsequent calls to the same instance (between 1 and 10 seconds slower). This is because there is substantial overhead associated with service creation (e.g. allocating resources), obtaining a grid service handle (GSH in OGSI terminology [15]), and resolving it to a Grid Service Reference (GSR). The support for stateful service instances in GT3 provides an opportunity to minimise service invocation response times compared with a purely stateless service approach, and therefore shares features of Object technologies such as Enterprise Java Beans. Although, it is likely that the EJB specification and implementations have better support for container management of statefulness, and for handling data consistency and concurrent access to "identical" instances using techniques such as optimistic concurrency or locking.

Scalability of service creation and destruction

In some tests we noticed that instance creation and destruction was not as scalable as expected. This is possibly because instance creation/destruction is being serialised at some point. In some tests exceptions were also noted, associated with instance destruction. This issue should be investigated further, as it could cause a problem in highly concurrent applications attempting to initially create a large numbers of service instances, and during the finalisation/cleanup phase when they all need to be destroyed cleanly.

3.2 Reliability and Availability

During the first round of preliminary tests using the Test container on all nodes, some runs repeatedly experienced exceptions at a rate of about 1 in 300 calls (a reliability of

0.9967%). However, we could not repeat these results with Tomcat installed, and believe that the test container was unable to service the incoming HTTP requests adequately. While we believe that GT3.2 with Tomcat now provides high reliability for at least our standard test, we were made aware of multiple possible causes of container failure, and reminded of the problems associated with handling remote exceptions.

For example, it is possible for clients to kill the container, as the container relies on clients being well behaved. A single container may be workable for single users, but with multiple users and applications hosted in the same container the chance of something going wrong increases dramatically.

Containers can be killed by invoking the same method on an instance more than once, for example, after an exception. We assume this is due to a problem with the thread-safeness of stateful instances. Part of the solution is knowing where/when a fault occurred, and what the correct response is. In this case the best approach (assuming no critical state is retained by services across invocations) is to try and kill the instance, and then create another and call it again. If service state is intended to persist across invocations, and if the state has been previously persisted in a database, then it may be possible to kill the instance, and the instance can be recreated and the state re-hydrated.

Containers can also be killed by consuming and saturating their resources. For example, by creating thousands of instances, and invoking them with high concurrency (or even just a few and calling them sequentially if they use sufficient memory). This is relatively easy to do, as numerous objects can be created in a container successfully, but then cause failures once methods on them are invoked due to lazy memory allocation. Eventually the container will saturate (cpu usage), or the memory capacity will be exceeded causing numerous exceptions when services are used or new services created (if you're lucky), or complete container failure (if you're unlucky). Similar problems have also been reported in [37].

Isolation of users, applications, and VOs through the use of more sophisticated security models (e.g. allowing role delegation), and sand-boxing of run-time environments, are likely to be important requirements in production grids. One solution is to isolate users and applications in different containers. This will have scalability limitations eventually, but is a well known enterprise solution (e.g. Borland and Fujitsu J2EE containers can be replicated on the same machine to provide customisable run-time environments, and increase reliability and scalability).

Another approach is to enable containers to manage their own run-time resource usage, by service instance activation/passivation, so that they do not run out of resources catastrophically. For example, J2EE Entity Beans can be transparently (from the client perspective) passivated/activated at any time, even *during* a transaction. This mechanism could be used to allow containers to provide a fine-grain (per instance) control over resources, rather than relying entirely on the good behaviour of clients, or external job/resource schedulers, thereby combining policy and finer grained resource handling at the container level. For example, long running jobs could be passivated allowing short-

term jobs to use container resources for a short period of time, and then reactivating the previously passivated jobs. Currently some grid job schedulers exclude short-term jobs from gaining resources if long-term jobs are already running (which can cause serious QoS isues, and can even be used to "cheat" to gain resources in advance of when they are actually needed.

Security and availability are often mutually exclusive. For example, due to problems with the host certificate expiration, notification, and renewal process, and the inability of the process to allow an overlap in the lifetime of certificates, one of the test-bed host machines was unavailable for approximately a week during a critical phase of the test activities. More attention needs to be paid to the management of certificates, in particular the process and infrastructure support for managing the whole lifecycle of certificates.

4 Deployment

During the project we elicited deployment requirements and investigated the use of one technology for an automated approach to the deployment of Grid infrastructure and services across organisations. A short paper on "Grid Deployment Scope and Requirements" was written ([16]), and used as an input to an MSc project at UCL to develop a trial solution in a laboratory setting.

We planned to investigate automated Grid deployment in the following phases:

- Within the lab
- Across sites
- Across firewalls (using port 80 only)
- Securely (secure deployment)
- Secured infrastructure (deploying secure infrastructure)
- Grid services (deploying services, and securing them)

The results of these investigations follow.

4.1 Within the laboratory

A UCL MSc student project ran concurrently with the second phase of the OGSA evaluation project, to investigate the use of SmartFrog (from HP, [17]) to deploy GT3.2 in a laboratory setting. This involved configuring SmartFrog to install and run GT3, Tomcat, and test grid service across multiple machines in the laboratory, and providing a management console to drive the process. The evaluation project acted as an end-user client for determining requirements, scope, and testing. The solution worked well in the laboratory, but relied on the freedom to install and run a completely new installation of GT3 and associated supporting software as an unprivileged user on a public file system. It was also constrained to the deployment of core/container infrastructure only, over a LAN, with no security (either for deployment, or for the GT3 infrastructure). A GUI was provided for selecting target machines (based on measured available resources, although

this was limited to JVM resources), and installing, configuring, starting and stopping either all the infrastructure, or just services. The progress of the installation (along with any exceptions) can be monitored, and a partial (just services) or complete (infrastructure) uninstall also carried out automatically. The deployment process is scalable and takes no longer for multiple machines compared to one machine.

4.2 Across sites

Upon completion of the MSc project we took delivery of the documentation and software and attempted to use it to deploy GT3.2 across the OGSA test-bed sites. However, a number of problems were encountered (some expected, others not) including: inability to get the project's version of SmartFrog deployment working correctly; different security policies at sites – some sites where prepared to open the SmartFrog daemon port with no security, others required security to be enabled, and others may not have been prepared to open an extra port even with security enabled (given the nature of the SmartFrog daemon as an automated deployment infrastructure which can be used to remotely install and run arbitrary code – potentially a perfect virus propagation mechanism).

We were able to use standard (unmodified) SmartFrog deployment across an unsecured port in a test situation. But we were unable to get the project's version of SmartFrog, their Grid specific deployment files, and their GUI management console all working together correctly. There were deployment and configuration issues with SmartFrog and the project software itself which could not be resolved in the time available, even given the assistance of some of the students and the SmartFrog support group.

Installing the base level, pure Java container (core) part of GT3 is considerably easier than the complete stack of GT3 packages (as we documented in Report 1.0, [13]), as some of them are platform specific, and require security to be enabled before they can be used. Given the reported experiences of cross-platform installation of GT3, automating remote deployment for the entire GT3 stack will be non-trivial.

4.3 Securely – secure deployment

The second problem was to get SmartFrog working with security. This was not possible in the available time frame due to issues related to the configuration, use, and debugging of SmartFrog security, and the SmartFrog support group was not able to resolve the security issues in time. This illustrates one of the problems of working with open source software with limited support and documentation, and the difficulties debugging both security infrastructure, and secured infrastructure. Basically as soon as you turn security on you loose the ability easily diagnose infrastructure problems when they occur. How you debug security infrastructure itself is also perplexing.

SmartFrog and Globus use different security models and certificates. In order to deploy infrastructure securely with SmartFrog an independent, and largely redundant, security

infrastructure, process, and certificates is required, which introduces yet another layer of complexity into already complex infrastructure and security environments. Nevertheless, the SmartFrog security architecture is relatively sophisticated, and includes code signing and multiple trust domains, and is probably well designed for the deployment domain.

4.4 Across firewalls (using port 80 only)

In theory RMI over HTTP (tunnelling [41]) could be used over port 80 to get through fire-walls without having to open extra ports. This may have been harder than it looks to set-up, and would still have required getting SmartFrog security to work to satisfy local security policies.

4.5 To deploy secured infrastructure

The next challenge was how to use SmartFrog to install, configure, and run a *secure* container. In theory all the static security configuration files and host certificates etc can be prepared and then installed using SmartFrog, although there may be some host specific information that needs to be changed. The obvious problem starts when any of the security infrastructure needs to be run as root. The "–noroot" option is an alternative for configuring GT3 security without root permission, but we are uncertain how well this works, or if it is appropriate for production environments [18]. A related issue is whether the GT3 infrastructure can be correctly installed and run by the SmartFrog daemon user. It seems probable that the Globus user will be required to install and run it.

Another issue is creating the grid-map files (host and service specific) which map certificates to local user accounts. This requires access to the user certificates for each node, and knowledge of the local user accounts that they are mapped to. It is possible in theory to use a generic single user to run all the jobs for a node, and it may even be the case that for non-mmjfs services a real user account is not needed at all [19]. However, there are significant issues to do with trust, security and auditing if the binding between users and accounts is weakened, although some sort of role-based security mechanism is inevitable [E.g. 42].

One alternative is to require the first installation and configuration to be done locally and manually, which would then enable subsequent updates or (re-)installations for different user communities to proceed automatically/remotely, as these can reuse the existing security configurations (i.e. the globally available certificates). This is what we found with multiple installations on the same machine – as long as one was installed and secured already, other versions could be installed as non-root users (although we didn't test this with more than GT3 core packages) as the security infrastructure was in place and capable of being shared across multiple installations.

4.6 Deploying services

A related problem is that of deploying *services* to an already deployed infrastructure. One solution is for the infrastructure itself to provide a service for deploying services. In GT3 this would most logically be a container based "deployment service" for deploying services into the container, and would need to use "hot" deployment so that the container does not need to be restarted each time. The only capability GT3 currently has is to use MMJFS, but it can't deploy "1st order" services, just "executables".

SmartFrog in the laboratory demonstrated the ability to deploy services to a container, and then start the container. However, this assumes that the container is not being shared, and there is only one user (community) per container. The problem with a shared container is that there is no "hot" deployment of services in GT3 – a random user can't just shut the container down whenever someone else is using it (because there is no remote facility to do this, and it would interfere with other users who have services running in the container) – unless there is one container per user (community) or application (which is possible by running multiple containers per server). The extended problem is deploying *secured* services. Service-specific grid-map files are used, which currently requires knowledge of user certificates and local user accounts.

In the above discussion the assumption has been made that each user can remotely deploy infrastructure and services. In practice this is extremely unlikely, and a confusion of roles. More feasible roles could be: Grid infrastructure deployers, who would deploy infrastructure to a set of resources remotely; and service deployers (for applications or user-community), although they would need to be able to restart containers. This wouldn't be such an issue if services were persistent across container restarts (which is possible in theory with container hosted services as long as the required per-service persistence code has been implemented, but is harder to support with legacy code). How a deployment infrastructure such as SmartFrog can be configured to allow multiple deployment roles (with different capabilities) to share the same deployment infrastructure would need to be considered further. Alternatively the sand-boxing approach of having one container per application or user-community is worth investigating.

We also consider the requirement for a single developer (or user, as in many Grid scenarios scientists both develop and use code) to install/update services across some or all of the grid sites. This is likely as developers are typically in charge of their own code and need to deploy, test, distribute, and update it. This could be done with a configuration management approval mechanism (via the application or user community deployers), or by further delegation of the deployment roles to the user and developer communities. There would need to be mechanisms to prevent conflict in the case of deployment of the same service at the same time, and to ensure service versioning (i.e. that clients use the correct version of services, with backwards compatibility maintained if necessary), and probably audit trails.

In a SOA there must be some way for potential consumers to discover deployed services. This is normally done by registering new services with a publicly readable registry/directory service. This can be done automatically (i.e. everything deployed in a container is automatically registered in the registry service of that container, and all

registry services that are subscribed to that container), or manually as an extra step in the deployment process. It is important to note that, because service registration is an aspect of the deployment role, both deployment and registration need to be adequately treated as activities in typical Grid SOA deployment roles.

4.7 Differences in deployment philosophy, requirements, roles, and practice between 1st and 2nd order Grid approaches

In the standard 1st order SOA world, deployment of new services is done within an enterprise, behind firewalls, by enterprise developers and deployers (E.g. In the J2EE specification deployment is an explicit role, with good product support). Deployed components are exposed as services for intra- and inter- organisational interoperability. Users typically don't (and can't) develop or deploy code. However, in the grid community these roles need to be supported across firewalls and enterprise/organisational boundaries (i.e. inter-enterprise), and for multiple different types of deployers, some of whom are essentially end-users.

The typical 2nd order approach for Grid services (where science isn't exposed as a 1st order service, but only invokable via a 1st order execution service) does support a type of end-user deployment, whereby executables and data are moved onto target execution machines in preparation for execution. This is typically called "staging". However, there is no way to deploy real 1st order services using this mechanism, and no support for explicit description and registration/discovery of the science code itself (requiring the applications or clients to have explicit built-in knowledge of the correct code to use, and tight compile-time coupling). We also note (see section 6 on data movement) that the mechanism used to invoke MMJFS, and the data movement mechanism, are entirely different, even to the point of using different security approaches. Intuitively we assume that there is substantial overhead in the approach to deployment used by MMJFS, depending on the size of the executables, network latency and bandwidth, and scalability of the originating file server. However, one paper reports that there may not be much impact on throughput [22] even if everything is copied every time. One difference is likely to be response time for short jobs. The 1st order approach allows services to be in place ready to run with minimal setup overhead per service call. The 2nd order approach will inevitably incur a much higher overhead for short jobs, as staging followed by job forking will increase response times.

There is a difference in security between the 1st order and MMJFS approaches. The 1st order approach allows services in the same container to have different security settings, thereby allowing different users to access them. Using the 2nd order approach to execution of code, such as MMJFS, only one set of security policies can apply to the execution of all "jobs" run in the container. This difference in the security models is likely to have an impact on the deployment, use, and management options.

For more information see an article on data services in Grid [21], and a paper measuring the impact of staging [22].

5 Index Service

We intended to evaluate Index, Data Movement and Deployment services in the context of the 1st and 2nd order architectural choices. However, as this approach wasn't pursued as extensively as planned, a stand-alone analysis will be used instead (based partially on [23]). In reality, given the nature of the benchmark, and the limited number of nodes and configuration choices available, it would have been problematic to conduct more extensive but realistic scalability tests of the registry service.

The Index Service (MDS3 - the Monitoring and Discovery System) is an information service that uses an extensible framework for managing static and dynamic data for Grids built using the Globus Toolkit 3.2. This framework provides the following functionality:

- Creates and manages dynamic Service Data via Service Data Provider programs
- Aggregates Service Data from multiple Grid service instances
- Registers Grid service instances using the ServiceGroup port type

Index Services provide the functionality within which Service Data Elements (SDEs) can be collected, aggregated, and queried. Each *Grid service instance* has a set of *service data* associated with it, and this data is represented in a standardized way as *Service Data Elements (SDEs)*.

MDS3 is not exactly equivalent to a Web Services directory service such as UDDI, although the list of features supported by UDDI 3.0 is growing [38]. There are many possible roles associated with MDS3, some of which (as we have noted above) overlap with other roles: Deployment, configuration, replication and topology of multiple index services, lifecycle management, registration/de-registration of services, security, etc. A more rigorous analysis of the relationship between these roles, features of MDS3, and the support provided by the Grid (and other standards or industry based) infrastructures would be informative.

One of the differences between UDDI and MDS3 is the ability of MDS3 to obtain, cache and display data associated with each service instance (essentially any public state of the instance) in SDEs. A related concept is service data aggregation [24]. The index service caches SDE from subscribed services until updated (this is also known as "notification caching"), and enables SDEs from multiple *sources* to be aggregrated into single *sink* services [25].

MDS3 supports both push and pull models for information update and query. By using the ServiceDataAggregator class in the service code, service data from both locally executing information providers and other OGSA service instances can be aggregated into any given service. By using Registry and ServiceDataAggregator components, multiple MDS3 services can be interconnected together into arbitrarily complex topologies to form both Virtual Organizations and Virtual ServiceDataSets. Services can

contain a union of all the service data defined in the portTypes they implement, and service data aggregation is related to inheritance.

The advantages claimed for this programming model are that it:

- Provides an aggregated view of the service state rather than individual state values or properties
- Gives a document (XML documents) centric view of the data, thereby avoiding specific methods for state access
- Is flexible for supporting dynamic state information and service state introspection
- Provides lifetime and subscription support on the state properties
- Enables dynamic construction of state data information elements and values

For more information see [15, 26, 27].

A limited number of experiments with the MDS3 service were conducted between test-bed sites. The basic test involved creating and starting an index service on one machine, registering local and remote services in it, and querying the index service to determine if the registered services and associated information were visible and current. Registration of services in the index from the client/service side is also possible.

A Brief summary of the steps involved for each index service related task follows (Based on the more detailed description in [23]).

1. To set up an Index Service (8 operations)
    a. Configure the server configuration file (2 operations)
    b. Configure index service configuration file (5 operations, then restart container)
2. To register a service to an Index Service (4 operations)
3. To test Service Data Aggregation (8 operations)
4. To register a service on one container to the index service on another host (4 operations)

In task 3 we discovered that the information entered during configuration did not seem to be checked until execution time ("success" was always reported even if incorrect information was entered for the "New Service Data Name/Namespace/from Provider"). Most of these operations require manual local server-side editing of configuration file entries. Higher-level tool support for both local and remote configuration and checking would be an improvement.

We also ran a version of the test-bed benchmark which exercised local container index services. This demonstrated that service data could be obtained (using pull based

querying) for each service, to determine things like the progress of the computation (using dedicated code in the service which updated the progress SDE when the next computational phase was entered), and some resource information (e.g. the number of active services per server). There was no measurable overhead (i.e. drop in throughput) for frequent polling (up to several times a second per service instance) of the index services for this information across the four node test-bed.

A number of open questions raised by our investigations are:

- What are the alternative ways of using the MDS3 service in our benchmark, and for the two architectural approaches? Can MDS3 information be used to provide more sophisticated load balancing?
- How does this approach to state visibility compare with approaches taken by Enterprise Java Beans? (Particularly related to issues of data consistency, persistence of data, state modification, etc). A partial answer is that it supports transparent service data persistence through the use of the Apache Xindice XML-native database platform.
- How does this approach to state visibility compare with the directions that UDDI and Web Services standards and technologies are heading? Will it be compatible or interoperable with them?
- What is the worst case behaviour expected due to the size of SDEs? What if the SDE is just the state of an entire database or long running computation? If SDEs are large, will queries become inefficient? Will they scale? What if a notification to a subset of a large SDE is desired, or notification that a subset of a large SDE has changed? Current conclusion is that these are indeed real problems.
- There are obviously security aspects related to registration and discovery of services in the MDS3 service. Some open questions remain including: Is the index service just like any other service as far as setting security goes? Can different security levels be set for registration and viewing entries in the index service? Can registration from external services be limited? What impact does the security setting of external services have on the ability to register these services?
- What are the intended use cases of the different features of the index service, and are there any documented examples in the literature?
- Reliability of contents. Service data may be out of date, and possibly inconsistent across caches and compared with original data sources etc. Is this an issue with the index or programming models? (E.g. [28]).

The GT3 MDS3 service model is relatively sophisticated, but complex, and we didn't experiment with it to the full extent required to evaluate it exhaustively for all possible uses. However, it does seem to address relatively obvious holes in the standard Web Services stack related to service management and information transparency. Integrated tool support for the secure remote management of index services and service registration is desirable. For more information about MDS3 see [23-31].

6 Data movement

This section of the report covers some approaches to data movement using GT3.2 (Based partially on [32]).

6.1 Direct SOAP encoding in the message body

Encoding data in the SOAP message body is the obvious approach to take as it is the default data interchange mechanism for Web Services. However, it is only appropriate for relatively small amounts of non-binary data, as the size of the encoded data is enormous, serialisation and de-serialisation take forever (and connection times out), and the JVM memory usage blows up. It thus only works for under 1MB of non-binary data.

6.2 SOAP attachments

SOAP messages with attachments [43] is the default Web Services approach to data movement for larger or binary data sets, and is supported by Axis [44]. However, there is no mention of SOAP attachments in the Globus documentation, and we received no response from globus newsgroups about the level of support offered by GT3.2, resulting in a series of experiments to try attachments with GT3.2. We discovered that up to 1GB of data can be transferred (with DIME, less with MIME), but not with GT3 security enabled. However, scalability would be non-existent with more than one transfer of this size at a time due to memory exhaustion. We did not reach a limit to the number of attachments per SOAP body (100,000 very small attachments worked). But, as Axis didn't clean up temporary disk files correctly (and a disk file is created for anything other than tiny attachments), unless the files are manually deleted, space on the file system was soon exceeded due to quotas. Because temporary disk files are created for even modest sized files a fair performance test couldn't be performed, as attachments should be processed in memory as long as there's sufficient available. Fixes need to include the ability to specify that attachments should be processed in memory below a dynamically configurable size threshold (or below a heap usage threshold), and automatic cleanup of temporary disk files.

Provisional results of these experiments were posted to the globus newsgroup in response to several requests for assistance [33], and are summarised here.

1. Axis attachments and GT3.2 test container don't work. The server throws an exception: org.xml.sax.SAXParseException: Content is not allowed in prolog.
2. There's a "feature" in axis attachment methods. Calling getAttachments() on the client side to check if the attachments were created results in the attachments being deleted before the message is sent.

3. SOAP/Axis attachments and GT3.2 with Tomcat work. There were initially reliability problems and eventually Tomcat wouldn't even start correctly. After a complete reinstallation it worked better for no apparent reason.
4. Using MIME attachments we can pass:
    a. 40,000 attachments, total size of 386MB before an exception: java.net.SocketException: Broken pipe
    b. Increasing memory on client/server doesn't help (from 1/2GB to 1GB).
5. Using DIME we can pass more (with 1GB heap):
    a. 90,000 attachments, total size of 869MB, but
    b. 100,000 attachments, total size of 966MB blows up with exception: java.lang.reflect.InvocationTargetException
    c. We could increase the memory above 1GB, but didn't.
6. GT3.2 security won't work with attachments. Not sure why, but as soon as client side security is turned on, no attachments are received on the server.
7. Single large attachments cause problems. There is a bug in Axis which fails to clean up temporary files, and no way to increase the threshold for the size of files that are processed in memory with no temporary file created.

6.3 GridFTP

GridFTP is a legacy Globus component and not well integrated with GT3 services and the container (basically because it's not OGSI compliant). It reuses the GT3 security, but requires a GridFTP server to run as well as the container. However, using GridFTP to pass data to a Grid service is far from ideal, as it requires the use of a separate (poorly documented) API, and the client and service must use additional mechanisms to coordinate transfer and lifecycle events (such as notification of data transfer completion) and file names, etc. We were unable to get it work correctly. The problems originated from poor documentation, lack of example code, bugs in the GridFTP server, and certificate issues.

6.4 Alternatives

More complex solutions exist, such as Globus' Reliable File Transfer (RFT, [45]) subsystem and the OGSA-DAI suite [34], but both of these seem overkill for the very simple task of moving data from A to B.

What about the far simpler approach of using wget? (essentially HTTP GET). Wget is used in MMJFS to stage both data and executables, and requires a web server on each machine (client and server). This is typical of the Globus assumption that each machine involved in the Grid is really a server and has most, if not all, the Globus infrastructure installed on it. The notion of a lightweight client machine with only "client-side" infrastructure installed is not widely supported.

Some other data transfer performance results include [35, 36]

More extensive tests of the different approaches, with different applications and configurations of services and servers should be carried out to determine the real impact of different data transfer approaches on performance (scalability, response times and reliability), application programming models, security, and other qualities of interest.

7 Security

As documented in Report 1 [13], cross-organisational Grid security is an intrinsically complicated area, as it must address security from multiple concurrent perspectives including intra- and inter- organisational policy and administration, grid infrastructure and deployment/configuration, users and communities (VOs) including developers, deployers, maintainers, debuggers, end-users, etc, and science security requirements (even in some cases legal and legislative requirements).

Critical attributes of the security infrastructure include usability (from the perspective of different tasks and roles), initial installation and configuration, scalability (setup, runtime, maintenance), functionality and flexibility (what levels of security are offered, how do they apply to subsets of users, resources, services, etc), impact of security on availability, how easy is it to debug, how manageable and maintainable is it, etc.

We have also noticed a distinction between the 1st and 2nd order approaches to security. The 1st order approach enables services to have different security settings, thereby allowing different users to use different services. However, the 2nd order approach, using MMJFS, can only apply one set of security policies to the execution of all jobs in a container – i.e. there is no way to differentiate users and security levels for different jobs. We also believe that there are potentially security issues surrounding index and data transfer services which require further investigation.

We discovered that there are complex issues surrounding the interaction of deployment and security: "Trust" with Systems Administrators, deploying a secured deployment infrastructure, and deploying and configuring secured grid infrastructure and services are different aspects of the problem, and offer their own challenges. Moreover, there are multiple deployment roles and tasks, each with their own security requirements. Consequently, the ability to automate secure deployment of secured grid infrastructure and services remains an unsolved problem.

8 Exception handling and Debugging

In a complex distributed system a well known problem that requires better understanding and infrastructure support is exception handling and debugging. Things go wrong, and there must be adequate means in place to understand them (predicting and averting

failure, recognising and recording failures, diagnosing and rectifying failures). Failures can occur in many parts of the system, and phases of the lifecycle, including deployment of infrastructure (even in the deployment of the deployment infrastructure), securing of infrastructure, deployment of services, use of services, and use of secured infrastructure and services, initial deployment and testing of an application, and subsequent use of the application (i.e. even if something works correctly once, it may not continue to work in the same way).

Programming and design of infrastructures and applications in the presence of security requirements and mechanisms is relatively poorly understood – in a sense, security constraints are reflected in the handling of exceptions – models of development which treat (combinations of) abnormal situations as first order requirements with an appropriate design process to address them are needed.

The problem is also related to the interaction of systems and roles as follows. It is painful to debug both *security* infrastructure, and *secured* grid infrastructure and applications. There must also be the ability to do "secure" debugging of both of these – i.e. how to debug securely without utilising or opening up potential security holes.

We believe that a possible approach to providing better support for debugging grid infrastructures and applications is motivated by two observations:

1. Many faults in distributed systems arise because of state management issues – typically the only way to fix them is to restart the offending part of the system to put it back into a workable state.
2. Deployment is obviously part of the problem, but maybe it's part of the solution to.

We therefore propose an approach to debugging called "deployment-aware debugging", based on a combination of an understanding of the relationship between an application and how it is deployed (including its run-time environment) in order to diagnose problems, and the use of a deployment infrastructure to attempt to rectify them.

In order to have an explicit relationship between applications and deployment, explicit deployment plans (or deployment-flows) can be generated from work-flows. Deployment-flows can be executed prior-to, concurrently, or interleaved with the application work-flows, and are used in conjunction with the deployment infrastructure to provide automatic, consistent, resource aware, JIT deployment (and un-deployment) of infrastructure and services required by the application.

If there is failure at the application level, then the deployment flow corresponding to the failure can be analysed to determine possible causes of failure, and then portions of the deployment flow can be re-executed in an attempt to correct the situation.

This is just an outline of the problems and the approach. More details can be found in [12], and the planned paper.

9 Conclusions

The difference between 1st and 2nd order "science as services" approaches roughly corresponds to OGSA/GT3 vs legacy Grid (E.g. GT2) approaches. Some of the differences have been highlighted, particularly from cross-organisational deployment viewpoints. For example, the trade-offs in deployment vs discovery and security: A 1st order approach is worse for service deployment, better for discovery and security; a 2nd order approach is better for "service" deployment, but worse for discovery and security. However, it is doubtful if there has been much progress in unifying the two views, or determining the full extent of the implications and tradeoffs of using one or the other approach for all tasks and roles, although a more focussed analysis is planned for a paper.

We conclude that there is a need for better understanding and research into the enactment of multiple tasks and roles associated with cross-cutting non-functional concerns across organisations. Production quality Grid middleware needs support (processes and tools) for remote, location independent, cross-organisational multiple role scenarios and tasks including:

- Automatic, platform independent, installation.
- Configuration and deployment creation, validation, viewing and editing.
- Managing grid, nodes, grid and supporting infrastructure, containers and services.
- Remote deployment and management of services.
- Secure remote distributed debugging of grid installations, services, and applications.
- Scalable security processes.

10 UK-OGSA Evaluation Project Presentations and Reports

16 March 2004	UK OGSA Evaluation Project, Status Report 16 March 2004, "Impact of GT3 termination, GT4, and WS-RF", UK-OGSA project NeSCForge site.
23 April 2004	Invited talk at the Core e-Science Programme Grid and Web Services Town Meeting on Grid and Web Services, presentation on UK OGSA Evaluation Project initial findings.
24 September 2004	UK-OGSA Evaluation Project Report 1.0: Evaluation of GT3.2 Installation Word document (28pp)
15 October 2004	Oxford University Computing Laboratory Talk: "Grid middleware is easy to install, configure, debug and manage - across multiple sites (One can't believe impossible things)" Power Point Document
1 November 2004	University College London, Computer Science Department Seminar: "Grid Middleware - Principles, Practice, and Potential" Power Point Document
February 2005	UK OGSA Evaluation Project, Report 2.0 (Final), Evaluating OGSA across organisational boundaries (This document, 27pp), http://sse.cs.ucl.ac.uk/UK-OGSA/Report2.doc

11 References

[1] The UK OGSA Evaluation Project: Establishment of an Experimental UK OGSA Grid, http://sse.cs.ucl.ac.uk/UK-OGSA/

[2] OGSA Evaluation Project NeSCForge site: https://forge.nesc.ac.uk/projects/uk-ogsa/

[3] The Future of Grid and Web Services, 28 January 2004, http://www.nesc.ac.uk/esi/events/385/

[4] UK OGSA Evaluation Project, Status Report 16 March 2004, Impact of GT3 termination, GT4, and WS-RF, UK-OGSA project NeSCForge site.

[5] NASA GT2 benchmark, http://www.nas.nasa.gov/News/Techreports/2004/PDF/nas-04-005.pdf

[6] SciMark2.0, http://math.nist.gov/scimark2/

[7] Stock-Online benchmark, http://jmob.objectweb.org/history.html

[8] Grid benchmarking, http://dps.uibk.ac.at/projects/zenturio/papers/grid2003.pdf

[9] Grid benchmarking, http://grail.sdsc.edu/projects/grasp/links.html

[10] Grid benchmarking, http://grid.ucy.ac.cy/Talks/GridBench-APART2Workshop.ppt

[11] http://www.sei.cmu.edu/pub/documents/01.reports/pdf/01tn025.pdf

[12] Brebner, P., "Grid Middleware - Principles, Practice, and Potential", University College London, Computer Science Department Seminar, November 2004, http://sse.cs.ucl.ac.uk/UK-OGSA/GridMiddlewarePPP.ppt
[13] UK-OGSA Evaluation Project Report 1.0: Evaluation of Globus Toolkit 3.2 (GT3.2) Installation, September 2004, http://sse.cs.ucl.ac.uk/UK-OGSA/Report1.doc
[14] Brebner, P., "Grid middleware is easy to install, configure, debug and manage - across multiple sites (One can't believe impossible things)", Oxford University Computing Laboratory, October 2004, http://sse.cs.ucl.ac.uk/UK-OGSA/ImpossibleThings.ppt
[15] Open Grid Services Infrastructure (OGSI), Version 1.0, http://www-unix.globus.org/toolkit/draft-ggf-ogsi-gridservice-33_2003-06-27.pdf
[16] Brebner, P., "Grid Deployment Requirements", UK-OGSA project NeSCForge site.
[17] SmartFrog, http://www.hpl.hp.com/research/smartfrog/, http://sourceforge.net/projects/smartfrog
[18] GT3.2 security installation, http://www-unix.globus.org/toolkit/docs/3.2/installation/install_installing.html#rootNonroot
[19] GT3.2 grid-map file requirements, http://www-unix.globus.org/mail_archive/discuss/2004/10/msg00276.html
[20] Grid Data Services, http://www.javaworld.com/javaworld/jw-08-2004/jw-0802-grid.html
[21] Impact of Staging, http://www.cs.utk.edu/~plank/plank/papers/CC-GRID-01.pdf
[22] MMJFS, http://www-106.ibm.com/developerworks/grid/library/gr-factory/?ca=dgr-lnxw03GridMMJFS
[23] Jake Wu, "GT 3.2 Index Service (MDS3) Evaluation", December 2004, UK-OGSA project NeSCForge site.
[24] Service Data Aggregation, http://www-unix.globus.org/toolkit/docs/3.2/infosvcs/ws/admin/testingaggreg.html
[25] GT3 Base Services Inside – Information Services, http://testbed.gridcenter.or.kr/kor/technical_doc/KISTI-IBM-0522/ppt/day2-04-GT%203%20Base%20Services%20Inside%20-%20Information%20Services.ppt#6
[26] MDS3, http://www-unix.globus.org/toolkit/mds/
[27] A developer's overview of OGSI and OGSI-based Grid computing, http://www-900.ibm.com/developerWorks/cn/grid/gr-ogsi/index_eng.shtml
[28] Index grid services using GT3, http://www-106.ibm.com/developerworks/grid/library/gr-indexgrid/?ca=dgr-lnxw02IndexGT3
[29] GT3 Index Service Users Guide, http://www.globus.org/ogsa/releases/final/docs/infosvcs/indexsvc_ug.html
[30] Jennifer Schopf, Grid Monitoring Futures with Globus, http://www.nesc.ac.uk/talks/190/JenniferSchopf.pdf
[31] GAIS, http://www.moredream.org/gais.htm
[32] Oliver Malham, "Grid Service Data Transfer Issues", December 2004, UK-OGSA project NeSCForge site.
[33] SOAP Axis attachments and GT3, http://www-unix.globus.org/mail_archive/discuss/2004/08/msg00257.html
[34] OGSA-DAI, http://www.ogsadai.org.uk/

[35] GridFTP Tests using GT3, http://lcg.web.cern.ch/LCG/peb/arda/public_docs/CaseStudies/GridFTP_3.x/
[36] Grid controlled Lightpaths for High Performance Grid Applications, http://bbcr.uwaterloo.ca/~iraqi/Papers/Journal/JOG-paper.pdf
[37] Increasing the Scope for Polymorph Prediction using e-Science, http://www.allhands.org.uk/2004/proceedings/papers/95.pdf
[38] UDDI v3.0, http://www.uddi.org/pubs/uddi_v3_features.htm
[39] OGSA-WG, http://forge.gridforum.org/projects/ogsa-wg/
[40] Wang, X., Allan, R., GT3.2 MMJFS Test Report, http://esc.dl.ac.uk/Testbed/mmjfs.pdf
[41] HTTP tunnelling of RMI, http://java.sun.com/developer/onlineTraining/rmi/RMI.html
[42] PERMIS, http://www.permis.org/en/index.html
[43] SOAP attachments, http://www.w3.org/TR/SOAP-attachments
[44] Axis, http://ws.apache.org/axis/java/index.html
[45] Globus RFT, http://www-unix.globus.org/toolkit/docs/3.2/rft/index.html