Space-Based Approach to High-Throughput Computations in UNICORE 6 Grids

Bernd Schuller and Miriam Schumacher

Jülich Supercomputer Centre
Distributed Systems and Grid Computing Division
Forschungszentrum Jülich GmbH
Jülich, Germany

Abstract. We explore a novel approach to high-throughput, embarassingly parallel applications in UNICORE 6 based Grids. This is an XML centric tuple space based approach inspired by JavaSpaces, with an implementation using the UNICORE 6 WSRF framework. Other approaches such as the layered workflow and data splitting architecture developed in the Chemomentum project, batch processing using the UNICORE commandline client UCC and concurrent programming using the HiLA Java API are discussed as well. Performance and scalability evaluations are presented, backed up by preliminary experimental results comparing our approach to the standard batch mode of the UNICORE commandline client.

1 Introduction

In this paper we explore an approach to high-throughput computing on UNICORE Grids that is based on an XML tuple space, i.e. a globally shared storage, that offers a simple API for storing,reading and removing arbitrary XML documents. The term high-throughput computing is used here in the sense that many relatively small computational jobs are run on a Grid system composed of relatively many compute nodes. To give an example, a typical run might involve several thousands of jobs, where each job consumes about 15 minutes of computational time, and with a number of compute nodes that is on the order of 100. For the purposes of this paper, the application throughput is assumed to be limited by the computation itself, not by the associated data transfers. High-throughput computing as defined here is highly relevant in many application fields from drug discovery to multi-media applications such as image or video rendering. A prominent example is in-silico screening of chemical substances using docking techniques, for example in the WISDOM initiative [2]. The main problems to be overcome are scalability, efficient resource discovery, selection and usage. Conventional approaches to resource discovery and selection involve information systems, where the resource providers have to publish detailed information about the state of their resources. This is often undesirable, as this information may be confidential. It is difficult to keep this information up-to-date, and the information system may become the bottleneck.

Additionally, sites may have to give up some of their autonomy to allow efficient resource management by the Grid scheduling systems. As will be shown, the tuple space-based approach removes the need for these information systems, and the sites can trivially enforce their local policies.

The remainder of the paper is organised as follows. Section 2 introduces the UNICORE 6 Grid middleware. Existing approaches to high-throughput computing using UNICORE are summarised in section 3, while our tuple space-based approach is introduced in section 4, Some preliminary performance results are given in section 5. A summary and outlook concludes the paper.

2 The UNICORE 6 Grid Middleware

UNICORE, developed in the course of several German and European projects since 1997 [1], is a mature Grid middleware that is deployed and used in a variety of settings, from small projects to large (multi-site) infrastructures involving high-performance computing resources. UNICORE can be characterised as a vertically integrated Grid system, that comprises the full software stack from clients to various server components down to the components for accessing the actual compute or data resources. Its basic principles are abstraction of site-specific details, openness, interoperability, operating system independence, security, and autonomy of resource providers. In addition, the software is easy to install, configure and administrate. The latest version is UNICORE 6 [4], which is based on Web Services and particularly the Web Service Resource Framework (WSRF). UNICORE is licensed under the liberal BSD license, and is available as open source from the SourceForge repository [3].

UNICORE 6 is a four-tiered system, consisting of the client, gateway, services and target system tiers. A wide variety of clients exist, from programming APIs [14], commandline client [15], simple Java clients to a rich client based on the Eclipse framework. The Gateway is a thin authentication and routing service that can be considered as a web service firewall and router. It resides outside the networking firewall, protecting the services behind it. Thus, UNICORE by default only requires a single open port to the public internet. The basic services (UNICORE atomic services) provide resource discovery (Registry service), job execution (Target System Factory and Target System services), and file access (Storage and FileTransfer services).

The target system tier consists of the interface to the local operating system, file system and resource management (batch) system. UNICORE 6 uses XML based standards in all functional areas: WS(RF)/SOAP for communication, JSDL for job submission, SAML assertions and XACML for authentication and authorisation.

The UNICORE 6 service registry contains the available target system factory (TSF) services, not the target system services (TSS) themselves. The reason for this is that each client (i.e. Grid user) creates their own target systems, which are accessible only for that particular client. The TSFs keep a list of TSS created by them. Thus, the discovery of available target systems is a fairly expensive operation involving several web service calls, because clients have to iterate over these TSS lists, and check for accessible services.

The basic sequence to run a computational job on UNICORE 6 is as follows.

- A suitable computational resource (target system service, TSS) needs to be found. If no TSS is available to the client, a suitable target system factory (TSF) must be discovered and invoked to create a TSS
- The job is submitted to the TSS, resulting in a new job management service (JMS) instance
- Input data can be staged in to the job's working directory
- The job is started. Usually the client sends a "start" message to indicate it has finished staging data in.
- After the job finishes, output data can be staged out.

At the time of writing (April 2008), UNICORE does not support client notification on job status changes, so a polling approach has to be used to find out if a job has finished.

3 High-Thoughput Approaches for UNICORE 6

3.1 Batch Mode of the Commandline Client

The UNICORE commandline client (UCC) [15] is a core component of UNICORE 6 and offers full access to the functions of a UNICORE 6 Grid. The UCC includes a batch processing mode, where a set of job files is read and jobs are submitted to the available compute resources. This batch mode can be used for high-throughput computation, where the user just has to generate the individual job files. There is no built-in fault handling, so the user has to deal with job failures herself, for example by re-running failed computations. Fault-handling features could however be added to UCC in the future. Resource discovery is performed by looking up target systems that offer the required application. More detailed brokering, for example by operating system or number of processors is not done by UCC. UCC selects resources using a round-robin strategy. A number of jobs are submitted concurrently, and their status is checked using a polling approach. The total number of concurrent jobs, the number of client threads used, and the polling interval used for job status updates can be controlled. This allows some tuning of the batch mode performance and controlling of the load generated on the Grid. UCC seems well suited for simple batch applications that do not require complex brokering or fault-handling strategies. Its scaling behaviour is fairly good due to its simplicity.

3.2 HiLA Java API

HiLA [14] is a Java API to a UNICORE Grid, offering a simple set of abstractions (such as Grid, Site, Task and File) and a familiar programming model. Using HiLA, applications can make use of Grid resources and run remote computations easily. HiLA can be used to develop high-throughput applications, with expected characteristics similar to the UCC batch mode.

3.3 UNICORE 6 / Chemomentum Workflow System

This workflow system has been developed within the European Chemomentum project [12]. It is fully integrated with the UNICORE middleware since the 6.1 release. The workflow system adds two layers to the basic UNICORE 6 architecture. A workflow engine layer deals with execution of high-level workflows, while a service orchestrator deals with resource discovery, selection and job execution. The system has been designed to allow easy scaling. The service orchestrator component is stateless in the sense that it operates on a per-job basis. Thus, multiple service orchestrator instances can be deployed to allow load-balancing. This workflow system can be used as-is for high-throughput computations, because it supports semi-automated data splitting and the service orchestrator component. The system is very simple to use for end-users, and needs no further programming or customisation work. The workflow engine receives a simple XML description of the task to be performed, with some workflow options that control the data splitting. The workflow engine then auto-generates a more detailed workflow with all sub-tasks specified.The sub-tasks are then sent to the service orchestrator which executes them on a suitable UNICORE 6 resource. The workflow system supports fault handling in the sense that failed computations can be repeated, and the system can deal with disappearing and newly appearing execution systems. At the time of writing, more elaborate, rule-based fault-handling is still under some development. The split and merge operations are performed internally, because a suitable UNICORE application for data splitting/merging has to be available on the Grid, and will be invoked automatically by the workflow engine. A special Resource Information Service (GRIS) is used for resource discovery. This service keeps Grid resource information which is periodically updated. Resource selection is performed by a brokering sub-component of the service orchestrator, which queries the GRIS and selects an execution resource based on current GRIS data and a set of configurable strategies. The basic strategy is based on application availability, combined with a round-robin approach in the common case of multiple execution host candidates. The Service orchestrator deals with job control, submitting jobs to the selected resources, checking their status, and sending notifications to the workflow engine when jobs succeed or fail. At the time of writing it is not yet clear how well this architecture scales in practice with increasing number of Grid sites, due to the limited deployment experiences.

4 Tuple Space Based Approach

The bottlenecks when using Grid systems for high-throughput computations are usually the resource discovery and selection processes. These are expensive operations, involve many web service calls, and tend to scale badly with increasing number of Grid resources. To completely bypass this procedures, we propose a different approach based on the tuple space concept.

A tuple space is essentially a shared memory accessible by distributed clients and servers. It stores data as records with typed fields (called tuples). The tuple space provides a small number of operations to insert, read and remove (take) tuples from the space, using template-based queries. The original concept was designed by David Gelernter and others for the Linda system in the mid-80s [5] and many implementations exist, for example several Java implementations based on SUNs JavaSpaces APIs [7]. Commercial implementations such as GigaSpaces [8] have gotten a lot of publicity recently due to their promise to deliver horizontally scalable, "share-nothing" enterprise architectures.

In an XML centric web-services system such as UNICORE 6 the idea to build a tuple space for XML documents is quite natural. XML-based tuple space implementations are not very common, however. It has been noted that XML and web services might be a promising way forward for Linda-like systems [6] especially in conjunction with web services. A .NET based XML tuple space was implemented by Tolksdorf et al. [9].

In the course of a diploma thesis [10], a tuple space for storing and retrieving arbitrary XML documents has been designed and implemented, based on the WSRFlite web services framework used in UNICORE 6. It is composed of two services, the Space service itself and a WSRF service for storing the tuple space entries. Each entry corresponds to a WS-Resource. Reading and taking entries involves matching the entries in a brute-force manner against a template.

4.1 Job Execution Using the Tuple Space

The XML documents used for realising the job execution application look as follows

```
<Job xmlns="http://www.unicore.eu/unicore6/spaces/job">
   <JobID>
   <ServerJobID>
   <ServerID>
   <Status>
   <Address>
   <JSDL>
</Job>
```

Here, the "JSDL" element stores the job description, and the "Status" field can take the values NEW, SUBMITTED and DONE. The other fields are used to store information relevant to the client, such as the endpoint reference of the UNICORE 6 job management service for managing the job.

As Figure 1 shows, the basic scheme is as follows:

- The client submit jobs to the space
- At the target system, a worker component (“job taker”) takes jobs from the tuple space and submits them to the target system and thus to the underlying UNICORE 6 XNJS execution manager
- When the job is done, the XNJS notifies the worker, and the job is written to the tuple space with status "DONE".
- The client checks for done jobs, and can download results

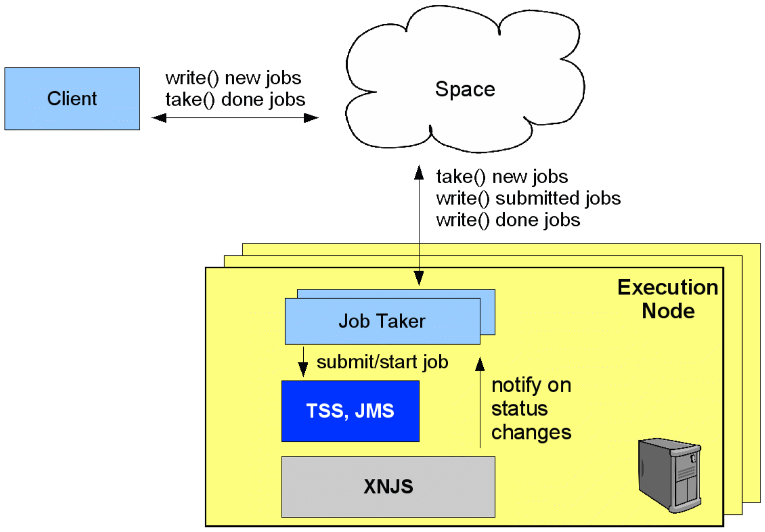

Fig. 1. Tuple space based job execution

This approach promises several advantages to the usual schemes. The tuple space can be seen as a globally shared queue, thus resource usage should be very efficient. The crucial point is that the job takers at the TSS decide when to fetch and submit the next job. This makes applying local scheduling policies trivial. Resource discovery and explicit resource selection by Grid clients are not necessary, nor is publishing of local scheduling information to a Grid scheduler. Another major benefit is that workers can be added (and removed) easily and transparently, without the need to make their presence known to other Grid components such as Grid schedulers or information systems.

The existing implementation is very simple, each job taker will just process exactly one job at a time. Of course, more complex policies are quite easy to implement (for example, on a cluster system it would be better to accept one job per compute node). Another limitation of the current implementation is that there is no well-defined order (e.g. FIFO) in which the jobs will be processed.

The UNICORE Spaces module including a prototype of the job execution application discussed in this section is available on the UNICORE Subversion repository [11]. This prototype does not include any security functionality (except for the standard TLS), the space is freely accessible, and jobs are submitted under the worker node's identity. However, we note that reasonable security mechanisms would be straightforward to add.

5 Some Performance Results

We performed some preliminary performance measurements on a small test Grid composed of six AMD Opteron 2GHz machines, with 2GB of memory, connected to the LAN with Gigabit Ethernet. They are running SUSE Linux 10.1 and Sun Java 1.5. Each UNICORE 6 service container is configured to use 128Mb of memory, and persistence is activated using the HSQLDB database. Node 1 hosts Gateway, Registry, XUUDB and the Space service, Nodes 2-5 are the workers, and Node 6 is used as client. The worker nodes are configured to run at most two jobs at a time.

Running 100 "Date" jobs (no data staging) on 4 worker nodes using the UCC batch mode took about 120 seconds, where we switched off the download of result files and the checks for application availability.

Our tuple space based job execution performed significantly better. Even using just a single worker, the 100 jobs were finished in about 100 seconds. This shows that in the space-based case there is much less overhead associated with each job.

Using two and four worker nodes, the 100 jobs were finished in 48 and 26 seconds respectively, showing linear scaling in this region. Also, each node consumed an approximately equal share of the total workload.

Similar scaling behaviour can be achieved with higher numbers of jobs, since the client limits the number of concurrent jobs to 100. When this limitation is removed, performance decreases slightly due to the longer lookup times in the space.

We have also measured the average time needed to lookup 100 random entries in the space, while varying the total number of entries. We find the linear increase that would be expected due to the brute-force lookup algorithm. The average lookup times are 12ms for 2000 entries, rising to 40ms for 10000 entries. This indicates a potential bottleneck: as the number of clients increases, the tuple space will become blocked for longer periods of time and the throughput will decrease. More measurements are needed to find the true limits here.

6 Summary and Outlook

The space-based approach presented here is highly promising for applications that do not have complex resource requirements, and that do not requiring high-level Grid features such as co-scheduling. For example, docking or other types of high-throughput screening are very well suited for this approach. Several issues remain though. Most importantly, real-world security requirements still have to be implemented. However, we are convinced that the XML space is flexible enough to handle these requirements. The excellent scalability characteristics would be very hard to achieve with other, more traditional Grid tools. Also, the space-based approach is very simple, and does not require complex broker and scheduler components.

Our preliminary performance numbers confirm the expectations we had when designing the system. Still, it remains to be seen how well the space-based approach scales up to higher numbers of worker nodes and concurrent clients. It should be expected that the performance of the central Space service will degrade under heavier loads, and load-balancing and clustering techniques will have to be employed. Also, the storage and lookup techniques for tuple space entries will have to be improved, to avoid needless searches and XML matching. For example, in the job execution application it would be highly beneficial to partition the tuple space using the job's Status field, which is used as the major search criterion in the application. Of course, the tuple space itself is generic, so the application programmer needs to provide some hints to the system how to do the partitiong. Similarly, indexes could be built on selected parts of the XML to decrease lookup times.

In summary, the basic overall simplicity of the tuple space concept and the applications in facilitates is clearly attractive and fits in very well with the UNICORE philosophy.

Acknowledgement

Part of this work was funded by the European Commission in the Chemomentum project (IST-5-033437).

References

1. Streit, A., Erwin, D., Lippert, T., Mallmann, D., Menday, R., Rambadt, M., Riedel, M., Romberg, M., Schuller, B., Wieder, P.: In: Grandinetti, L. (ed.) Grid Computing: The New Frontiers of High Performance Processing Advances in Parallel Computing, vol. 14, pp. 357–376. Elsevier, Amsterdam (2005)
2. Initiative for grid-enabled drug discovery against neglected and emergent diseases (April 2008), `http://wisdom.eu-egee.fr`
3. UNICORE website (April 2008), `http://www.unicore.eu`
4. UNICORE 6 overview (April 2008), `http://www.unicore.eu/documentation/files/Unicore6Overview.pdf`
5. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang.Syst. 7, 80–112 (1985)
6. Wells, G.: Back to the future with Linda. In: Second international workshop on coordination and apaptation techniques for software entities (in conjuction with ECOOP 2005), Oslo 2005 (April 2008), `http://wcat05.unex.es/Documents/Wells.pdf`
7. SUN Microsystems: Jini (April 2008), `http://java.sun.com/software/jini`
8. Gigaspaces commercial JavaSpaces implementation (April 2008), `http://www.gigaspaces.com`
9. Tolksdorf, R., Liebsch, F., Nguyen, D.M.: XMLSpaces.NET: An extensible Tuple Space as XML Middleware. In: 2nd International Workshop on.NET Technologies 2004 (submitted) (April 2008), `http://www.ag-nbi.de/research/xmlspaces.net`
10. Schumacher, M.: Realisierung eines XML-Tupelraumes unter Verwendung des Web Service Resource Framework, University of Applied Science Aachen/Juelich (February 2008)
11. UNICORE Spaces Subversion repository (April 2008), `http://unicore.svn.sourceforge.net/svnroot/unicore/contributions/unicore-spaces/trunk`

12. Chemomentum: Grid-Services based Environment for enabling Innovative Research (April 2008), `http://www.chemomentum.org`
13. Schuller, B., Demuth, B., Mix, H., Rasch, K., Romberg, M., Sild, S., Maran, U., Bała, P., del Grosso, E., Casalegno, M., Piclin, N., Pintore, M., Sudholt, W., Baldridge, K.K.: Chemomentum - UNICORE 6 based infrastructure for complex applications in science and technology. In: Bougé, L., Forsell, M., Träff, J.L., Streit, A., Ziegler, W., Alexander, M., Childs, S. (eds.) Euro-Par Workshops 2007. LNCS, vol. 4854, pp. 82–93. Springer, Heidelberg (2008)
14. HiLA High-level API for Grids (April 2008), `http://www.unicore.eu/community/development/hila-reference.pdf`
15. UNICORE commandline client (April 2008), `http://www.unicore.eu/documentation/unicore6/manuals/ucc`