Universiteit Gent
Faculteit Ingenieurswetenschappen
Vakgroep Informatietechnologie

Architecturen en algoritmen voor netwerk- en dienstbewust Grid-resourcebeheer

Architectures and Algorithms for Network and Service Aware Grid Resource Management

Bruno Volckaert

Proefschrift tot het bekomen van de graad van
Doctor in de Ingenieurswetenschappen:
Computerwetenschappen
Academiejaar 2005-2006

Universiteit Gent
Faculteit Ingenieurswetenschappen
Vakgroep Informatietechnologie

Promotoren: Prof. Dr. Ir. Bart Dhoedt
Prof. Dr. Ir. Filip De Turck

Universiteit Gent
Faculteit Ingenieurswetenschappen

Vakgroep Informatietechnologie
Gaston Crommenlaan 8 bus 201, B-9050 Gent, België

Tel.: +32-9-331.49.00
Fax.: +32-9-331.48.99

Dit werk kwam tot stand in het kader van een specialisatiebeurs van het IWT-Vlaanderen (Instituut voor de aanmoediging van Innovatie door Wetenschap en Technologie in Vlaanderen).

Proefschrift tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen:
Computerwetenschappen
Academiejaar 2005-2006

Dankwoord

Plots is het er dan, het beginnen schrijven aan je doctoraatsthesis, de samenvatting van 4 jaar doctoraatsonderzoek, eindstation van een lange en boeiende reis in zicht. De trein was nochtans traag op gang gekomen: het inwerken in je onderwerp vraagt veel opzoek- en leeswerk, waarbij je vooral het kaf van het koren moet proberen scheiden. Eens dit stadium voorbij, en na het behalen van een specialisatiebeurs, hadden we al een beter zicht op wat “Grids” nu eigenlijk waren en hoe ver het onderzoek naar deze nieuwe vorm van gedistribueerde verwerking gevorderd was. Kort daarop begonnen we te werken aan onze eigen inbreng en, ook al begaven we ons af en toe enkele dagen op een verkeerd spoor, al gauw waren we niet meer te stoppen.

Tijdens deze boeiende en leerrijke periode kwam ik vanzelfsprekend ook in contact met tal van mensen die onontbeerlijk waren bij het tot stand komen van dit werk. Bij deze wil ik die mensen dan ook eens in de bloemetjes zetten, en hun bedanken voor de steun en het vertrouwen dat ze in me stelden. Eerst en vooral wil ik prof. Paul Lagasse bedanken voor het gebruik mogen maken van de uitgebreide faciliteiten van onze vakgroep. Verder wil ik ook prof. Piet Demeester, prof. Bart Dhoedt, prof. Filip De Turck en dr. Stefaan Vanhastel bedanken om me de kans te geven doctoraatsonderzoek uit te voeren, voor het advies tijdens de vele brainstorm sessies en de bijbehorende begeleiding. Pieter Thysebaert verdient zeker ook een uitgebreid woord van dank, eerst en vooral omdat veel van ons onderzoeks-, programmeer- en schrijfwerk in nauwe samenwerking verricht is, maar ook voor de vlotte, aanvullende en behulpzame manier waarop dit alles plaatsvond. Naast Pieter waren ook Marc De Leenheer en Tim Wauters van belang bij het produceren van onderzoeksresultaten en het redigeren van papers, alsook onze partners in “Grid-crime”: Maria Chtepen en Jurgen Baert. Mijn oprechte dank gaat bovendien uit naar het Instituut voor aanmoediging van Innovatie door Wetenschap en Technologie in Vlaanderen (IWT) voor de financiële steun die ik gedurende 4 jaar mocht ontvangen.

Geen werk zonder administratie, en daarbij is onze vakgroep in excellente handen: Martine Buysse, Ilse Van Royen, Davinia Stevens, Karien Hemelsoen, Ilse Meersman, Marleen Van Duyse en Bernadette Becue zijn maar enkele van de personen waar ik regelmatig wel eens bij terechtkwam met vragen omtrent onkostennota’s, reisaanvragen, buitenlijnen en dergelijke meer. Engelengeduld moeten ze

gehad hebben, toen ik nog maar eens mijn personeelsnummer niet terugvond, of voor de zoveelste keer de fax niet kon laten doen wat ik het wou laten doen (in dit geval faxen).

Ook de mensen die verantwoordelijk zijn voor het goed functioneren en onderhouden van ons computerpark en netwerk verdienen een bedankje. Niet in de minste plaats dr. Brecht Vermeulen, Bert De Vuyst, Pascal Vandeputte en (in het geval van onze Gridinfrastructuur) Stijn De Smet. Dankzij hen waren er nooit grote oponthouden toen hardware en/of software samen met de occasionele stroompanne alweer eens voor problemen zorgde. Dag en nacht kon je wel bij een van hen terecht voor een “quick fix”.

Verder wil ik ook van de gelegenheid gebruik maken om eens mijn bureaugenoten in de spreekwoordelijke verf zetten: dr. Erik Van Breusegem, Liesbeth Peters, Sofie Verbrugge, Filip De Greve, Thomas Bouve, Nico Goeminne, Tom Verdickt, Jurgen Baert en onze nieuwste aanwinst: Stefanie De Maesschalck. In de eerste plaats wil ik dit doen omdat ze altijd te vinden waren voor een ontspannende en leuke babbel terwijl ik hen van mijn kant nochtans teisterde met onregelmatige aanwezigheidsuren en een nogal onstuimig ventilerende desktop pc.

De boog kan niet altijd gespannen zijn, dus met enige regelmaat was er ook wel tijd voor wat ontspanning. Daarbij wil ik zeker mijn badmintonvrienden een pluim geven: prof. Bart Dhoedt, Koert Vlaeminck, dr. Chris Develder, Thijs Lambrecht en de vele freelancers die af en toe meespeelden. Meer dan eens bleven we stevig nakaarten, waarna we de volgende dag (meestal) weer fris aan de aftrap van onze onderzoeksactiviteiten stonden. Dr. Steven Van den Berghe en Sofie Van Hoecke verdienen hierbij zeker ook een vermelding als regelmatige chatpartners en aanbrengers van werktips en levenswijsheden. In dit kader wil ik ook enkele andere dichte vrienden en prettig gestoorde uitgaansmedeplichtigen, met name Kristof Van Landschoot, Nico Bataillie, Matthias Heyneman en Miguel de Bruyckere bedanken voor de vele toffe avonden in Gent en sinds kort ook het verkennen van Maldegem. Ook m’n vriendin Sarah zou ik bij deze willen bedanken voor de vele plezante jaren samen.

Als slot zou ik graag nog mijn familie extra willen bedanken voor de vele hulp en onvoorwaardelijke steun die ik reeds al die jaren mogen ontvangen heb, en waar ik in goede en kwade tijden steeds bij terecht kon. Zonder jullie zou ik niet staan waar ik vandaag sta.

Hoogstwaarschijnlijk heb ik in dit dankwoord nog vele mensen vergeten vermelden, en daarom wil ik ook u, als lezer, bedanken voor de interesse in m’n werk.

Gent, maart 2006
Bruno Volckaert

Table of Contents

List of Figures

List of Tables

List of Acronyms

A

ACE	Adaptive Communication Environment
ASAP	As Soon As Possible

C

CDR	Common Data Representation
CM	Connection Manager
CORBA	Common Object Request Broker Architecture
CR	Computational Resource

D

DIT	Directory Information Tree
DN	Distinguished Name
DR	Data Resource

E

EGEE	Enabling Grids for E-Science in Europe

F

FCFS	First Come First Served

G

GA	Genetic Algorithm
GGF	Global Grid Forum
GIIS	Grid Information Index Server
GMA	Grid Monitoring Architecture
GRIS	Grid Resource Information Service
GS	Grid Scheduler
GUI	Graphical User Interface

H

HTTP	HyperText Transfer Protocol

I

IAT	Inter Arrival Time
ILP	Integer Linear Program
IP	Internet Protocol
IS	Information Service
ISP	Internet Service Provider

J

JAMM	Java Agents for Monitoring and Management

L

LAN	Local Area Network
LCG	LHC Computing Grid
LDAP	Lightweight Directory Access Protocol
LHC	Large Hadron Collider

M

MAN	Medium Area Network
MDS	Globus Monitoring and Discovery Service
MPLS	MultiProtocol Label Switching

N

NWS	Network Weather Service

O

OGSA	Open Grid Services Architecture
OGSI	Open Grid Services Infrastructure

Q

QoS	Quality of Service

R

R-GMA	Relational Grid Monitoring Architecture
RM	Replica Manager
RTT	Round Trip Time

S

SM	Service Manager
SMON	Service Monitor
SOAP	Simple Object Access Protocol
SQL	Standard Query Language

SR	Storage Resource

T

TCP	Transport Control Protocol
TIO	Tivoli Intelligent Orchestrator
TPM	Tivoli Provisioning Manager

U

UDP	User Datagram Protocol

V

VO	Virtual Organisation
VPG	Virtual Private Grid
VPN	Virtual Private Network

W

WAN	Wide Area Network
WS-IS	Web Service Information Service
WSDL	Web Service Description Language
WSRF	Web Service Resource Framework

X

XML	eXtensible Markup Language

Nederlandse samenvatting –Summary in Dutch–

De grote verspreiding van het internet en de beschikbaarheid van krachtige computers en hogesnelheidsnetwerken tegen een relatief lage kostprijs geeft ons tegenwoordig de mogelijkheid om een ruime variëteit aan gedistribueerde resources met elkaar te verbinden en, mits voorzien van de nodige software, gebruik te maken van hun gebundelde kracht. Deze aaneenkoppeling van heterogene, dynamische resources tot een geünificeerde, krachtige machine wordt "metacomputing" of "Grid computing" [1, 2] genoemd. Toepassingen die nuttig gebruik kunnen maken van een Gridinfrastructuur zijn onder meer gedistribueerde dataverwerking (v.b. CERN DataGrid [3]), computationeel intense jobverwerking en geavanceerd collaboratief onderzoek (vb. EScience Grid [4]). Tegenwoordig vereisen bepaalde toepassingen méér rekenkracht dan de huidige clusters en supercomputers kunnen leveren en kunnen daarom enkel economisch verantwoord uitgevoerd worden in de context van een Grid.

Vanuit de bedrijfswereld en onderzoeksinstellingen is er dan ook een grote interesse in software die eenvoudige ingebruikname van Grids en de bijbehorende Gridapplicaties toelaat. Deze software moet toelaten om gedistribueerde, heterogene resources op een transparante, veilige en performante manier te gebruiken, en is reeds beschikbaar onder de vorm van enkele standaard Grid middleware toolkits zoals Globus [5] en de LCG middleware [6]. Tot voor kort werd er echter weinig onderzoek gedaan naar de invloed van het netwerk op onder meer de algehele resource-efficiëntie en de optimaliteit van resource-assignaties door Gridschedulingalgoritmen. Dit is deels te verklaren vanuit de historische verbondenheid van Grids met het clustercomputingconcept. Bij clustercomputing kan verwerking ook gedistribueerd plaatsvinden, maar bevinden de resources zich in tegenstelling tot Grids allemaal op geografisch dezelfde locatie en zijn deze geïnterconnecteerd door middel van een hogesnelheidsnetwerk met aanzienlijk kleinere vertragingen. Hierdoor is de status van netwerkresources bij het nemen van o.a. schedulingbeslissingen minder belangrijk dan bij Grids. Bovendien is de resource- en topologieconfiguratie van een cluster volledig gekend en kan de status eenvoudiger gecontroleerd worden dan bij geografisch gedistribueerde en dynamische Gridsystemen.

Het werk dat we in dit boek presenteren kan opgesplitst worden in meerdere delen. Allereerst bespreken we hoe we een performante en schaalbare Grid-monitoringarchitectuur ontwikkeld hebben met als doel realistische waarden met be-

trekking tot typische job- en resourcekarakteristieken op te meten. De algemene opbouw van deze monitoringarchitectuur (met sensoren, producenten, consumenten en een directory service) wordt uit de doeken gedaan, alsook de gebruikte implementatietechnologieën. We vergelijken de performantie en functionaliteit van het ontwikkelde Gridmonitoringplatform met die van andere monitoringpakketten (o.a. Globus Monitoring and Directory Service 2 - MDS2 [7]) en moeten na uitgebreide metingen concluderen dat ons monitoringplatform met een minimum aan systeembelasting de beste performantie en schaalbaarheid aanbiedt. Zo is de processorbelasting bij het simultaan bevragen van een producer uit ons monitoringplatform door 600 gebruikers lager dan 6 procent, waarbij een antwoord geformuleerd wordt met een gemiddelde responstijd van 0,01 seconde (wat een factor 10 beter is dan bij MDS2). Als we de performantie van de directory services met elkaar vergelijken, zien we dat onze oplossing zo'n 458 bevragingen per seconde kan afhandelen, terwijl dit bij MDS2 slechts 311 bevragingen per seconde is. Deze resultaten worden mogelijk gemaakt door de combinatie van de gebruikte implementatietechnologieën met een schaalbaar architecturaal ontwerp.

In een tweede stadium bespreken we de nood aan een Gridsimulator die in staat is om netwerkinterconnecties accuraat voor te stellen en tegelijkertijd toelaat om een grote variëteit aan Gridtopologieën en resourceconfiguraties te modelleren. Dit leidt tot de ontwikkeling van NSGrid, een op ns-2 [8] gebaseerde Gridsimulator die modellen bezit om jobs en resources (computationele, opslag-, data- en netwerkresources) voor te stellen. NSGrid is bovendien voorzien van verschillende Gridmanagementcomponenten, zoals een scheduler, replica-, netwerk- en servicemanager, monitoring- en informatiediensten. Vervolgens worden schedulingalgoritmen voorgesteld die rekening houden met de toestand van resource-interconnecties om performante jobschedules op te stellen. Deze algoritmen worden geëvalueerd op verschillende Gridtopologieën en jobpatronen, gebruik makend van NSGrid. Uit de resultaten blijkt dat het opnemen van de status van netwerkconnecties in het schedulingproces kan leiden tot betere jobassignaties: bij Gridinterconnectiebandbreedtes van 10 Mbps wordt een gemiddelde jobresponstijd opgemeten die 61 procent beter is dan wanneer geen netwerkinformatie opgenomen wordt in het schedulingproces. Bovendien tonen we bij hetzelfde scenario aan dat computationele resourcereservaties gemiddeld 30 procent van hun tijd ongebruikt worden, wegens het wachten op invoer- en/of uitvoerdata (wat vermijdbaar is indien er correct rekening gehouden wordt met de status van de netwerkresources op het moment van schedulen).

Er wordt dieper ingegaan op een geautomatiseerde service-managementarchitectuur die, rekening houdend met het gemonitorde applicatiegedrag, een verdeling van Gridresources over verschillende serviceklassen afdwingt. Serviceklassen zijn in dit geval een verzamelnaam voor jobs die eenzelfde gedrag vertonen met betrekking tot hun gevraagde verwerkingskracht en data-intensiteit. Deze verdeling van resources over serviceklassen met verschillende prioriteiten, samen met het dynamisch instantiëren van managementcomponenten (scheduler, informatiediensten, etc.) exclusief toegewijd aan die serviceklassen, laat toe een Grid op te splitsen in meerdere "Virtual Private Grids", elk afgestemd op de karakteristieken

van een welbepaalde serviceklasse. We bespreken hierbij verschillende resourcepartitioneringsalgoritmen gebaseerd op genetische algoritmen, en evalueren de performantie van deze algoritmen gebruik makend van NSGrid. Uit de resultaten blijkt dat niet alleen de gemiddelde computationele en netwerkresource-efficiëntie verbeteren (met gemiddelde winsten tot respectievelijk 17 en 5,5 procent), maar ook dat gemiddelde jobresponstijden omlaag gaan (bij scheduling waarbij rekening gehouden wordt met de status van netwerkresources kan dit tot 30,5 procent winst opleveren) en dat het mogelijk wordt om geautomatiseerd serviceklasseprioriteiten af te dwingen.

Tot slot wordt in dit werk een recente ontwikkeling besproken: het gebruik van Gridtechnologie voor de verwerking van digitale audiovisuele data bij organisaties verantwoordelijk voor radio- en televisieproducties. Verschillende organisatieprofielen (gebruikers, applicaties, vereisten, etc.) worden voorgesteld en kunnen gesimuleerd worden met MediaNSG, een uitbreiding op NSGrid die in staat is om het gebruik van Gridtechnologie in de audiovisuele sector na te bootsen.

Referenties

[1] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[2] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure 2nd Edition*. Morgan Kaufmann, 2003.

[3] *CERN*. `http://www.cern.ch/`.

[4] *Enabling Grids for E-Science in Europe*. `http://egee-intranet.web.cern.ch`.

[5] I. Foster. *Globus Toolkit Version 4: Software for Service-Oriented Systems*. Lecture Notes in Computer Science, 3779:2–13, 2005.

[6] *LHC Computing Grid project*. `http://lcg.web.cern.ch/LCG`.

[7] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. *Grid Information Services for Distributed Resource Sharing*. In Proc. of the 10th IEEE International Symposium on High-Performance Distributed Computing, 2001.

[8] *The Network Simulator - NS2*. `http://www.isi.edu/nsnam/ns`.

English summary

Widescale adoption of the internet and the availability of powerful computational resources and high-bandwidth networks at a relatively modest cost, presents us with the opportunity to interconnect a wide variety of distributed resources in order to harness their combined capabilities. This interconnection of heterogeneous, dynamic resources in order to construct a single unified, powerful machine is called "metacomputing" or "Grid-computing" [1, 2]. Applications that can use this Grid infrastructure to their advantage are among others distributed dataprocessing (e.g. CERN DataGrid [3]), computationally intense batch processing and advanced collaborative research (e.g. EScience Grid [4]). Nowadays, some applications demand more computational power than can be provided by current clusters or supercomputers, and can therefore only be processed (under economically sain conditions) when utilising a Grid.

Industrial and research communities are increasingly more interested in software that allows easy deployment of Grids and the accompanying Grid applications. This software has to support utilising distributed, heterogeneous resources in a transparent, secure and well-performing manner, and is readily available in the form of standard Grid middleware toolkits such as Globus [5] and the LCG middleware [6]. Until recently however, little research has been done on the influence of the network on (among others) computational resource efficiency and the optimality of assigning resources to jobs by different Grid scheduling algorithms. This can be partially explained because of the historic connection of Grids with the cluster computing concept. Cluster computing allows jobs to be processed in a distributed way, but the computational and storage resources are (in contrast with Grids) all located at the same geographical location and are interconnected by means of high speed network links with relatively small delays. This renders the status of network resources when taking scheduling decisions less important than when scheduling jobs in a Grid. On top of this, when a cluster is employed for processing jobs, its resource and topology configuration is fully known making it easier to monitor resource state than when a geographically distributed and dynamic system (such as a Grid) is used.

The work we present here can be subdivided in multiple parts. First off, we discuss how we have developed a performant and scalable Grid monitoring platform capable of delivering realistic values regarding typical job and resource characteristics. The overall architectural structure of this monitoring platform (with sensors, producers, consumers and a directory service) will be explained, together with the implementation methods that were used. We compare performance and function-

ality of the developed Grid monitoring platform with that of existing leading monitoring platforms (a.o. Globus Monitoring and Directory Service 2 - MDS2 [7]) and have to conclude that our monitoring platform offers the best performance and scalability, with a minimum of resource load; processor load when simultaneously querying a producer by 600 users is lower than 6 percent, with query answers formulated with an average response time of 0.01 seconds (which is ten times better than MDS2). If we compare the performance of the directory services, we notice that our solution can handle 458 queries per second, while MDS2 can only handle 311 queries per second. These results are made possible because of the combination between utilised implementation technologies and an architectural design focussed on scalability.

In the next part we discuss the need for a Grid simulator that accurately represents network interconnections and at the same time allows modelling of a large variety of Grid topologies and resource configurations. This leads to the development of NSGrid, a Grid simulator based on ns-2 [8] offering models to represent jobs and resources (computational, storage, data and network resources). NSGrid also provides different Grid management components: schedulers, replica managers, network managers, service managers, monitoring and information services, etc.. We present Grid scheduling algorithms that take into account the state of resource interconnections to construct well-performing job schedules. These algorithms are then evaluated on different Grid topologies and job patterns, utilising the NSGrid simulator. The results show that taking into account the state of network connections when scheduling jobs on a Grid can lead to better job assignations: with Grid interconnection bandwidths of 10 Mbps we measure average job response times that are 61 percent better than when no network information is included in the scheduling process. For the same scenario, we show that computational resource reservations spend on average 30 percent of their time idling, while waiting for input and/or output data to arrive/be sent (which can be avoided when taking into account the state of network resources at the time of scheduling).

We give an in-depth discussion of an automated service management architecture that, taking into account monitored Grid application behaviour, enforces a partitioning of the Grid resource pool among the different service classes. Service classes in this case are collections of jobs that present similar behaviour regarding their processing and I/O requirements. This partitioning of resources amongst service classes with different priorities, together with the dynamic deployment of management components (scheduler, information services, etc.) for exclusive use by those service classes, allows subdividing a Grid into multiple “Virtual Private Grids”, each tuned to the characteristics of their service class. We discuss different resource partitioning heuristics based on genetic algorithms and evaluate the performance of these heuristics using NSGrid. The results show that computational and network resource efficiency improve (with average improvements of 17 and 5.5 percent respectively), combined with lower average job response times (when employing a network aware scheduling algorithm this can lead to 30.5 percent better job response times) and that it becomes possible to automatically enforce service class priorities.

Finally, a recent trend will be discussed: the use of Grid technology for processing digital audio/visual data in organisations that are responsible for radio and television production / distribution. Different organisation profiles (users, applications, requirements, etc.) are presented and can be simulated by means of MediaNSG, an extension to NSGrid capable of modelling different media company Grid deployments.

References

[1] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[2] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure 2nd Edition*. Morgan Kaufmann, 2003.

[3] *CERN*. `http://www.cern.ch/`.

[4] *Enabling Grids for E-Science in Europe*. `http://egee-intranet.web.cern.ch`.

[5] I. Foster. *Globus Toolkit Version 4: Software for Service-Oriented Systems*. Lecture Notes in Computer Science, 3779:2–13, 2005.

[6] *LHC Computing Grid project*. `http://lcg.web.cern.ch/LCG`.

[7] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. *Grid Information Services for Distributed Resource Sharing*. In Proc. of the 10th IEEE International Symposium on High-Performance Distributed Computing, 2001.

[8] *The Network Simulator - NS2*. `http://www.isi.edu/nsnam/ns`.

1 Introduction

1.1 An introduction to Grid computing

The main workhorses for processing computationally complex problems have long been supercomputers and clusters, with work often originating from the research and development community. However, problems are becoming increasingly demanding, challenging the capabilities of even the most powerful supercomputer or cluster systems. This led to the idea of joining resources to solve these problems in a reasonable time frame, effectively interconnecting geographically remote computational, storage and data resources into a single number crunching system. As a first step in realizing this concept, the maturation of the Internet in the nineties led to the first global distributed computing projects. Two projects in particular have proven that the concept works extremely well. The first project, distributed.net [1], used thousands of independently owned computers across the Internet to crack encryption codes. The second is the SETI@home project [2]. Over two million people have installed the SETI@home software agent since the project's start in May 1999. This project proved that distributed computing could accelerate computing project results while at the same time managing project costs (IBM's ASCI White [3] supercomputer is rated at 12 TeraFLOPs and costs $110 million. SETI@home currently gets on average 15 TeraFLOPs and has cost $500K so far).

Grid computing is increasingly being viewed as the next phase of distributed computing and enables organizations to share computing, storage and data resources across department and organizational boundaries in a secure, efficient

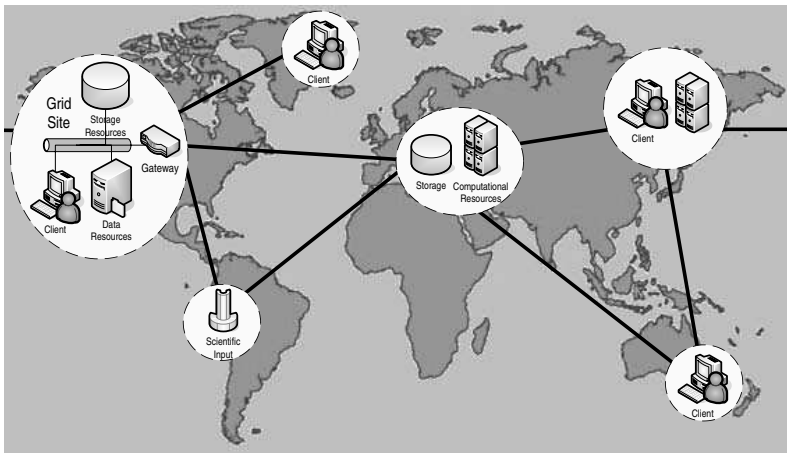


Figure 1.1: Global Grid computing

manner. One of the main motivations for this new computing paradigm lies in the observation that the demand for resources is ever growing while on the other hand vast resource pools remain underused. Grid technology aims at solving this mismatch by offering its users transparent access to a variety of resources, making abstraction of the exact geographic location of the physical resource.

Although some similarities exist, Grid computing differs from cluster computing in a number of key aspects. First, due to the geographic distribution of Grid resources (see figure 1.1), a Grid does not have a central administration point (instead it consists of resources from multiple administrative domains), whereas all cluster resources can usually be administered from one location. Second, this geographic distribution entails drastically different resource usage policies and heterogeneity of equipment: a variety of resources will be connected by a wide range of network technologies, whereas a cluster will usually consist of a large collection of homogeneous resources interconnected by a proprietary bus or high speed / short range network links. This again indicates an important distinctive Grid feature: communication links can be long haul, possibly subject to congestion, while the Grid topology itself is subject to frequent change, due to the possibly dynamic nature of resources and the decentralized authority over resource usage (this dynamic behaviour can easily be spotted in the case of SETI@home type Grids, based on desktop pc's donating unused CPU cycles).

The term "Grid computing" [4, 5] suggests a computing paradigm similar to the operation of an electric power grid: the same way an electrical outlet delivers power without the consumer knowing exactly where that electrical power is generated, a variety of geographically dispersed resources can be transparently joined into a shared resource pool for consumers to access on an as-needed basis. Al-

though this ideal is still a few years off, standardization is the key to realizing Grid computing benefits, so that the diverse resources that make up a modern computing environment can be discovered, accessed, allocated, monitored, and in general managed as a single virtual system - even when provided by different vendors and/or operated by different organizations. The Global Grid Forum (GGF [6]) acts as a standardization body for the Grid, comparable in terms of philosophy as the IETF [7] for Internet related matters. The GGF is a community-initiated forum of thousands of individuals from industry and research leading the global standardization effort for Grid computing. The GGF's primary objectives are to promote and support development, deployment, and implementation of Grid technologies and applications through the creation and documentation of "best practices" - technical specifications, user experiences, and implementation guidelines. Some notable GGF groups are the Open Grid Services Architecture working group (OGSA-WG [8]), Grid High-Performance Networking research group (GHPN-RG [9]), the Grid Scheduling Architecture research group (GSA-RG [10]) and the Job Submission Description Language working group (JSDL-WG [11]).

Today, organizations around the world are utilizing Grid computing in diverse areas as collaborative scientific research, drug discovery, financial risk analysis and product design. Grid computing enables research-oriented organizations to solve problems that were infeasible to solve due to computational and data-integration constraints. Grids also reduce costs through automation and improved resource utilization. Finally, Grid computing can increase an organization's agility enabling more efficient business processes and greater responsiveness to change.

Based on their main application area, Grid systems can be divided in three classes: computational Grids, data Grids and service Grids. Computational Grids are tailored to provide huge amounts of processing power (e.g. NEESgrid for earthquake simulation studies [12]). Cycle scavenging Grids are special cases of computational Grids: they allow desktop users to donate their idle CPU time to help scientific research (mostly global interest projects e.g. SETI@home based on the BOINC [13] Grid framework, Folding@home [14] for research regarding protein folding which could in time lead to cures for diseases like Alzheimer and Parkinson, fightAIDS@home [15]).

A second Grid system class is coined the term "data Grid". These synthesize new information from distributed data repositories. The Large Hadron Collider Computing Grid (LCG [16]) project is a well-known example of this type of Grids. In the LCG project data generated by CERN's Large Hadron Collider [17] (which is to be operational in 2007 and will roughly produce 15 Petabytes annually) will be distributed around the globe, according to a four-tiered model. Thousands of high energy physics scientists from around the world will access and analyse fragments of this data.

Service Grids are Grids that do not focus on batch job processing, but instead

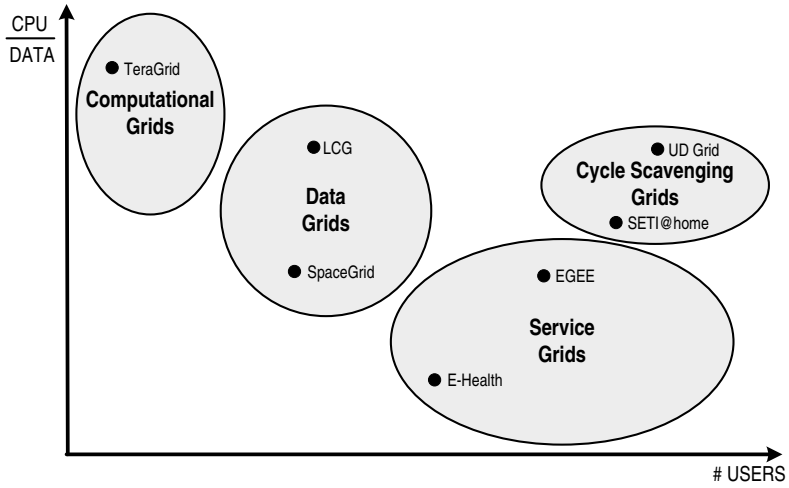


Figure 1.2: Grid taxonomy

offer access to a wide variety of real-time services that cannot be provided by a single machine. Services can range from collaborative working (enabling interaction between users and applications through a virtual workspace), to multimedia Grids and "on demand" Grids, enabling a user to dynamically increase the amount of machines processing on its jobs. A thorough look at Grid taxonomy can be found in [18].

Current Grid deployments include the Belgian BEgrid [19], covering 6 Grid sites for a total of 259 computational resources and 3.4 TB of storage space interconnected by a 2.5 Gbps backbone. BEgrid in turn is connected to the Enabling Grids for E-Science in Europe (EGEE [20]) Grid which houses 240 TB of storage space and 1846 KSI2K of computational power (kiloSpecINT2000 is the computational power rating based on the SpecINT2000 benchmark [21]) interconnected by network links varying in bandwidth between 34Mbps and 10Gbps. Currently, one of the most powerful computational Grid deployments is the US-based TeraGrid [22], which interconnects 8 data and computing centres providing a total of 50 TeraFlops of processing power and 1046 TB of on-line storage space for scientific purposes. The network connectivity of each TeraGrid site ranges from 10Gbps to 30Gbps.

We refer to appendix A for a more elaborate introduction to Grid computing.

1.2 Problem statement

Despite the current deployment of operational Grid systems, important research challenges still exist. Whereas initial Grid research mainly focused on tackling

batches of computationally intense problems and parameter sweep applications, the rise of data and service based Grids (e.g. the aforementioned EGEE project) is imposing new requirements in terms of responsiveness, scaling behaviour, Quality of Service-support, network requirements and robustness. Since one of the main characteristics of a Grid is its geographically distributed nature, the network interconnecting the different computational, storage and data resources should be treated with the same importance as any other resource, as it can pose a significant bottleneck. With this in mind, and noting that the deployment of various Grid configurations for testing/research purposes is a time-consuming (if not impossible) feat, accurate network aware Grid simulation becomes a necessity in order to be able to easily experiment with new scheduling and resource management algorithms on a variety of Grid topologies.

Furthermore, to cope with the advent of a new generation of service based Grids, research into novel Grid service management architectures (and accompanying management algorithms) is required in order to be able to meet the necessary QoS requirements.

1.3 Main research contributions

In a first stadium, we develop a Grid resource monitoring platform tuned to scalability, performance and easy extendibility and compare it (with positive results) with notable existing Grid monitoring platforms. Inspecting monitored Grid status information, we notice that the state of the network links interconnecting the various Grid resources has a big impact on the overall Grid job throughput and usage efficiency. In order to allow research into new management and scheduling algorithms that incorporate network resource state information to produce better management/scheduling decisions, we develop the NSGrid simulator (based on the ns-2 network simulator [23]). NSGrid is capable of accurately simulating the interconnecting Grid network links and offers advanced models for the different Grid resources (computational, storage, data and network resources), jobs and management components. While the work in this thesis focuses on allowing accurate and advanced Grid simulation, ir. Pieter Thysebaert simultaneously developed multiple novel network aware Grid scheduling and dimensioning algorithms using NSGrid.

As services are becoming more and more important in the context of Grids, we develop and (using NSGrid) evaluate a distributed Grid service management architecture incorporating resource-to-service partitioning of Grid resources in multiple "Virtual Private Grids" (VPGs). Our results show that this partitioning of Grid resources in multiple service-dedicated resource pools, together with the dynamic deployment of VPG management components (automatically) tuned to the service class they are responsible for, can improve Grid service QoS support, resource

efficiency and overall scalability.

A recent development is the incorporation of Grid technology in the audio/visual production industry (i.e. radio and television production companies). In this thesis we present architecture, requirements and characteristics of these MediaGrids. Furthermore, we incorporate our simulation, scheduling and service management research into the MediaGrid-case and present the results.

1.4 Outline

This thesis continues as follows: in chapter 2 we discuss the developed Grid monitoring framework and compare its functionality and performance to notable existing Grid monitoring architectures. Chapter 3 covers NSGrid, a network-centered Grid simulation environment built on top of ns-2, and provides details about Grid, resource and job models that have been implemented along with an overview of the various management components and their functionality. We continue in chapter 4 by presenting a distributed Grid service monitoring/management framework, allowing for automated resource-to-service partitioning. Different partitioning algorithms (based on Genetic Algorithm heuristics) are detailed and their performance is evaluated using NSGrid. We conclude this thesis in chapter 5 with the introduction of Grid computing in the audio/visual production industry, studying requirements, characteristics and architecture of MediaGrids.

1.5 Publications

1.5.1 Publications in international journals

- F. De Turck, S. Vanhastel, **B. Volckaert**, P. Demeester, *Generic middleware-based platform for scalable cluster computing*, Elsevier Journal on Future Generation Computer Systems, 18:549-560, 2002.

- P. Thysebaert, **B. Volckaert**, F. De Turck, B. Dhoedt, P. Demeester, *Evaluation of Grid scheduling strategies through NSGrid: a network-aware Grid simulator*, published in Neural, Parallel & Scientific Computations, Special Issue on Grid Computing, Dynamic Publishers Atlanta, Editors H.R. Arabnia, G.A. Gravvanis, M.P. Bekakos, 12:353-378, 2004.

- **B. Volckaert**, P. Thysebaert, M. De Leenheer, F. De Turck, B. Dhoedt, P. Demeester, *Grid computing: the next network challenge!*, published in The Journal of The Communications Network, Proceedings of FITCE 2004, 43rd European Telecommunications Congress, 3:159-165, 2004.

- **B. Volckaert**, P. Thysebaert, F. De Turck, B. Dhoedt, P. Demeester, *Application-specific hints in reconfigurable Grid scheduling algortihms*, published in Lecture Notes in Computer Science, Proceedings of ICCS 2004, Springer-Verlag Berlin Heidelberg, Krakow, LNCS 3038:149-157, 2004.

- P. Thysebaert, **B. Volckaert**, M. De Leenheer, F. De Turck, B. Dhoedt, P. Demeester, *Resource partitioning algorithms in a programmable service Grid architecture*, published in Lecture Notes in Computer Science, Proceedings of the 5th Intern. Conf. on Computational Science ICCS 2005, Atlanta, LNCS 3516:250-258, 2005.

- P. Thysebaert, M. De Leenheer, **B. Volckaert**, B. Dhoedt, P. Demeester, *Scalable Dimensioning of Optical Transport Networks for Grid Excess Load Handling*, accepted for publication in Photonic Network Communications (PNC).

- M. De Leenheer, P. Thysebaert, **B. Volckaert**, F. De Turck, B. Dhoedt, P. Demeester, D. Simeonidou, R. Nejabati, G. Zervas, D. Klonidis, M. J. OMahony, *A View on Enabling Consumer Oriented Grids through Optical Burst Switching*, accepted for publication in IEEE Communications Magazine, 2006.

- **B. Volckaert**, P. Thysebaert, M. De Leenheer, F. De Turck, B. Dhoedt, P. Demeester, *Flexible Grid service management through resource partitioning*, accepted for publication in the Journal of Supercomputing.

- **B. Volckaert**, T. Wauters, J. Baert, M. De Leenheer, P. Thysebaert, F. De Turck, B. Dhoedt, P. Demeester, *Design of a MediaGrid framework and simulation of workflows for collaborative audiovisual organizations*, submitted to the Journal of Computer Communications.

1.5.2 Chapters in international publications

- **B. Volckaert**, P. Thysebaert, M. De Leenheer, F. De Turck, B. Dhoedt, P. Demeester, *Network and Service Aware Grid Resource Assignment*, to be published as a chapter in Grid Technologies: Emerging from Distributed Architectures to Virtual Organizations, Editors: M.P. Bekakos, G.A. Gravvanis and H.R. Arabnia, WIT Press.

1.5.3 Publications in international conferences

- S. Vanhastel, P. Thysebaert, F. De Turck, **B. Volckaert**, P. Demeester, B. Dhoedt, *Service brokering in an enhanced grid environment*, published in

Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02), Las Vegas, 2:712-718, 2002.

- F. De Turck, S. Vanhastel, P. Thysebaert, **B. Volckaert**, P. Demeester, B. Dhoedt, *Design of a middleware-based cluster management platform with task management and migration*, published in 2002 IEEE International Conference on Cluster Computing and the Grid, Chicago, pages 484-487, 2002.
- **B. Volckaert**, P. Thysebaert, F. De Turck, P. Demeester, B. Dhoedt, *Evaluation of grid scheduling strategies through a network-aware grid simulator*, published in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA'03, Las Vegas, 1:31-35, 2003.
- P. Thysebaert, **B. Volckaert**, M. De Leenheer, E. Van Breusegem, F. De Turck, B. Dhoedt, D. Simeonidou, M.J. O'Mahony, R. Nejabati, A. Tzanakai, I. Tomk, *Towards consumer-oriented photonic grids*, published and presented at Workshop on Optical Networking for Grid Services at ECOC2004, Stockholm, 2004.
- **B. Volckaert**, P. Thysebaert, M. De Leenheer, F. De Turck, B. Dhoedt, P. Demeester, *On the use of NSGrid for accurate grid schedule evaluation*, published in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA'04, Las Vegas, 1:200-206, 2004.
- **B. Volckaert**, P. Thysebaert, M. De Leenheer, F. De Turck, B. Dhoedt, P. Demeester, *Network aware scheduling in grids*, published in Proceedings of NOC2004, 9th European Conference on Networks & Optical Communications, Eindhoven, pages 311-318, 2004.
- P. Thysebaert, **B. Volckaert**, F. De Turck, B. Dhoedt, P. Demeester, *Network aspects of grid scheduling algorithms*, published in Proceedings of the ISCA 17th International Conference on Parallel and Distributed Computing Systems, San Francisco, pages 91-97, 2004.
- M. De Leenheer, P. Thysebaert, **B. Volckaert**, F. De Turck, B. Dhoedt, P. Demeester, *Evaluation of a job admission algorithm for bandwidth constrained grids*, published in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA'04, Las Vegas, 2:591-594, 2004.
- M. De Leenheer, E. Van Breusegem, P. Thysebaert, **B. Volckaert**, F. De Turck, B. Dhoedt, P. Demeester, D. Simeonidou, M.J. O'Mahony, R. Nejabati, A. Tza, *An OBS-based grid architecture*, published in 2004 IEEE

Globecom Telecommunications Conference Workshops, Dallas, pages 390-394, 2004.

- S. De Smet, P. Thysebaert, **B. Volckaert**, M. De Leenheer, D. De Winter, F. De Turck, B. Dhoedt, P. Demeester, *A performance oriented grid monitoring architecture*, published in Proceedings of the 2nd IEEE Workshop on End-to-End Monitoring Technics and Sercives (E2EMON), Monitoring Emerging Network Services, San Diego, pages 23-28, 2004.

- **B. Volckaert**, P. Thysebaert, M. De Leenheer, F. De Turck, B. Dhoedt, P. Demeester, *A Distributed Resource and Network Partitioning Architecture for Service Grids*, published in Proceedings of (on CD-ROM) the 5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid05), Cardiff, 2005.

- F. Farahmand, M. De Leenheer, P. Thysebaert, **B. Volckaert**, F. De Turck, B. Dhoedt, P. Demeester, J.P. Jue, *A multi-layered approach to optical burst-switched based grids*, published in Proceedings (on CD-ROM) of Workshop on Optical Burst/packet Switching (WOBS2005), 2nd International Conference on Broadnet Net, Boston, pages 127-134, 2005.

- M. De Leenheer, F. Farahmand, P. Thysebaert, **B. Volckaert**, F. De Turck, B. Dhoedt, P. Demeester, J. Jue, *Anycast routing in optical burst switched grid networks*, published in Proceedings of ECOC2005, 31st European Conference on Optical Communications, Glasgow, 3:699-702, 2005.

- **B. Volckaert**, P. Thysebaert, M. De Leenheer, F. De Turck, B. Dhoedt, P. Demeester, *A scalable and performant grid monitoring and information framework*, published in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '05, Las Vegas, 2005.

- P. Thysebaert, M. De Leenheer, **B. Volckaert**, F. De Turck, B. Dhoedt, P. Demeester, *Using Divisible Load Theory to Dimension Optical Transport Networks for Grid Excess Load Handling*, published in Proceedings of the International Conference on Networking and Services (ICNS), Papeete, French Polynesia, 2005.

- J. Baert, M. De Leenheer, **B. Volckaert**, T. Wauters, P. Thysebaert, F. De Turck, B. Dhoedt, P. Demeester, *Hybrid optical switching for data-intensive media grid applications*, published in Proceedings of the workshop on Design of Next Generation Optical Networks: from the Physical up to the Network Level Perspective, Gent, 2006.

- M. De Leenheer, F. Farahmand, K. Lu, T. Zhang, P. Thysebaert, **B. Volckaert**, F. De Turck, B. Dhoedt, P. Demeester, J. Jue, *Anycast Algorithms Supporting Optical Burst Switched Grid Networks*, accepted for publication in the proceedings of the International Conference on Networking and Services (ICNS), Silicon Valley, USA, 2006.

1.5.4 Publications in national conferences

- P. Thysebaert, **B. Volckaert**, F. De Turck, S. Vanhastel, P. Demeester, *Management of Network Resources in a Grid-Computing Environment*, published in 2nd FTW PHD Symposium, Interactive poster session, Ghent, paper nr. 72, 2001.
- **B. Volckaert**, P. Thysebaert, F. De Turck, B. Dhoedt, P. Demeester, *A generic grid simulator for evaluating network-aware grid scheduling algorithms*, published in 3rd FTW PHD Symposium, Interactive poster session, Ghent, paper nr. 21, 2002.
- M. De Leenheer, E. Van Breusegem, J. Cheyns, P. Thysebaert, **B. Volckaert**, F. De Turck, B. Dhoedt, P. Demeester, *Optical burst switching for consumer grids*, published in 5th FTW PHD Symposium, Interactive poster session, Ghent, paper nr. 102, 2004.
- P. Thysebaert, **B. Volckaert**, F. De Turck, B. Dhoedt, P. Demeester, *Grid scheduling and dimensioning using divisible load theory*, published in 5th FTW PHD Symposium, Interactive poster session, Ghent, paper nr. 123, 2004.

References

[1] *Distributed.Net*. http://www.distributed.net/.

[2] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. *SETI@home: An Experiment in Public-Resource Computing*. Communications of the ACM, 45:56–61, 2002.

[3] *IBM ASCI White supercomputer*. http://www-03.ibm.com/servers/eserver/pseries/hardware/largescale/supercomputers/asciwhite/.

[4] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[5] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure 2nd Edition*. Morgan Kaufmann, 2003.

[6] *Global Grid Forum*. http://www.gridforum.org/.

[7] *Internet Engineering Task Force*. http://www.ietf.org/.

[8] *Global Grid Forum Open Grid Services Architecture working group*. https://forge.gridforum.org/projects/ogsa-wg.

[9] *Global Grid Forum: Grid High-Performance Networking research group*. https://forge.gridforum.org/projects/ghpn-rg.

[10] *Global Grid Forum: Grid Scheduling Architecture research group*. https://forge.gridforum.org/projects/gsa-rg.

[11] *Global Grid Forum: Job Submission Description Language working group*. https://forge.gridforum.org/projects/jsdl-wg/.

[12] *NEESgrid*. http://it.nees.org/.

[13] *Berkeley Open Infrastructure for Network Computing*. http://boinc.berkeley.edu/.

[14] Stefan M. Larson, Christopher D. Snow, Michael R. Shirts, and Vijay S. Pande. *Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology*. Computational Genomics, 2002.

[15] *FightAIDS@home*. http://fightaidsathome.scripps.edu/.

[16] *LHC Computing Grid project*. http://lcg.web.cern.ch/LCG.

[17] *CERN*. http://www.cern.ch/.

[18] K. Krauter, R. Buyya, and M. Maheswaran. *A Taxonomy and Survey of Grid Resource Management Systems*. International Journal of Software: Practice and Experience (SPE), 32:135–164, 2002.

[19] *Belnet Grid Initiative*. http://www.begrid.be/.

[20] *Enabling Grids for E-Science in Europe*. http://egee-intranet.web.cern.ch.

[21] *SpecInt2000 benchmark*. http://www.spec.org/cpu2000/.

[22] *The TeraGrid project*. http://www.teragrid.org/.

[23] *The Network Simulator - NS2*. http://www.isi.edu/nsnam/ns.

2
Grid Monitoring

2.1 Introduction

Grids provide a uniform interface to a collection of heterogeneous, geographically distributed resources. Each and every resource has its own specific static properties (e.g. processor speed, total amount of memory, total amount of disk space, networking capabilities) and dynamic status information (e.g. processor usage, available memory, available disk space, network usage). Furthermore, these resources are dynamic in nature: resources can join/part from the Grid, hardware failures can occur, etc. In order to make intelligent Grid resource management and scheduling decisions, accurate resource property and state information is required. In a distributed computing environment, one of the key components necessary to be able to perform effective job scheduling, resource management, fault detection, performance analysis and tuning is on one hand an information repository storing static resource properties and on the other a monitoring service, capable of measuring resource status and predicting future resource state. Monitored computational, storage, data and network resource status can also be used to construct realistic Grid topologies for simulation purposes (see NSGrid in chapter 3). The requirements for a framework delivering information and monitoring services are in no specific order: efficiency, accuracy, non-intrusiveness, scalability, portability and extensibility.

In this chapter, we present a Grid information and monitoring service facilitating a constraint-based resource selection mechanism for use by different Grid

management components (scheduler, service manager, resource broker, etc.). Static resource properties are stored in a distributed directory service, while resource status information can be fetched from a highly extensible yet performance oriented Grid monitoring platform, developed according to the Global Grid Forum's Grid Monitoring Architecture (GMA [1]) specifications. Important features include configurable caching mechanisms, non-intrusiveness, support for third party sensor plugins and an intuitive Graphical User Interface. We discuss the technology decisions that were made when developing this platform, and compare its performance with the widely deployed Globus [2] Toolkit version 2 Monitoring and Discovery Service (MDS [3]) and Globus Toolkit version 3.2 Web Services Information Service (WS-IS [4]).

This chapter continues as follows: Section 2.2 gives an overview of important related work and highlights the differences with our framework. A high-level description of the constituent components is given in section 2.3, while technical decisions made during implementation are discussed in section 2.4. Our test results are presented in section 2.5 followed by concluding remarks in section 2.6.

2.2 Related work

The Grid Monitoring Architecture, as defined by the GGF [5], is a reference architecture for feasible Grid monitoring systems and consists of three important components: producers, consumers and a directory service (see figure 2.1). The directory service stores the location and type of information provided by the different producers, while consumers typically query the directory to find out which producers can provide their needed event data (after which they contact the producers directly). Producers in turn can receive their event data from a variety of providers (software/hardware sensors, whole monitoring systems, databases, etc.). The GMA does not specify the underlying data models or protocols that have to be used.

Multiple monitoring architectures for distributed computing systems have already been successfully deployed. Not all of them follow the guidelines set by the GMA (e.g. Condor's HawkEye [6, 7] does not support a decentralized architecture), and some are geared towards monitoring one single resource type (e.g. Remos [8, 9], focusing on network parameters). Below we present some notable Grid monitoring platforms with an architecture similar to our framework, and point out the differences with our implementation. For a complete overview of Grid monitoring tools we refer to [10].

GMA-compliant Grid monitoring systems include the European DataGrid's [11] Relational Grid Monitoring Architecture R-GMA [12, 13] and GridRM [14, 15]. R-GMA offers a combined monitoring and information system using a Relational Database Management System as directory service and monitoring data reposi-

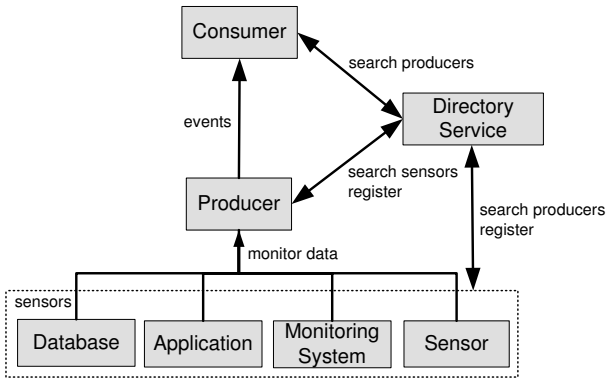


Figure 2.1: Grid Monitoring Architecture overview

tory. The implementation is based on Java [16] servlet technology (using the Tomcat [17] servlet container), trading performance for portability and limited software dependencies. GridRM is an open source two-layer Grid monitoring framework, the upper layer being structured according to the GMA. This upper layer connects the per-site monitoring systems in a scalable way. Like R-GMA, GridRM makes use of Java and SQL to query data producers. Currently, GridRM's directory service (containing info on the location of the different resource status providers) can be a bottleneck and/or single point of failure, but work is under way to remedy this problem.

Network Weather Service [18, 19] is an architecture for measuring the performance of distributed systems in processing intense environments. It can predict network and processing load in the near future based on monitored historical data. Measured data includes cpu load, packet round trip time, TCP connect/disconnect time and network bandwidth between two endpoints. The Network Weather Service is a robust and scalable system generating low computational and network overhead, but does not measure some important parameters (memory usage, swap, packets sent/received, etc.) and is hard to extend. It is also platform dependent as it relies on UNIX/Linux system tools for measuring cpu load. It is mostly used for its network forecasting capabilities.

Java Agents for Monitoring and Management (JAMM [20]) is a monitoring architecture fully implemented in Java. JAMM is mainly based on the GMA architecture and offers automated deployment of sensor agents on hosts from a central HTTP server. These sensors are actually wrappers for popular UNIX/Linux system utilities such as netstat, iostat and vmstat and are therefore badly deployable on other operating systems. JAMM does not offer support for application monitoring and performance analysis/forecasting.

MDS2 is the Globus Toolkit (version 2) Monitoring and Discovery Service [3],

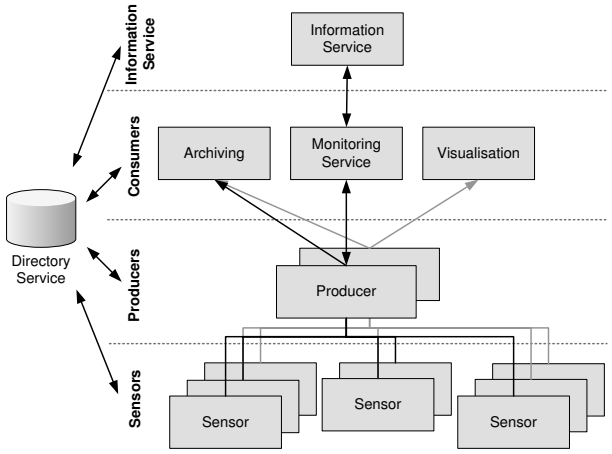


Figure 2.2: Information & Monitoring Framework components

and although MDS development was started before the GMA architectural reference appeared, it can still be seen as a GMA implementation. MDS2 only supports latest-state queries, making it mandatory for the consumers to actively retrieve status information from the GRIS (MDS2 component offering producer-like functionality). In addition, MDS2 does not offer visualization features. An extensive comparison of MDS2 against other monitoring frameworks has already been carried out in [21, 22]. It was shown that MDS2 outperforms (i.e. exhibits lower response times and better scalability) the other frameworks mentioned in most use cases. Therefore, we have only compared our platform's performance to that of MDS2 and its successor, the web services based Information Service (WS-IS [4]) from the Globus 3.2 Toolkit.

2.3 Information & Monitoring Framework components

A sample setup of the developed framework is shown in figure 2.2. Each component's function is detailed below.

2.3.1 Sensor

Every resource to be monitored has at least one sensor attached to it. Each sensor can monitor different load properties of a single resource by means of plugins (e.g. we have implemented a CPU plugin capable of monitoring CPU load, idle time and time spent executing system/user processes). The actual monitored values are communicated to one or more producers. This list of producers (and conversely,

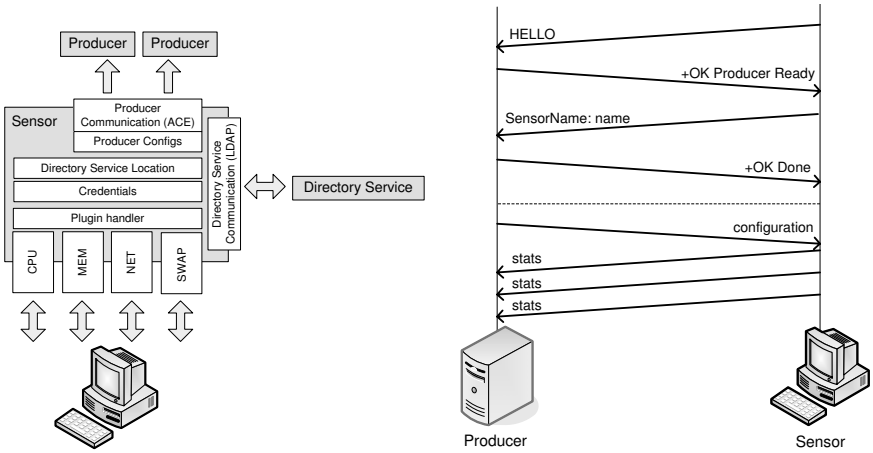
the list of sensors that is allowed to communicate with each producer) is found in the directory service (see figure 2.3(a)). The only configurable parameters of a sensor (read from a configuration file) are the location and authentication parameters to query the directory service and the plugins that need to be loaded.

As shown in figure 2.3(b), a sensor can register itself with one or more producers, by sending them a "*HELLO*" message. The producer will answer this with an "*OK producer ready*" message to acknowledge that it is on-line and ready. The sensor in turn will send its unique name to the producer, who is then capable of looking up the sensor's name in his configuration and can decide to either allow the sensor to sign in (return "*+OK, done*") or to reject the sensor login (return "*-ERR*"), preventing illegitimate sensors from sending (possibly tampered) monitoring data to the producer. Once this is done the registration connection is closed and the sensor is registered with the producer.

When a producer decides it wants to start receiving monitored data from one of its registered sensors, the producer can open a TCP connection with this particular sensor, sending the sensor a monitoring configuration message detailing from which plugins the producer would like to receive monitored data (together with the desired monitoring frequency of each plugin the producer is interested in). Once the configuration has been received by the sensor, it will start pushing the required monitoring values over the already open TCP connection, avoiding a connection setup overhead on every update.

The currently implemented sensor plugins can provide detailed status information on CPU, memory, swap and network usage. We give a brief overview of some of the most important resource characteristics that can be monitored:

1. *CPU Monitor*
    - CPU Total: percentage of time CPU was active during last measurement interval.
    - CPU Sys: percentage of time CPU was processing system tasks during last measurement interval.
    - CPU User: percentage of time CPU was processing user tasks during last measurement interval.
    - CPU Idle: percentage of time CPU was idle during last measurement interval.
2. *MEM Monitor*
    - MEM Total: total amount of memory installed in the system.
    - MEM Used: amount of memory in use.
    - MEM Free: amount of free memory.



(a) Sensor (b) Sensor-Producer protocol

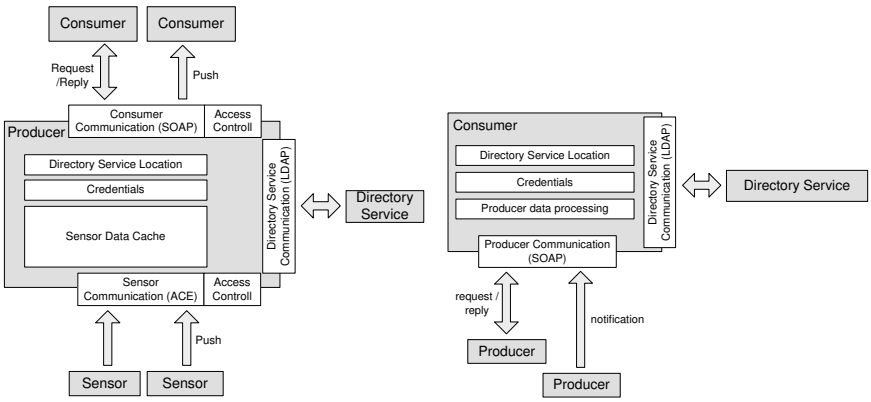
Figure 2.3: Sensor

- MEM User: amount of memory used by user processes.

3. *SWAP Monitor*
   - SWAP Total: total amount of swap space in the system.
   - SWAP Used: amount of swap space in use.
   - SWAP Free: amount of free swap space.
4. *NET Monitor*
   - Bytes in/out: total amount of received/sent bytes since the start of the network monitoring.
   - Errors in/out: total amount of packets with errors received/sent since the start of the network monitoring.
   - Bytes/s in/out: amount of received/sent bytes per second during last measurement interval.

2.3.2 Producer

Producers (see figure 2.4(a)) register themselves with the directory service and publish the type of information (aggregated from the sensors that report to the producer) they provide. This data can be queried by authorized consumers (using a request/reply model) or can be pushed to authorized consumers using a subscription/notification event-based model.



(a) Producer (b) Consumer

Figure 2.4: Producer and consumer

Each producer has its own cache, storing a configurable number of status updates from the different plugins of each registered sensor. Consumers can retrieve either the last known status update from a specific sensor plugin, or historic data from the producer's cache. Furthermore, producers provide a limited number of statistical operations (average, minimum, maximum, standard deviation, etc.) on cached data. Sensor failures (e.g. resources going off-line abruptly) can be detected and a failure notification will be sent to consumers who were interested in this sensor's data. Access control prevents unauthorized consumers from retrieving data from a producer and prevents unauthorized sensors from pushing monitored data.

2.3.3 Directory service

The directory service (see figure 2.5) contains information on the registered producers (and their respective offered status information), the producer-sensor mappings and producer access control lists. It is queried by producers and sensors to retrieve these mappings, and by consumers and the monitoring service to find a set of providers matching a given criterium. On top of this, the directory service also stores static resource properties (e.g. installed software/hardware) to be used by the information services.

2.3.4 Consumer

Consumers (see figure 2.4(b)) query the directory service to find producers capable of delivering the desired monitoring data. They then proceed by directly contacting these producers, either to retrieve data using a request/reply pattern or

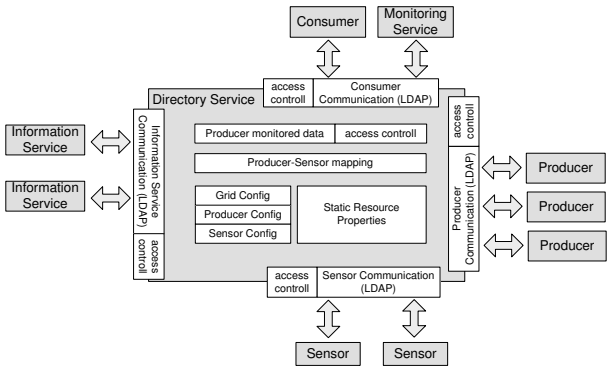


Figure 2.5: Directory service

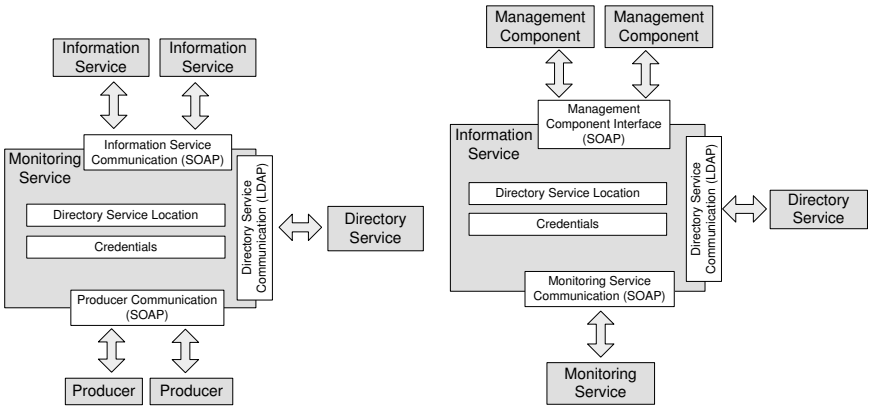
to register themselves in order to retrieve future data using an event-based subscription model. Several consumers have been implemented, the most important being an archiving consumer (storing data in a relational database and offering a GUI view of historical data), a real-time Java-based visualization agent (see figure 2.10) and a network status correlation agent supporting usage prediction (by passing monitored network status to the Network Weather Service) and network failure localization (by automated comparison of historical status information).

2.3.5 Monitoring service

The monitoring service is in effect an optimized *consumer*: its primary role is to retrieve dynamic status information (from the appropriate producers) pertaining to the resources presented to it by the information services. Producers capable of supplying the requested status data can be located through the directory service. The monitoring service only gathers the most recent value published by the producers, and communicates with the producers using a request-reply type transaction.

2.3.6 Information service

The information service provides Grid management components with a resource status/property match-making functionality: management components can contact this service by submitting resource queries, containing one or more *(attribute, relational operator, value)* triplets limiting the resulting resource set (e.g. available memory $>=$ 256 MB and architecture $==$ multiprocessor). The information service will first query the directory service for resources adhering to the requested static resource property demands (e.g. installed software/hardware, installed memory). Once the resources complying to the static requirements of this request are known, the necessary dynamic status information for these resources will be sup-



(a) Monitoring service (b) Information service

Figure 2.6: Monitoring and information service

plied by the monitoring service. Finally, the resulting resource set (to be sent back to the management components) will be further reduced by removing resources that do not adhere to the requested dynamic resource demands (e.g. available memory, remaining storage space).

Figure 2.7 shows the message passing order between the different components when a management component submits a resource query.

1. A management component sends a query to the information service for resources adhering to static resource properties (installed CPU, total memory, installed software/hardware, etc.) and dynamic resource state (available memory, available storage space, etc.).

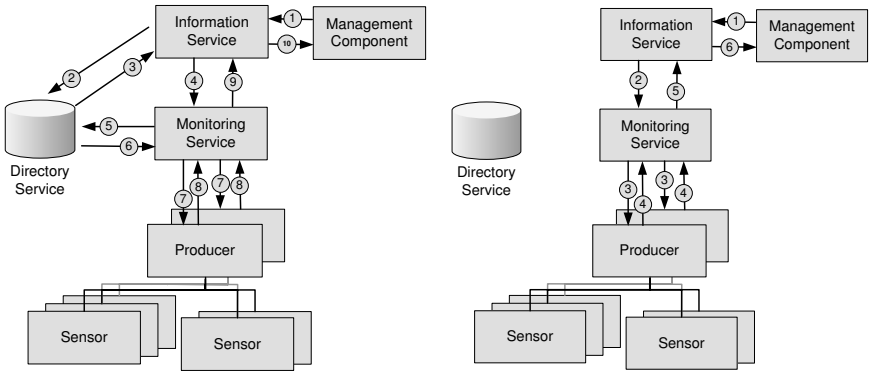
2. The information service sends a query to the directory service for resources adhering to the static resource property requirements.

3. The directory service supplies a subset of its registered resources adhering to the requested static resource properties.

4. The information service queries the monitoring service for monitored resource state information of the resources supplied by the directory service in step 3.

5. The monitoring service queries the directory service for producers capable of providing the required resource state info.

6. The directory service supplies a list of producers that receive monitoring data from sensors installed on the resources the monitoring service is interested in.



(a) Message transition (non-cached) (b) Message transition (cached)

Figure 2.7: Resource query message passing order

7. The monitoring service asks the different producers for monitored resource state information.

8. Producers provide monitored resource state info.

9. The monitoring service sends the information service the required dynamic resource state information.

10. The information service answers the management component's query with the resources that adhere to both the static resource properties and the dynamic resource state information.

Note that step $2, 3, 5$ and 6 are not necessarily mandatory (as shown in figure 2.7(b)), as both the information service and the monitoring service have been equipped with a directory service caching mechanism (storing static resource properties and producer offerings respectively). Producers do not need to contact their sensors, as they consult their sensor update cache (which is filled by the sensors' status update push).

2.4 Technology analysis and implementation

The Information & Monitoring Framework presented here was designed to achieve good performance while maintaining a high level of portability. Our performance requirement has driven us to the use of the C++ language [23] in the implementation of the different components. While C/C++ programming languages provide good performance and offer more control over memory allocation/deallocation (than for instance the Java programming language), the C++ standard library

does not offer cross-platform solutions for various vital application patterns such as networked communication and multiple threads of execution. Therefore, the need arises to use a portable and well performing C++ middleware platform for implementations of these high-level features. In our framework, we have decided to use the Adaptive Communication Environment (ACE [25, 26]). It offers cross-platform multithreading, and its *reactor* concept allows for easy implementation of event-based (including network events) applications. Furthermore, we make use of the *Acceptor/Connector* pattern offered by ACE to open networked communication channels between producers and sensors.

Whenever a sensor needs to send monitored data to the producers they are registered with, the data is sent in CORBA Common Data Representation (CDR) format [27], offering a portable, network optimized way of communicating. The resource monitoring data gathered by the sensors is obtained through the GTop library [28], a portable C/C++ library offering access to performance values related to system resources. Each resource is monitored by a sensor plug-in, which is essentially a shared library. The plug-in approach is enabled by the fact that ACE features cross-platform dynamic loading of shared libraries. The methods of the abstract MonitorPlugin class which every plugin must implement is shown in listing 2.1.

Listing 2.1: abstract class MonitorPlugin

```
class MonitorPlugin: public ACE_Task<ACE_MT_SYNCH>{
public:
  MonitorPlugin();
  virtual ~MonitorPlugin();
  virtual int receive_configdata(char* data)=0;
  virtual int convertToChar(char* buffer)=0;
  virtual int convertFromChar(char* data)=0;
  virtual NameValVector getValues()=0;
  virtual bool needActivate()=0;
  virtual PluginCapabilities getCapabilities()=0;
  virtual int WriteToCDR(ACE_OutputCDR &cdr)=0;
  virtual int ReadFromCDR(ACE_InputCDR &cdr)=0;
};
```

In what follows we briefly given an overview of the supported methods in this abstract class:

- *int receive_configdata(char* data)*: enables the plugin to receive additional configuration information from a producer or the directory service (e.g. frequency of monitoring).

- *int convert(To/From)Char(char* data)*: used mainly for debugging internal plugin data.

- *NameValVector getValues()*: constructs a vector of (name,value) pairs with name and value both string representations of the name of the measured item and the actual monitored data (this string representation enables us to generically store different data types: the actual data type used by the monitored value can be retrieved by querying the plugin's capabilities).

- *bool needActivate()*: a sensor uses this function to check if it needs to start a separate thread for the plugin. For small calculations no such thread is required, but if the plugin needs to perform heavy processing on the monitored data, a separate thread can prevent the sensor from blocking while retrieving the plugin's monitoring data.

- *PluginCapabilities getCapabilities()*: in order to support generic plugins, this function enables retrieval of metadata information regarding the items that can be monitored using this plugin. It returns name, description and data type for each item that the plugin is capable of measuring.

- *int (ReadFrom/WriteTo)CDR(...)*: The Adaptive Communication Environment (ACE [25]) offers ACE_CDR classes for optimised (w.r.t. data size) and machine independent storing of primitive data types (and arrays of the aforementioned types), useful for transporting over a network.

Our directory service is essentially a decentralized LDAP [29] directory server. While the expressive power of LDAP queries does not match that of e.g. an SQL query over a relational database [30], LDAP is performance-tuned for write-once, read-many operations (which is ideal in our case since directory service queries are much more frequent than static resource property changes or monitoring architecture component deployment changes). Entries in this LDAP directory service are stored in a Directory Information Tree (DIT) based on their Distinguished Name (DN). Each entry is uniquely identifiable by its DN (comparable to a primary key in a relational database system). In figure 2.8 we show the DIT for the Grid monitoring framework (along with some sample attributes). One or more Grid entries can be stored in the directory service, with each Grid entry consisting of a set of producer entries. Each producer entry is parent to one or more sensor entries, which in turn are parenting one or more "MeasurementModule" entries. Host information is stored in a separate entry and can be linked to by producer and sensor entries, as we allow multiple producers and/or sensors to be deployed on a single host. An example of a unique DN in figure 2.8 is *SensorName='Sensor1',ProducerName='Producer1',GridName='testGrid1',cn='test'*.

In order to ease directory service management, an easy-to-use Graphical User Interface was developed (as seen in figure 2.9).

Producers provide a Web Service Description Language (WSDL [31]) interface through which they are contacted by consumers using SOAP [32]. SOAP is

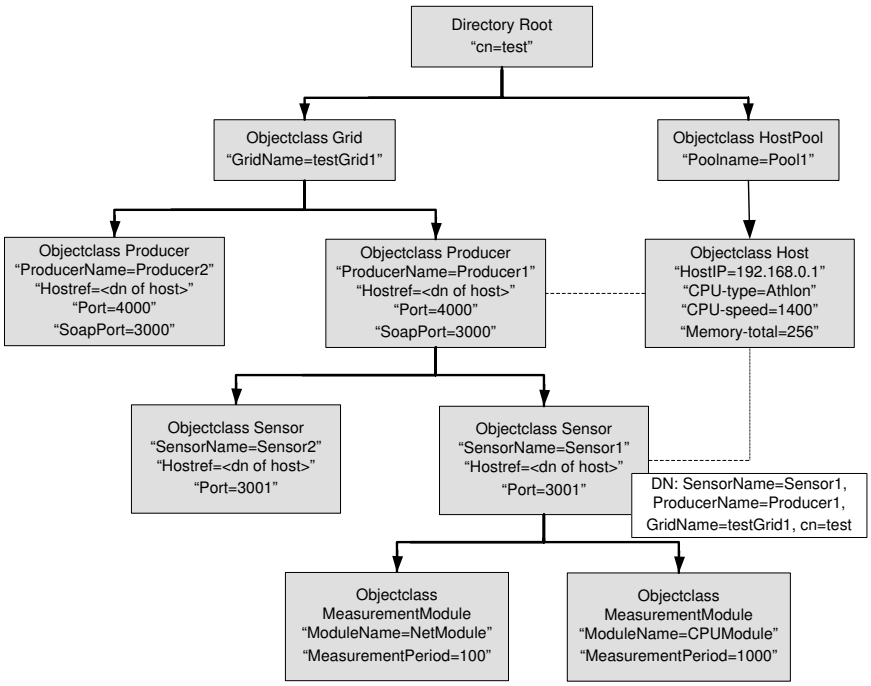


Figure 2.8: Sample LDAP directory service information tree

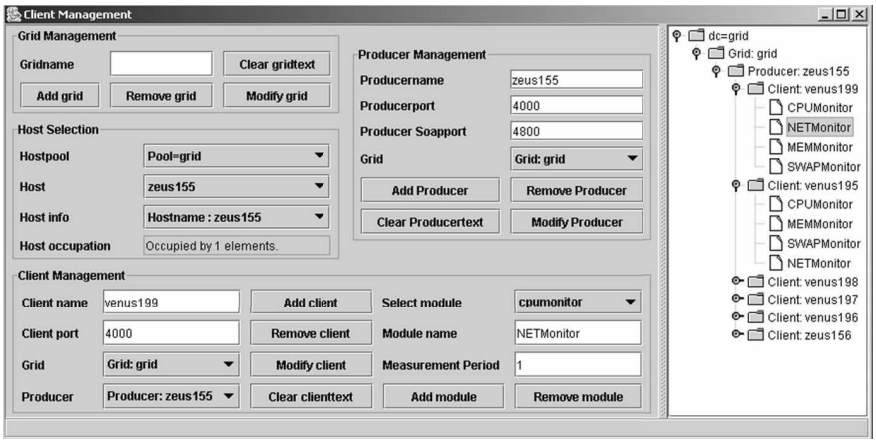


Figure 2.9: Directory service management GUI

from/to	Sensor	Producer	Consumer	Dir. Service	Inf. Service
Sensor	/	ACE	/	LDAP	/
Producer	ACE	/	SOAP	LDAP	/
Consumer	/	SOAP	/	LDAP	SOAP
Dir. Service	LDAP	LDAP	LDAP	/	LDAP
Inf. Service	/	/	SOAP	LDAP	/

Table 2.1: Communication technologies

an XML-based RPC protocol which can be transferred over HTTP; as such, the use of SOAP allows for the easy integration of our monitoring architecture in a web services-based Grid environment. In our implementation we used gSOAP as reported on in [33].

The information service also provides a WSDL interface to the different Grid management components. In listing 2.2 we provide a sample query, in which a scheduler asks the information service for resources with at least a 500Mhz processor (a static resource property) and 256MB of available memory space (a dynamic and monitored resource state), with query results ordered according to the amount of memory available. Listing 2.3 provides a sample answer to the previous query, with the information service providing the IP addresses of 3 resources adhering to the scheduler's requirements.

The information service contacts the directory server using native LDAP commands (to retrieve resource sets that comply to static resource property demands) and offers a WSDL interface towards the monitoring service component.

An overview of the communication methods used between the various components of our Grid Monitoring Architecture is given in table 2.1. It should be noted that SSL encryption [34] is possible for both SOAP-over-HTTP and LDAP communication. This allows access control through the use of user and server certificates. The data updates between sensors and producers can be secured by enabling an SSL socket adapter in these components.

A realtime data visualization consumer was implemented in Java and offers GUI visualization of select monitored data (see figure 2.10) and can print data to HTML files in a presentable manner (see figure 2.11). An archiving consumer was also implemented in Java, capable of storing monitored data in a PostgreSQL [35] relational database (using a JDBC [36] connection) and offering GUI visualization of archived data.

In order to provide network resource state forecasting functionality, a "ping" and "trace" sensor plugin was developed, capable of measuring round trip time (RTT), packet loss and routing details between two hosts. These sensors send monitored network data to interested producers, who in turn can push this data to a specialized network forecasting consumer, storing received data, along with

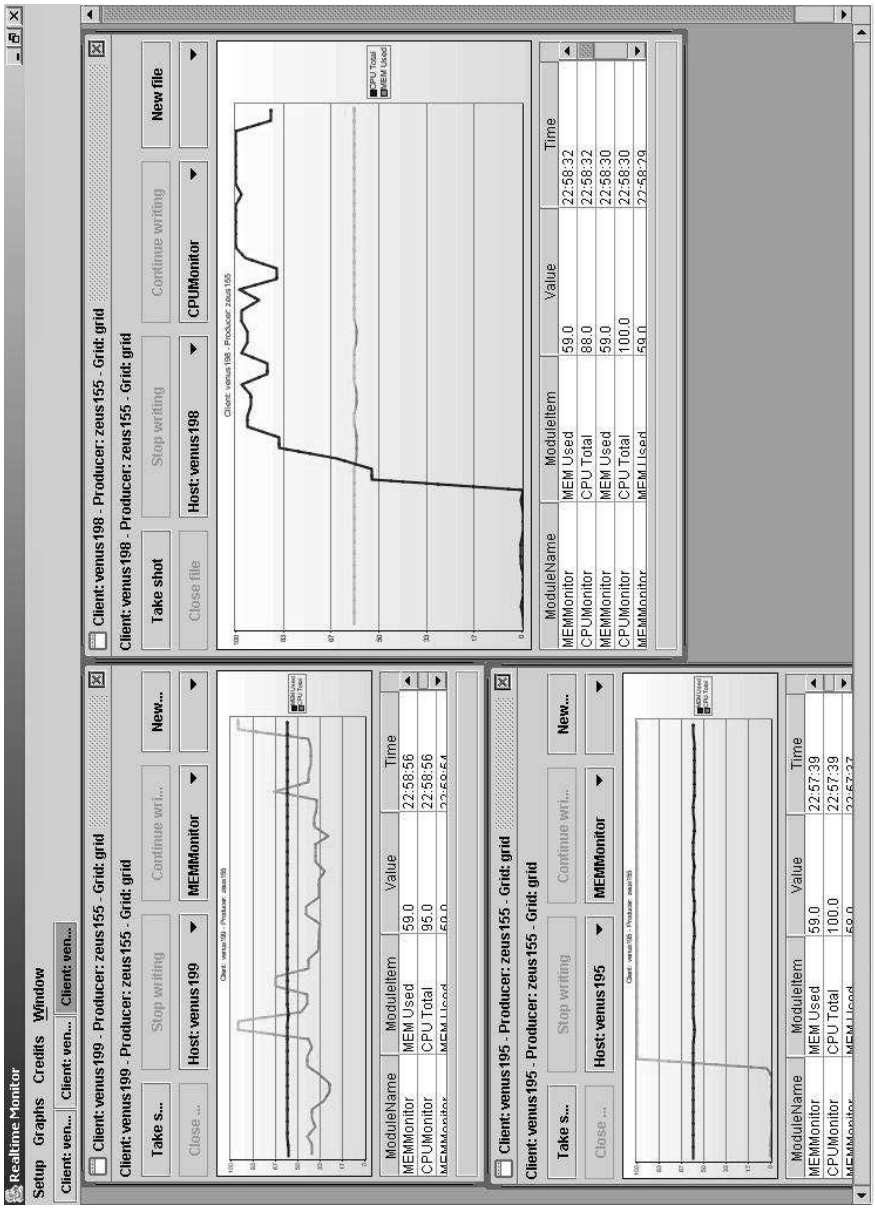


Figure 2.10: Realtime consumer GUI

Listing 2.2: Scheduler-information service query example

```
<?xml version="1.0" encoding="UTF−8"?>
<SOAP−ENV:Envelope xmlns:SOAP−ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:SOAP−ENC="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:ms="urn:monitoringserversoap"
 xmlns:is="urn:informationserversoap">

 <SOAP−ENV:Body id="_0">
  <getRestrictedList xmlns="urn:informationserversoap">
   <q xsi:type="is:Query">
    <numberOfMachines>3</numberOfMachines>
    <constraints xsi:type="SOAP−ENC:Array" SOAP−ENC:arrayType="is:Constraint[2]">
     <item xsi:type="is:Constraint">
      <constraintItem>CPU_Speed</constraintItem>
      <constraintType>greater than or equal to</constraintType>
      <constraintValue>500</constraintValue>
     </item>
     <item xsi:type="is:Constraint">
      <constraintItem>MEM Free</constraintItem>
      <constraintType>greater than or equal to</constraintType>
      <constraintValue>256</constraintValue>
     </item>
    </constraints>
    <sortData xsi:type="SOAP−ENC:Array" SOAP−ENC:arrayType="xsd:string[2]">
     <item>MEM Free</item>
     <item>by greatest</item>
    </sortData>
   </q>
  </getRestrictedList>
 </SOAP−ENV:Body>
</SOAP−ENV:Envelope>
```

timestamps, in a PostgreSQL database. The stored data can then automatically be fed to the Network Weather Service's forecasting program, generating predictions about RTT and packet loss between hosts.

2.5 Results

2.5.1 Testbed setup

The testbed used in our performance comparison is depicted in figure 2.12 and summarized in table 2.2. Six machines (AMD Duron 750Mhz, 64MB RAM) have a sensor deployed on them (a single sensor can have multiple sensor plugins installed), and four other machines (Intel P4 3GHz, 1GB RAM) carry one producer each; two producers have two sensors registered with them, and the other two have one sensor registered. An OpenLDAP [37] directory server was deployed on a sep-

Listing 2.3: Information service answer example

```
<?xml version="1.0" encoding="UTF−8"?>
<SOAP−ENV:Envelope xmlns:SOAP−ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:SOAP−ENC="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:ms="urn:monitoringserversoap"
 xmlns:is="urn:informationserversoap">

 <SOAP−ENV:Body id="_0">
  <getRestrictedListResponse xmlns="urn:informationserversoap">
   <Result xsi:type="SOAP−ENC:Array" SOAP−ENC:arrayType="xsd:string[3]">
    <item>10.10.10.197</item>
    <item>10.10.10.198</item>
    <item>10.10.10.156</item>
   </Result>
  </getRestrictedListResponse>
 </SOAP−ENV:Body>
</SOAP−ENV:Envelope>
```

Measurements of the Client venus199 , Producer zeus155 , Grid grid

CPU: 550 Mhz - Memory: 64 MB - HOST: venus199 - HOSTPOOL: grid - HOSTIP: 10.10.10.199

Module: CPUMonitor - Item: CPU Total - Description: The total number of CPU cycles since boot
Module: MEMMonitor - Item: MEM Used - Description: The total Memory used

ModuleName	*ModuleItem*	*Measured Value*	*Time*
CPUMonitor	CPU Total	22.0	15:35:57
MEMMonitor	MEM Used	59.0	15:35:57
CPUMonitor	CPU Total	29.0	15:35:58
MEMMonitor	MEM Used	59.0	15:35:58
CPUMonitor	CPU Total	47.0	15:35:59
MEMMonitor	MEM Used	59.0	15:35:59
CPUMonitor	CPU Total	41.0	15:36:01
MEMMonitor	MEM Used	59.0	15:36:01

Figure 2.11: HTML output for measurements

arate machine featuring dual Xeon processors (2.8Ghz, 1GB RAM). Lastly, the consumers (implemented as concurrent threads) used in the tests are located on a second dual Xeon machine. All machines are interconnected through a 100Mbps switched Ethernet LAN (see figure 2.12); this setup allows us to evaluate intrusiveness and scalability of the different developed components without suffering significant network bottlenecks.

Globus MDS2 was deployed as follows: a Grid Resource Information Service/Grid Information Index Server (GRIS/GIIS) pair ran on the Intel P4 machines (instead of the producers), sensors were replaced with GRIS components whose monitoring data was cached by the Intel P4 GIIS. Our LDAP directory service was replaced by a GIIS (on the dual Xeon machines) connected to the lower level GIISs. Consumers in our MDS2 tests were spawned from the same machine as our

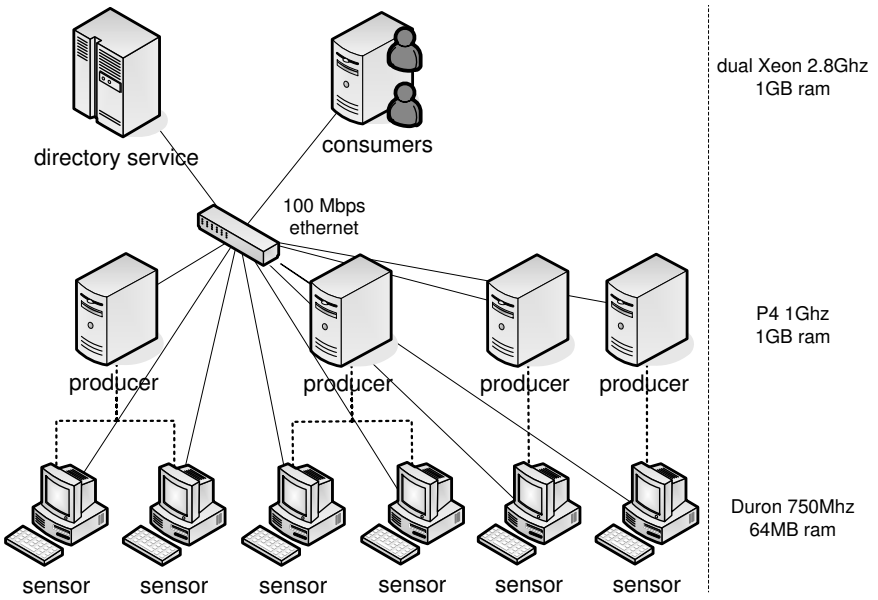


Figure 2.12: Testbed topology

	Sensor	**Producer**	**Consumer**	**Dir. Service**
Monitoring Framework	Sensor	Producer	Consumer	LDAP
MDS2	GRIS	GRIS/GIIS	Consumer	GIIS
GT 3.2	WS-IS	WS-IS	Consumer	/

Table 2.2: Testbed component setup

first tests.

The Globus Toolkit 3.2 Web Services based Information Services (GT 3.2 WS-IS) was deployed on the AMD Duron and Intel P4 using the default supplied OGSI-compliant container [38]; on the AMD Duron machines, the WS-IS was configured to submit its data to a Pentium 4 machine (which used to run a producer). Again, consumers were spawned from a dual Xeon machine. GT 3.2 WS-IS does not offer a component comparable to our directory service.

2.5.2 Metrics

Two metrics were used to evaluate component performance: *throughput* and *response time*. During a 10 minute period, "users" submitted blocking queries to the component under investigation, while waiting for 1 second between the moment an answer to the previous query has been received (the waiting time is used to

reinitialise the querying threads) and the time a new query is sent. The throughput was then taken to be the number of queries handled by the component per time unit; the response time is the average amount of time taken to process 1 user query. Note that, in practice, multiple components can take on the "user" role: the information service, monitoring service and producers all communicate with the directory service, whereas a producer is always contacted by consumers (in this context, the consumer is the monitoring service).

2.5.3 Information and Monitoring Service

From the previous section, it follows that the two most prevalent communication patterns are the ones where either the directory service or the producers are queried. We have therefore chosen to limit our performance and scalability tests to these patterns. SOAP communication between the Grid's management components and the information service on the one hand, and between the information service and the monitoring service on the other hand, consists of simple data transfers with no extra intelligence. Processing time spent in the monitoring service includes per-resource directory service lookups (if no producer location caching is used) and producer queries.

2.5.4 Producer intrusiveness

The network and computational intrusiveness (i.e. overhead) of our producer components is shown in figure 2.13 and figure 2.14, and compared to the load generated by the Globus MDS GRIS. The network traffic generated was monitored using the SCAMPI [39] multi-gigabit monitoring framework. The CPU load is the average (over the 600 second interval) one minute CPU load average, as measured by *uptime*.

The higher network load generated between our producer and the consumers (i.e. users in this test) stems from the use of the SOAP-over-HTTP XML-based communication mechanism (note that we did not enable *zlib* [40] compression). The web services approach used by Globus Toolkit 3.2 Web Services based Information Service (WS-IS) imposes a network load comparable to that of our architecture for low ($<$100) amounts of concurrent users. However, beyond 100 concurrent users, the GT3.2 WS-IS do not scale well (with time-outs and dropping of user requests), which explains their apparent low network intrusiveness. In analogy, the CPU load generated by the WS-IS seems to degrade with increasing number of concurrent users, but this is slightly deceptive: as the WS-IS does not scale beyond 100 users, it is no longer able to keep up with an appropriate pace of query response generation from this point on. A producer from our monitoring architecture imposes a processor load of less than 6 percent when 600 consumers are concurrently querying it, while the MDS GRIS needs 213 percent of processor

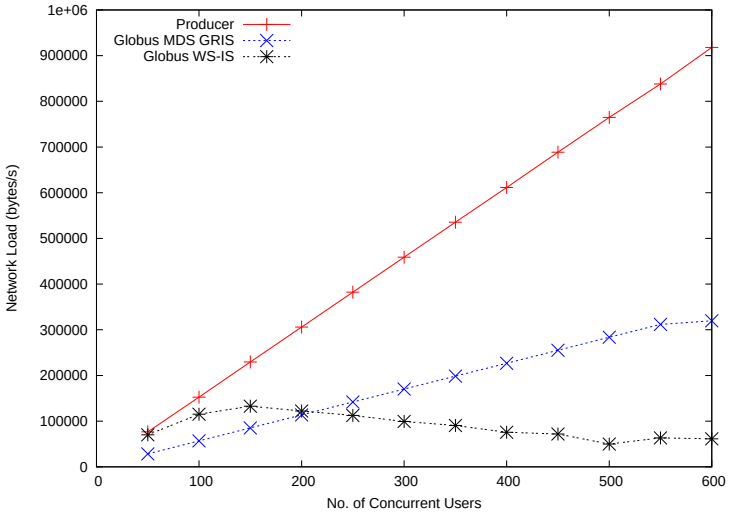


Figure 2.13: Producer vs. MDS GRIS vs. WS-IS network intrusiveness

time under the same user load conditions (note that a load value higher than 1 in figure 2.14 denotes insufficient processing power available to handle all jobs).

It should be noted that the WS-IS framework is the only monitoring and information framework in these tests which is completely web services based, trading performance for a standards based interface.

2.5.5 Producer scalability

In figure 2.15 and figure 2.16, we have compared producer scalability with increasing number of concurrent users for both our framework's producers, MDS GRIS and WS-IS components. Again, only cached data was requested from the MDS GRIS; due to the use of a push-model our framework's producers always contain up-to-date information, while the GRIS would have to invoke information providers to refresh its data. We measured only small differences between MDS GRIS and our producers (best visible on the response time graph). Beyond 550 concurrent users and using the given machines, MDS GIIS performance started to degrade. We also compared our producer scalability to the scalability of the GT 3.2 information service. Again, it is clear that GT 3.2 WS-IS does not scale well beyond 100 concurrent users (the GT 3.2 WS-IS results even forced us to use a logarithmic scale in figure 2.16, which shows that response times differ by as much as a factor of 100).

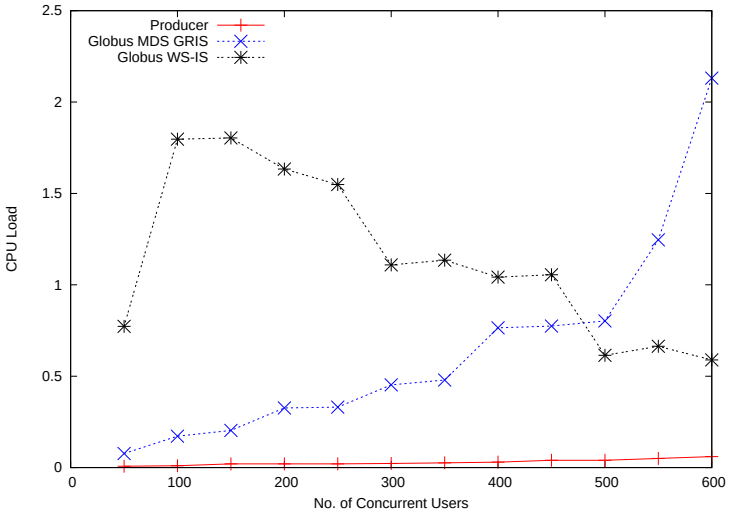


Figure 2.14: Producer vs. MDS GRIS vs. WS-IS cpu intrusiveness

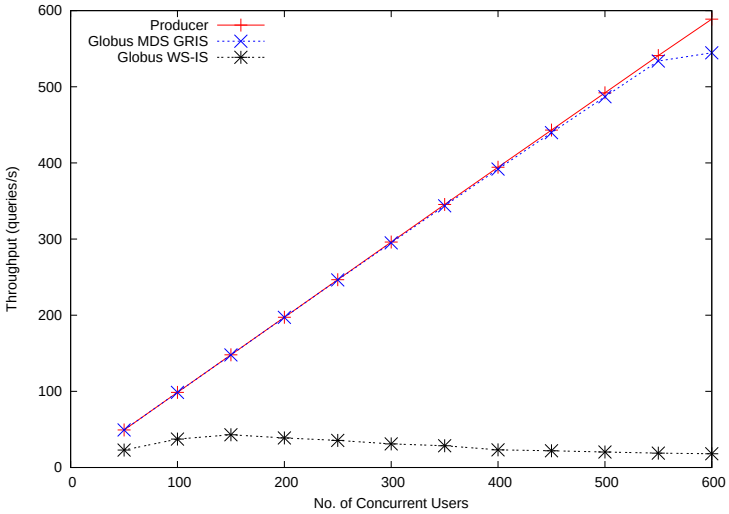


Figure 2.15: Producer vs. MDS GRIS vs. WS-IS throughput

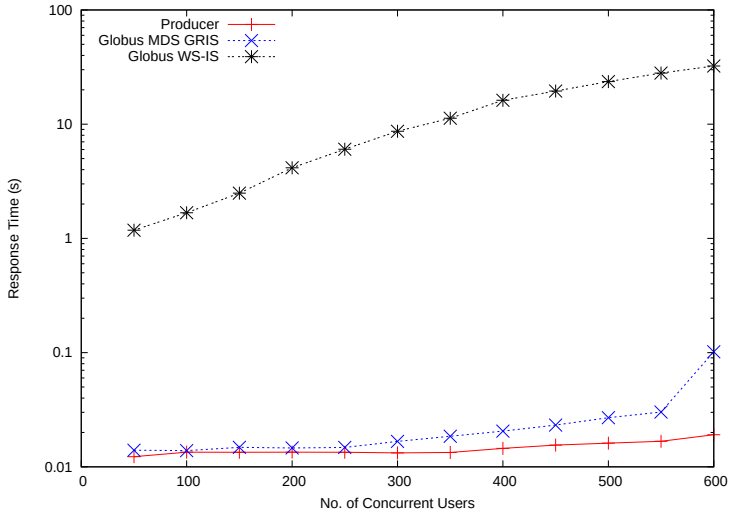


Figure 2.16: Producer vs. MDS GRIS vs. WS-IS response times

2.5.6 Directory Service Scalability

The throughput and response times for directory service queries were compared (figure 2.17 and figure 2.18) to those obtained for queries against the MDS GIIS (operating on cached data). Average response times are lower for our directory service; however, both our directory service and the MDS GIIS don't scale well beyond 450 concurrent users in this scenario on the given hardware. It should be noted however, that a GIIS typically contains more data (including cached monitoring data) than our directory service, which only stores configuration data, never monitoring data (this should be requested straight from a producer or from an archiving consumer). This led to a bigger result set when the GIIS was queried. In addition, our directory service is a plain OpenLDAP server, without module extensions. We chose not to show results for the GT 3.2 because of the absence of a dedicated component offering functionality which corresponds to our directory service or the MDS2.2 GIIS.

2.6 Conclusions

In this chapter a well-performing, scalable and portable information and monitoring framework was presented. Performance was obtained through the use of C++ as base implementation language, together with caching mechanisms at key locations (e.g. producers caching sensor data, eliminating the need for producers to contact sensors directly); portability then dictated the use of appropriate middle-

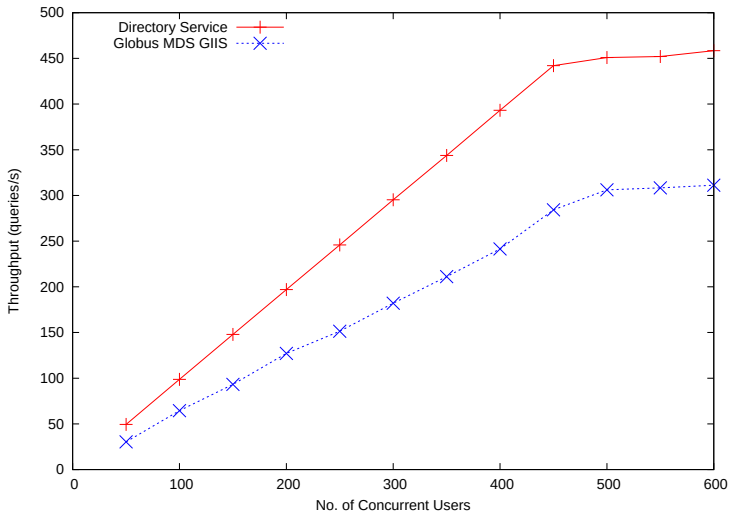


Figure 2.17: Directory Service vs. MDS GIIS throughput

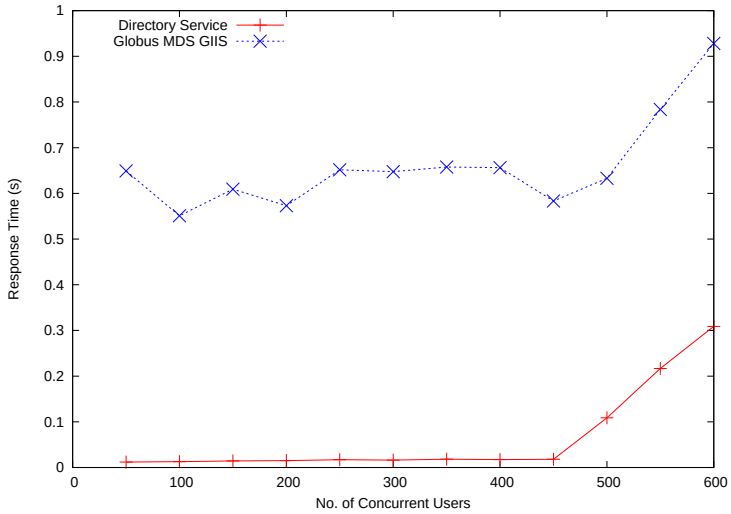


Figure 2.18: Directory Service vs. MDS GIIS response times

ware for which we chose the Adaptive Communication Environment (ACE) with Corba CDR data exchange, the GTop libraries for monitoring resource status and Java based consumers. Scalability was achieved by using a GMA-compliant architecture consisting of sensors, producers, consumers and a decentralized directory service. Multiple ready-to-use consumers (e.g. network usage prediction/failure detection, real-time visualization, archiving) have been implemented, and the information service offers a fast resource matchmaking portal for use by management components.

We compared our Information and Monitoring Framework to that of the Globus MDS2 system and its successor, the Globus Toolkit 3.2 web services based information service, in terms of performance and intrusiveness. Good results were measured in terms of query throughput and response times, both for our producers and directory service. Network intrusiveness was comparable to the GT 3.2 information service but worse than Globus MDS2, although this can be remedied by applying a compression algorithm to the SOAP-over-HTTP XML messages between producers and consumers. The computational intrusiveness of our framework was measured to be near negligible.

Monitored resource properties (e.g. typical processing capabilities, network bandwidths) and state information data (e.g. failure probabilities) can be used to construct realistic Grid topology descriptions for simulation purposes.

References

[1] B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, and R. Wolski. *A Grid Monitoring Architecture*. `http://www-didc.lbl.gov/GGF-PERF/GMA-WG/papers/GWD-GP-16-3.pdf`, 2002.

[2] *The Globus Alliance*. `http://www.globus.org/`.

[3] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. *Grid Information Services for Distributed Resource Sharing*. In Proc. of the 10th IEEE International Symposium on High-Performance Distributed Computing, 2001.

[4] *WS Information Services website*. `http://www-unix.globus.org/toolkit/docs/3.2/infosvcs/ws/key/index.html`.

[5] *Global Grid Forum*. `http://www.gridforum.org/`.

[6] Douglas Thain, Todd Tannenbaum, and Miron Livny. *Condor and the Grid in Grid Computing: Making The Global Infrastructure a Reality*. John Wiley, 2003.

[7] *HawkEye: A Monitoring and Management Tool for Distributed Systems*. `http://www.cs.wisc.edu/condor/hawkeye/`.

[8] Bruce Lowekamp, Nancy Miller, Thomas Gross, Peter Steenkiste, Jaspal Subhlok, and Dean Sutherland. *A resource query interface for network-aware applications*. Cluster Computing, 2(2):139–151, 1999.

[9] Bruce Lowekamp, Nancy Miller, Roger Karrer, Thomas Gross, and Peter Steenkiste. *Design Implementation and Evaluation of the Remos Network*. Journal of Grid Computing, 1:75–93, 2003.

[10] M. Gerndt, R. Wismuller, Z. Balaton, G. Gombas, P. Kacsuk, Zs. Nemeth, N. Podhorszki, H. L.Truong, T. Fahringer, M. Bubak, E. Laure, and T. Margalef. *Performance Tools for the Grid: State of the Art and Future*. Research Report Series, 30, 2004.

[11] *The DataGrid Project*. `http://eu-datagrid.web.cern.ch/eu-datagrid/`.

[12] A. Cooke, A.Gray, L. Ma, W. Nutt, J. Magowan, P. Taylor, R. Byrom, L. Field, S. Hicks, and J. Leake et al. *R-GMA: An Information Integration System for Grid Monitoring*. In Proc. of the 11th International Conference on Cooperative Information Systems, 2003.

[13] Andrew W. Cooke, Alasdair J. G. Gray, Werner Nutt, James Magowan, Manfred Oevers, Paul Taylor, Roney Cordenonsi, Rob Byrom, Linda Cornwall, Abdeslem Djaoui, Laurence Field, Steve Fisher, Steve Hicks, Jason Leake, Robin Middleton, Antony J. Wilson, Xiaomei Zhu, Norbert Podhorszki, Brian A. Coghlan, Stuart Kenny, David O'Callaghan, and John Ryan. *The Relational Grid Monitoring Architecture: Mediating Information about the Grid*. Journal of Grid Computing, 2:323–339, 2004.

[14] M.A. Baker and G. Smith. *GridRM: A Resource Monitoring Architecture for the Grid*. In Springer-Verlag, editor, Proc. of the 3rd International Workshop on Grid Computing, 2002.

[15] M.A. Baker and G. Smith. *GridRM: An Extensible Resource Monitoring System*. In proceedings of the IEEE Cluster Computing Conference, 2003.

[16] *Java*. `http://java.sun.com/`.

[17] *Apache Tomcat*. `http://tomcat.apache.org/`.

[18] R. Wolski, N. Spring, and Jim Hayes. *The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing*. Journal of Future Generation Computing Systems, 15(5-6), 1999.

[19] Rich Wolski. *Experiences with Predicting Resource Performance On-line in Computational Grid Settings*. ACM SIGMETRICS Performance Evaluation Review, 30:41–49, 2003.

[20] Brian Tierney, Brian Crowley, Dan Gunter, Mason Holding, Jason Lee, and Mary Thompson. *A Monitoring Sensor Management System for Grid Environments*. In Proc. of High Performance Distributed Computing' 00, 2000.

[21] X. Zhang, J.L. Freschl, and J. Schopf. *A Performance Study Of Monitoring and Information Services for Distributed Systems*. In Proc. of the 12th IEEE International Symposium on High-Performance Distributed Computing, 2003.

[22] X. Zhang and J. Schopf. *Performance Analysis of the Globus Toolkit Monitoring and Discovery Service, MDS2*. In Proceedings of the International Workshop on Middleware Performance (MP 2004), part of the 23rd International Performance Computing and Communications Workshop (IPCCC), 2004.

[23] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publication Company, 2000.

[24] Matthew Wilson. *Does C# measure up: Comparison with C, C++, D and Java*. Windows Developer Network `http://www.wdj.com/wdn/webextra/2003/0313/`, 2003.

[25] D.C. Schmidt and S.D. Huston. *C++ Network Programming: Mastering Complexity Using ACE and Patterns*. Addison-Wesley Longman, 2002.

[26] D.C. Schmidt and S.D. Huston. *C++ Network Programming: Systematic Reuse with ACE and Frameworks*. Addison-Wesley Longman, 2003.

[27] *Common Object Request Broker Architecture: Core Specification Version 3.0.3*. `http://www.omg.org/technology/documents/corba_spec_catalog.htm`, 2004.

[28] *LibGTop website*. `http://www.gnu.org/directory/libs/LibGTop.html`.

[29] Wahl M., T. Howes, and S. Kille. *Lightweight Directory Access Protocol (v3)*. IETF RFC 2251, 1997.

[30] D. Lu, P. Dinda, and J. Skicewicz. *Scoped and approximate queries in a relational grid information service*. In Proceedings of the 4th International Workshop on Grid Computing (Grid 2003), 2003.

[31] *Web Service Definition Language*. `http://www.w3.org/TR/wsdl`.

[32] *Simple Object Access Protocol (SOAP)*. `http://www.w3.org/TR/soap/`.

[33] R.A. van Engelen and K.A. Gallivan. *The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks*. In CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, page 128, 2002.

[34] A. O. Freier, P. Karlton, and P. C. Kocher. *The SSL Protocol Version 3.0*. `http://wp.netscape.com/eng/ssl3/draft302.txt`, 1996.

[35] *PostgreSQL*. `http://www.postgresql.org/`.

[36] *Java DataBase Connectivity*. `http://java.sun.com/products/jdbc`.

[37] *OpenLDAP website*. `http://www.openldap.org`.

[38] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, and D. Snelling. *Open Grid Services Infrastructure (OGSI) Version 1.0*. Global Grid Forum Draft Recommendation, 2003.

[39] J. Coppens, S. Van den Berghe, H. Bos, E.P. Markatos, F. De Turck, A. Oslebo, and S. Ubik. *SCAMPI: A Scalable and Programmable Architecture for Monitoring Gigabit Networks*. In Proc. of the 1st Workshop on End-to-End Monitoring Techniques and Services, 2003.

[40] *zlib compression library*. `http://www.zlib.net/`.

3
Grid Simulation

3.1 Introduction

Due to the size and complexity of typical Grid topologies and the large number of resources involved, it is a cumbersome (if not impossible) ordeal to have to build a real-life Grid testbed each time one wants to experiment with different Grid scheduling algorithms and resource management strategies on a variety of topologies. In order to facilitate research into Grid management and allow for a wider range of Grid configurations to be experimented with, one needs to resort to simulation. The Grid Information and Monitoring Framework described in the previous chapter can in this case be used to provide simulations with realistic computational, data, storage and network resource properties.

While a lot of Grid simulators are already in existence (see the related work in section 3.2), none of them provide accurate and up-to-date network resource modelling. Indeed, since one of the main characteristics of a Grid is its distributed nature, the interconnecting network should be treated with the same importance as any other resource (e.g. computational, storage), as the main focus of this dissertation is to incorporate network status information into the scheduling decision process. To cope with this hiatus, we have developed NSGrid, a Grid simulator based on Network Simulator 2 (ns-2 [1]). Ns-2 is an ongoing research project providing a discrete event network simulator with a multitude of accurate network protocol models (TCP/IP, multicasting, routing) on a variety of wired and wireless networks. NSGrid adds to ns-2 a layer modelling the Grid middleware as well

as components for simulating the application layer and user behaviour. It offers advanced Grid job, computational, storage and data resource models that can be mapped on an underlying ns-2 network. On top of the resource models NSGrid also provides a variety of management components such as a replica manager, monitoring and information service, scheduler, service manager, service monitor, etc.

The use of a Grid-specific simulation environment allows us to depart from restrictive resource and job models often encountered in machine and cluster scheduling [2]: over the years a lot of effort has been spent into developing algorithms to efficiently schedule jobs over a range of processing elements (i.e. cluster nodes, processors) [3, 4]. In order to obtain performance results, jobs were typically modelled as "work units" to be processed by (a set of tightly-coupled) "processors" that can perform work at a given rate, neglecting network and I/O requirements of jobs. As such, this model does not capture all ingredients essential to a Grid, where jobs process large amounts of data to be transferred between resources across networks exposing largely differing characteristics (i.e. less-than-infinite bandwidth, possible substantial delay, unpredictability, etc.) compared to a local area network/network bus.

One of the main tasks of NSGrid is evaluating different scheduling algorithms, most notably "network aware" heuristics: when choosing a computational resource from the Grid resource pool on which to run a job, suboptimal decisions can be made when selection is solely based on that computational resource's properties and status. Taking into account the current status of the network links interconnecting the Grid resources (such as computational resources, storage resources and data resources) can lead to better job placement strategies (especially when dealing with highly data-intensive Grid applications) and avoid cases where jobs are scheduled on a remote computational resource which is connected to the Grid through a low-bandwidth or already saturated network link. In such cases the Grid network links become bottlenecks for computational progress, meaning that a computational resource slot allocated to a job using such a link cannot be exploited to its full potential. We therefore see a need to adapt Grid scheduling algorithms to deal with the issue of network connectivity status. NSGrid can be used to accurately compare the performance of traditional network unaware versus network aware Grid scheduling algorithms and to evaluate the efficiency of different scheduling strategies (i.e. batch scheduling, asap scheduling, etc.) on a variety of Grid topologies and resource configurations. Sending data (either control messages or job data) between resources/management components is modelled as two layers in NSGrid: the first layer supports controlling the actual data stream between the distinct endpoints (i.e. point-to-point connection), while the second layer models the sending of network packets (routing, packet handling, network protocol simulation, etc.)

This chapter is structured as follows: section 3.2 details the related Grid simulation projects and the differences with NSGrid. In section 3.3 an overview of the simulation models is presented: the Grid, network resource, computational resource, storage resource, data resource and job models are thoroughly explained, followed by a presentation of the different management components (scheduler, replica manager, connection manager, information service, service manager and service monitor) and their functionality. Grid (re)scheduling strategies (whether to start a scheduling round upon arrival of a job, or delay a new scheduling round until multiple jobs are available) in NSGrid are discussed in section 3.4. Section 3.5 elaborates on the network aware and network unaware Grid scheduling algorithms, while the evaluation of those scheduling heuristics (using our NSGrid simulation environment) for different job classes in a typical Grid topology is detailed in section 3.6. Section 3.8 summarizes this chapter and gives some concluding remarks.

3.2 Related Grid simulators

Our Grid simulation environment (NSGrid) is based on the well-known ns-2 [1] network simulator. While not providing the most scalable simulation kernel (more scalable C++ simulation frameworks are available, such as DaSSF [5, 6] and OMNeT++ [7, 8]), ns-2 is an up-to-date, discrete event network simulator mostly used in academic networking research, partly due to its easy extendibility (open-source with a large support community). Ns-2 provides models for a wide range of protocols for both wired and wireless networks.

Notable existing Grid simulators include Bricks, MicroGrid, SimGrid, GridSim, ChicSim and OptorSim.

The Bricks Simulator [9, 10] focuses on client/server interaction in global high performance computing systems. It allows for a single centralized scheduling strategy, which does not scale well to large Grid systems and does not support the notion of multiple (competing) schedulers.

MicroGrid [11, 12] is an emulator modelled after Globus [13], allowing for the execution of Globus-enabled applications on a virtual Grid system. Research into the area of Grid scheduling algorithms can be cumbersome with this kind of approach, since it requires the construction of an actual Globus application to test.

SimGrid [14] is designed to simulate task scheduling (centralized or distributed) on Grids. Version 1 of SimGrid can be regarded as a low-level toolkit (which interfaces to the C programming language) from which domain-specific simulators can be built. The second version of SimGrid is dubbed MetaSimGrid [15] and is essentially a simulator built upon this toolkit to enable the construction of simulations with multiple schedulers (as C programs). Models for network links as well as for TCP connections are present in SimGrid. This validated TCP implementation allows for smaller simulation times when compared to the packet-level

TCP simulation performed by network simulators. Of course, simulations using other transport protocols that are not readily available in SimGrid require that these protocols are implemented first, whereas using a network simulator ensures easy access to a wide range of protocols. The simulated application consists of several tasks, organized into a Directed Acyclic Graph (DAG). MetaSimGrid is focused on scheduling this application type in a master-slave environment.

GridSim [16, 17] is a discrete-event Grid simulator based on JavaSim [18] (which has recently evolved to J-Sim [19] and has a similar Tcl/Java dual-language simulation environment as ns-2). This simulator allows to simulate distributed schedulers, and is specifically aimed at simulating market-driven economic resource models. While its computational resource models are highly configurable, only a basic notion of network connectivity is supported and underlying network dynamics are not simulated accurately.

The Chicago Simulator [20, 21] is a simulation framework built on top of Parsec [22] for studying scheduling and replication strategies in Grids. A Grid is modelled as a collection of interconnected Grid sites with network connectivity of each Grid site modelled as a single parameter (describing the bandwidth of the gateway connecting this Grid site to the other Grid sites). As such, it does not provide the level of network resource detail that is modelled in NSGrid.

OptorSim [23, 24] is a Java [25] based Grid simulator focussing on evaluating the performance of data access optimization algorithms. Its architecture is based on the EU DataGrid [26] architecture. OptorSim includes an economic model, using a peer-to-peer auction protocol that optimizes both the selection of replicas for running jobs and the dynamic creation of replicas in Grid sites using a file revenue prediction function. OptorSim takes network bandwidth into account when transferring job input/output data (although it does not actually simulate any existing network protocols) and currently has no notion of Grid services.

Simulation of Grid scheduling strategies which take both computational resources and data resources (more specifically, data locality) into account have been reported upon in [27]. In this work, however, the network connecting different sites is not simulated, but it is assumed that the different sites are connected through a VPN-like construction over which TCP communication occurs. Scenarios where files are pre-staged are considered, but data transfers in parallel with job execution are not.

Table 3.1 summarizes the main differences between the discussed Grid simulators.

	scheduler	network model	packet level simulation	generic
Bricks	single	basic	no	client-server
MicroGrid	distributed	very advanced	yes	Globus emulator
SimGrid	distributed	advanced	no	generic Grid
GridSim	distributed	basic	no	generic Grid
ChicagoSim	distributed	basic	no	generic Grid
OptorSim	distributed	basic	no	generic Grid
NSGrid	distributed	very advanced	yes	generic Grid

Table 3.1: Grid simulator characteristics

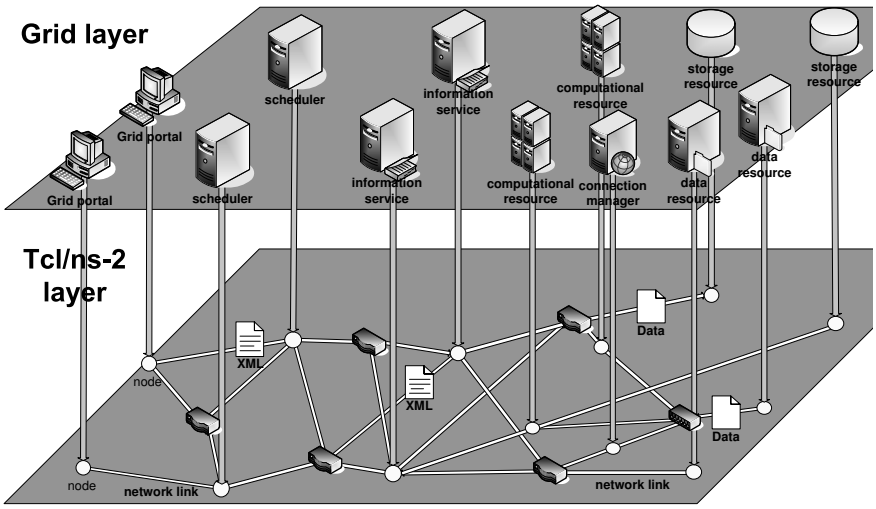


Figure 3.1: NSGrid Tcl/C++ dual layered architecture

3.3 NSGrid simulation framework

3.3.1 NSGrid architecture

We have designed and implemented a Grid simulation framework that takes into account network resource parameters and that accurately models network traffic using a variety of protocols and applications (FTP traffic over TCP connections, variable bitrate UDP traffic, wireless traffic, etc.). To this end the simulator was built on top of the widely used ns-2 network simulator. The key difference with other Grid simulation toolkits such as GridSim and SimGrid is that NSGrid makes use of an existing network simulator, which provides for realistic and accurate

models for network links and protocols (ns-2 is continuously being improved and updated with new network models and protocols). Ns-2 uses the Tcl [28] scripting language to drive the simulation, while C++ [29] is used for the implementation of the various Grid components in order to reduce simulation time. This Tcl front-end gives us the possibility to rapidly create new simulation scenarios and offers a high-level abstraction of the simulator entities' implementations. NSGrid's architectural specification has been developed together with colleague Pieter Thysebaert.

Major components of our simulator are management components (schedulers, information services, connection managers, service managers, service monitors, etc.) and resources (computational, data, storage and network resources). Each component (with the exception of network resources) is associated with a single ns-2 node. These nodes and the network links that interconnect them are regular ns-2 entities and model the Grid's physical topology (figure 3.1).

The simulator components can be seen as models for real-life software components running on the nodes they are hosted by, as communication between components is a source of network traffic in the simulation. The exact nature of this traffic can be modelled after e.g. the communication middleware used by the software components. Messages and RPC calls exchanged between components are implemented as XML messages that can easily be transported between the C++ and Tcl layers. Job I/O data is sent as raw data (e.g. modelled as TCP or UDP traffic) across ns-2 links and nodes from source to destination.

The NSGrid Grid simulation layer C++ source is composed of 47 classes (not including topology generation/GUI visualisation tools), comprising 19.000 lines of code (19 KLOC). The most important classes are shown in figure 3.2. All elements that can be assigned to an ns-2 node (resources, management components and clients) inherit from the *GridObject* class, which connects the Grid simulation layer with the ns-2 network simulation layer by providing methods for sending/receiving data and event control (see listing 3.1 for the most important methods supported by the GridObject class). Each *Resource* object contains a *ResourceInfo* object which stores resource properties/status information and provides methods for reading/writing from/to XML (for sending resource information between management components). The *GridLogger* class provides simulation logging functionality with support for multiple log levels (NSGrid users can select the logging levels they want to see messages from) and error reporting.

In what follows we briefly given an overview of the methods in listing 3.1:

- *virtual int command(...)*: events fired in the Tcl layer are delegated to the appropriate C++ method through the *command(...)* subroutine. This basically provides the glue between the Tcl and C++ layers.

- *static void sendAndExecute(...)*: sends commands between components. The command is sent over the simulated network links between source and des-

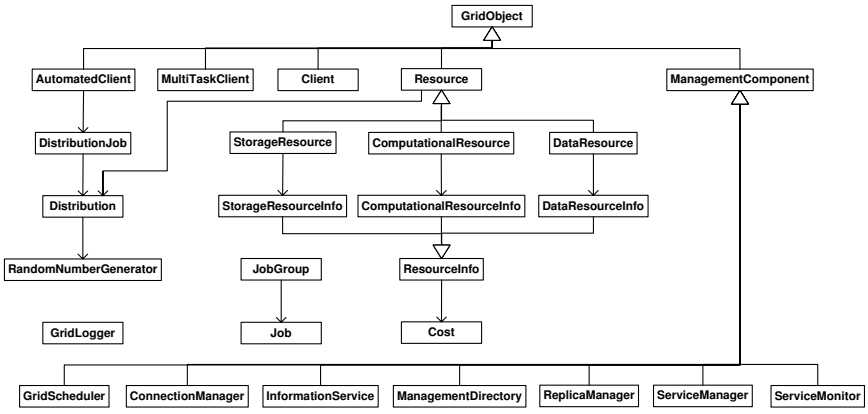


Figure 3.2: NSGrid implementation architecture

Listing 3.1: GridObject interface

```
class GridObject : public TclObject {
public:
    virtual int command(int argc, const char* const * argv) = 0;
    static void sendAndExecute(string sourceNode, string destinationNode, string command);
    static void sendDataAndExecute(string sourceNode, string destinationNode, int size, string
        command);
    static double getSimulatedTime();
    static string getNode(string componentName);
    static int getHops(string source, string destination);
    static string setEvent(double time, string command);
    static string setEventNow(string command);
    static void cancelEvent(string eventId);
    static string releaseReservation(string resourceName, string reservationID);
    void log(string source, string logMessage, int logLevel);
};
```

tination, and, once all command data has been received at the destination, the Tcl command specified by argument *command* is fired.

- *static void sendDataAndExecute(...)*: sends data between components. The data (with size specified by parameter *size*) is sent between source and destination, and, once all data has been received at the destination, the Tcl command is fired.

- *double getSimulatedTime()*: return simulated time.

- *string getNode(...)*: returns the network node associated with the given component.

- *static int getHops(...)*: returns the amount of network hops between source and destination nodes.
- *static string setEvent/setEventNow(...)*: sets event at given point in time and returns event ID.
- *static void cancelEvent(...)*: cancels event with given event ID.
- *static string releaseReservation(...)*: releases resource reservation with given ID.
- *void log(...)*; provides output logging functionality.

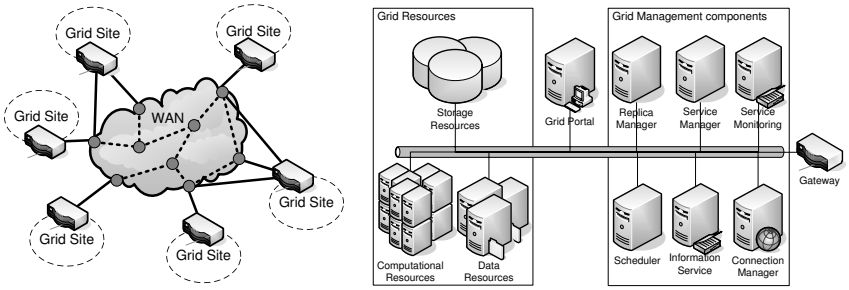
3.3.2 Grid model

We regard a Grid as a collection of *Grid sites* interconnected by WAN and MAN links (see figure 3.3). Each Grid site has its own resources and a set of management components, all of which are interconnected by means of LAN links. Management components include a *connection manager* (capable of offering network QoS by providing bandwidth reservation support, and responsible for monitoring available link bandwidth and delay), an *information service* (storing registered resources' properties and monitoring their status), a *scheduler*, a *service manager*, *service monitor* and *replica manager*. The explicit treatment of the network as a "resource" allows management components to take decisions based on observed and expected future load of the network.

Each Grid site can have one or more Grid portals, through which clients can submit jobs. These jobs are then scheduled on a collection of resources by a scheduler. To this end, the scheduler makes reservations with the resource managers; in our environment, a connection manager manages a collection of network links, while the computational, storage and data resources double as their own manager. To ensure connectivity with the outside world (and in particular with other Grid sites), each Grid site designates one or more of its underlying ns-2 network nodes as a gateway to the WAN/MAN.

3.3.3 Job model

The basic unit of work in our model is a *job*, which can roughly be characterized by its length (time it takes to execute on a reference processor), computational requirements (memory, operating system, temporary disk space, installed applications, etc.), maximum cost of processing, deadline, the needed input data, the output data size, the *burstiness* with which these data streams are read or written, and the service class to which it belongs (note that some of these job parameters are optional). It will be shown in chapter 4 that it can be beneficial to assign jobs



(a) Grid (b) Grid site

Figure 3.3: Grid model

from different Grid applications to one and the same service class in order to group jobs with similar resource/QoS requirements, but for now the service class of a job denotes the application type it spawned from. Knowing the job's total length and the frequency at which each input/output stream is read/written (when executed on a reference processor), the total execution length of a job can be seen as a concatenation of instruction "blocks". The block of input data to be processed in such an instruction block is to be present before the start of the instruction block; that data is therefore transferred from the input source (in this case a data resource offering the necessary input) at the start of the previous instruction block. Similarly, the output data produced by each instruction block is sent out at the beginning of the next instruction block. We assume these input and output transfers occur in parallel with the execution of an instruction block. Only when input data is not available at the beginning of an instruction block or previous output data has not been completely transferred yet, a job is suspended until the blocking operation completes.

A typical job execution cycle (with one input stream and one output stream) is shown in figure 3.4 and figure 3.5. The presented model allows us to mimic both *streaming* data (high read or write frequency e.g. a low/high resolution browse job from the domain of audio/visual production Grids; see table 5.1 for more information) and *data staging* approaches (read frequency set to 1 e.g. a rendering job where graphical data and scene rendering information is prefetched to a computational resource, the scene is then rendered and, when rendering is complete, output is stored onto a storage resource). Note that jobs can receive input data / store output data from multiple data / storage resources at the same time.

An overview of all job parameters is given by:

- Job arrival time at Grid portal
- Grid site from which the job originated

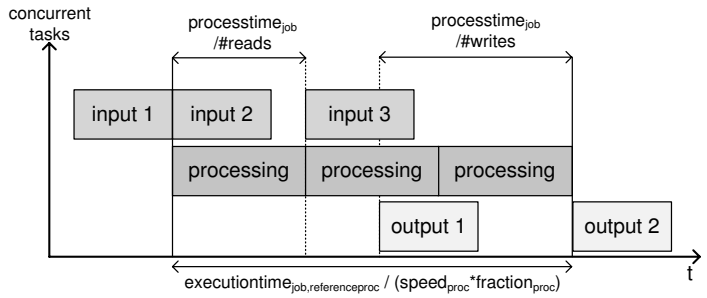


Figure 3.4: Non-blocking job, simultaneous transfer and execution

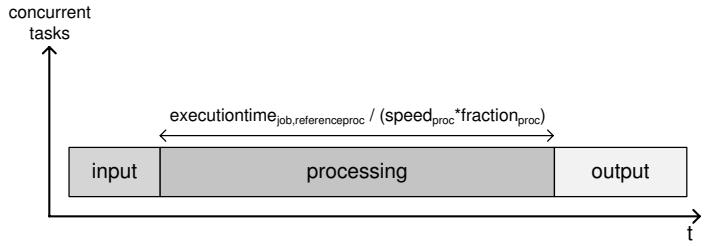


Figure 3.5: Non-blocking job, pre-staged input data

- Start time: time at which job may begin processing
- Amount of processing time needed on reference processor
- Duration: minimum duration of job (to ensure that job processing occurs at a defined rate e.g. video viewing jobs)
- Minimum processor speed asked for by job
- Temporary disk space required on processing element
- Software installation (operating system/programs) required for job to run
- Input data: access type (FTP, secure copy, etc.), minimum required retrieval speed, ID of input sets required, number of reads
- Output data: access type (FTP, secure copy, etc.), minimum required storage speed, number of writes, time output data needs to be kept available
- Budget: maximum cost of executing job
- Deadline
- Service class

3.3.4 Client model

In our simulation environment, a client is a component which automatically submits jobs from a particular service class to a Grid portal (which in turn delivers these jobs to a scheduler). The "home site" of these jobs is the Grid site where the client is located. All job characteristics, together with job submission start/end times and job interarrival times are specified statistically (normal, uniform, zipf and exponential distributions are supported) according to the client's configuration (specified in the Tcl startup script) or, alternatively, are read from trace files containing previously recorded job submission behaviour. The latter approach allows for identical job load reproduction under different Grid topologies / resource setups. Note that clients are not required to wait for a previously submitted job to finish before launching another job.

Multi-service class clients, capable of constructing and submitting jobs from different service classes (with job parameters generated from distinct service class' configurations) can be instantiated in NSGrid. Each multi-service client can have a service class workflow assigned to them. These workflows define the probability x of constructing and submitting a job from service class y when a job from service class z was last submitted by this client (see figure 3.6).

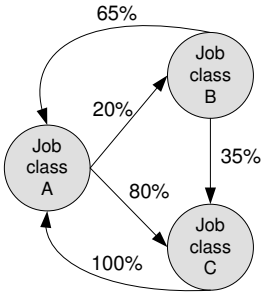


Figure 3.6: Sample multi-service client workflow

Clients are specified by:

- A list of job/service classes with job parameters specified statistically
- Workflow between the different job/service classes
- Grid portal location
- Job generation start time
- Job generation end time

3.3.5 Resource models

3.3.5.1 Computational resource model

Each computational resource is viewed as a monolithic entity with a certain processing power. Its main capabilities are defined by the following parameters:

- The number of processors and their respective processing power (relative to a reference processor)
- Memory available to jobs
- Disk space available for storing temporary job output
- Installed operating system and applications/software
- Load: job load (processing, memory, disk space) and reservations
- Dynamic resource model: resource failure probability, probability to go offline and average time before the resource restarts
- Cost: price when using this computational resource (depending on user class of client that submitted job to be processed)
- Service class ID: either "0", meaning the computational resource can be used by any job from any service class, or a non-"0" service class ID meaning the resource can only be used by jobs from service classes matching that ID. The service class ID can be dynamically assigned by the service manager components

This model can be used to represent both multiprocessors and clusters, provided that, in the latter case, the internal network connecting the various cluster nodes performs sufficiently (in a well balanced multiprocessor system, the network bus interconnecting the processors will only rarely be a performance-limiting bottleneck). If the network interconnecting the different processing elements can be the source of bottlenecks, one should instantiate a computational resource for each processing element and interconnect these resources by means of bandwidth limited network resources (see 3.3.5.4).

Before accepting a job for processing, a computational resource will check the requirements of the computational reservation (sent by a scheduler) to see if these do not conflict with existing and/or future computational reservations. If there is a conflict, the computational resource will reject the reservation and inform the requesting scheduler. Once a job is accepted for processing, the job description is parsed for information regarding (optional) data and storage resources that are to be used for retrieving/storing input/output data. In our model, the computational

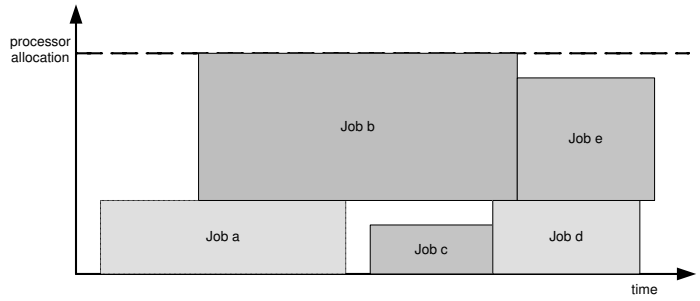


Figure 3.7: Computational resource processor allocation

resource is responsible for starting the retrieval of each input block and sending output blocks to the necessary storage resources.

As can be seen in figure 3.7, a computational reservation allocates a fixed fraction of the computational resource's processing power over a certain amount of time (so it can ensure that deadlines will be observed). During the lifespan of the reservation, the allocated fraction itself will never be modified, but, due to job blocking, the time this processor fraction is allocated to a job can be enlarged if it does not infringe other computational reservations.

When a job has finished sending its last output block (or, if no output had to be sent, once the job has no further processing to do) the resource will inform the responsible scheduler that the job is finished, and will release the job's computational reservation.

3.3.5.2 Storage resource model

Storage resources provide disk space to store job output data. In our model, storage resources are described by

- The total available storage space
- Load: storage space allocated to jobs and reservations
- Dynamic resource model
- Cost: price when using the storage resource (depending on user class of client that submitted job)
- Service class ID: either "0", meaning the storage resource can be used by any job from any service class, or a non-"0" service class ID meaning the resource can only be used by jobs from service classes matching that ID

When a storage resource receives a storage reservation request, it checks to see if this reservation does not conflict with the already granted reservations (in terms

of disk space) and, if possible, grants the reservation. A job can choose to keep its output data stored until the job finishes processing (at which time the storage resource will release the job's storage reservation), or it can opt to keep output data stored for a specified time (at the possible expense of extra cost).

While a storage resource does not perform computational processing of jobs, it can be attached to the same network node as some computational and/or data resource.

3.3.5.3 Data resource model

Data resources serve the purpose of providing input data for jobs. In our model, data resources are described by

- Available data sets (by ID), their respective sizes and usage characteristics (the latter is for replication purposes)
- Available storage space for datasets
- Load: content being read by jobs and reservations
- Dynamic resource model
- Cost: price when using the data resource (depending on user class of client that submitted job)
- Service class ID: either "0", meaning the data resource can be used by any job from any service class, or a non-"0" service class ID meaning the resource can only be used by jobs from service classes matching that ID

Each time a job retrieves an input data set, the data resource updates its internal data set usage properties (time accessed, number of times the data set has been accessed in last time frame). These usage characteristics can in turn be used to decide which data sets no longer will be supported in favour of new (replicated) datasets (i.e. storage space will be made available for storing often utilized, replicated data sets by freeing disk space taken up by less frequently used data sets; for more information see section 3.3.6.2).

While a data resource does not perform computational processing of jobs, it can be attached to the same network node as some computational and/or storage resource.

3.3.5.4 Network resource model

Interconnections between resources (i.e. between two non-network resources) are modelled as a set of network links, providing a route between the source and destination resource. Connection reservations, each offering a guaranteed total

bandwidth available to a particular Grid job or service class, can be set up by the connection manager. Of course, these connections can only be set up if, in the underlying ns-2 network topology, a route (with sufficient bandwidth capacity) exists between the nodes to which these resources are attached. Grid resources can also be interconnected by means of capacitated VPNs: in this case, a VPN tunnel (with guaranteed bandwidth availability) is set up between Grid resources for a particular Grid job service type. This VPN tunnel carries all connections matching the VPN's endpoints and service class. Such connections can be set up as long as the VPN's residual bandwidth can satisfy the connection demands. VPNs allow for the upfront reservation of bandwidth for a particular service type (see figure 3.8).

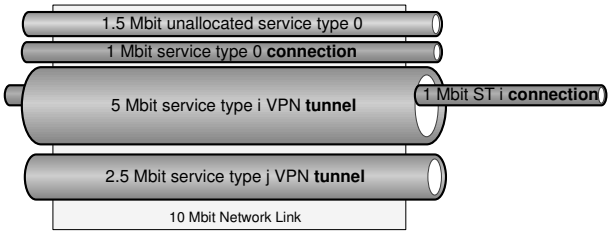


Figure 3.8: Network model

Connections in our model are a set if network links, with each network link described by:

- Endpoints
- Total bandwidth capacity
- Residual bandwidth of non-VPN connections over time (advance network reservations are supported)
- Residual bandwidth for each pre-assigned VPN over time (advance network reservations are supported)
- Delay
- Connection manager responsible for network link (see section 3.3.6.3)

3.3.6 Management components

3.3.6.1 Information service

An information service offers a computational, storage and data resources' property and status repository. Each time a resource is instantiated in the Grid, it will register its static properties and dynamic status info with at least one information service. An information service can be queried by any other Grid management

components (e.g. scheduler, service manager) for resources meeting certain requirements (e.g. available memory, installed software, available datasets). The information services perform matchmaking on this query, i.e. they select a set of resources (from their registered resource repository) that meet the requirements of the resource query. The resulting resource set is sent back to the management component requesting resource information. Typically, since a single central information service for a Grid would not be scalable, each Grid site offers one (or more) information service component storing (a subset of) the local resources.

Each time a resource's state changes (e.g. because of accepting a new job for processing) it contacts the information services it is registered with and sends up-to-date status information (note that these status update messages, as any control message in NSGrid, can suffer from network delays, temporarily rendering the status information contained within the information service outdated).

The information service also monitors the liveliness of each registered resource (in order to detect resources failures). At configurable intervals it sends a heartbeat message to these resources and expects to receive the responses in adequate time (see section 3.3.7). If these responses do not arrive in due time, the resource is unregistered from the repository.

Listing 3.2: Information service interface

```
class InformationService: public ManagementComponent{
public:
  void getCRs(string XMLQuery, string tclName, string requestId);
  void getSRs(string XMLQuery, string tclName, string requestId);
  void getDRs(string XMLQuery, string tclName, string requestId);

  void registerScheduler(string schedulerTclName);
  void registerResource(string resourceType, string XMLdescription, string resourceTclName);
  void updateResource(string resourceType, string XMLdescription, string resourceTclName);
  void updateServiceClass(string resourceType, string resourceTclName, int serviceClass);
  void unRegisterResource(string resourceType, string resourceTclName);

  void pingRegisteredResources();
  void pingAnswer(string resourceType, string resourceTclName);
  void checkPingAnswers();
};
```

Listing 3.2 shows the most important public methods (not including constructors/destructors/getters/setters etc.) supported by the information service. In what follows we briefly give an overview of these methods. Note that all methods return a *void* value, as method return values are not communicated between components on a C++ level, but instead act as a source of network traffic (i.e. methods that need to send return values do so by invoking the *sendAndExecute* method of GridObject, as seen in listing 3.1).

- *void getCRs/getSRs/getDRs(...)*: retrieves the computational/storage/data resources complying with the requirements stated in parameter *XMLQuery*. Results (along with request ID) will be sent back to the component specified by parameter *tclName*.
- *void registerScheduler(...)*: registers a scheduler with an information service. The information service keeps registered schedulers informed of sudden resource unavailability.
- *void registerResource(...)*: registers a resource with an information service. Resource properties and current state are specified by parameter *XMLdescription*, while the resource's Tcl name is specified by parameter *resourceTclName*.
- *void updateResource(...)*: when resource state changes (e.g. new job accepted for processing on CR), the resource calls this method to give the information service up-to-date status information.
- *void updateServiceClass(...)*: as will be explained in chapter 4, service managers can assign a resource to a service class in order to allow only jobs from that particular service class to use that resource.
- *void unRegisterResource(...)*: removes the resource from the information service resource repository (e.g. when resource goes off-line or when it fails).
- *void pingRegisteredResources()*: at regular intervals, the information service sends out heartbeat messages to each registered resource.
- *void pingAnswer(...)*: resources that answer the heartbeat message do so by calling this method.
- *void checkPingAnswers()*: at a configurable time after heartbeat messages have been sent out, the information service checks to see which resources have answered the heartbeat message. The resources that did not answer the message in due time, are assumed to be failing and unregistered from the information service.

3.3.6.2 Replica manager

A replica manager monitors job input data retrieval behaviour in such a way that when a job is in need of a dataset that is not present at the job's computational processing site (which is not necessarily the job's originating site), a replica note is sent to that Grid site's local data resources, asking them if they are interested in replicating that dataset locally (the job's needed data set will be transferred from

a remote data resource to the computational resource where the job is processing, so it will be available for replication). The data resources can then choose to either ignore this replication request (based on the usage characteristics of their offered datasets) or they can store the new data set locally. If a data resource decides to replicate the data locally, it can simple store the dataset along with the existing datasets (if there is enough free storage space) or it can choose to replace an existing dataset using a Least Recently Used algorithm (in this case the least recently used dataset(s) will no longer be supported from that particular data resource).

Listing 3.3: Replica manager interface

```
class ReplicaManager: public ManagementComponent {
public:
    void registerInformationService(string ISTclName);
    void getDRInformation(string requestId);
    void queryDRResult(string requestId, string DRListXML, string ISTclName);
    void jobUsedDataset(string jobId, int dataID, double size, int service, string CR);
};
```

Listing 3.3 shows the most important public methods (not including constructors/destructors/getters/setters etc.) supported by the replica managers. In what follows we briefly describe these methods:

- *void registerInformationService(...)*: adds information service (with name *ISTclName*) to the known replica manager's information services. These information services will be queried for data resource information.
- *void getDRInformation(...)*: queries information services for data resource information (supported data sets, data set storage space, etc.).
- *void queryDRResult(...)*: receives list of data resources registered with information service (with name *ISTclName*).
- *void jobUsedDataset(...)*: send replication note to data resource, informing this data resource that a recently used data set is available for replication at a local computational resource.

3.3.6.3 Connection manager

A connection manager monitors properties (e.g. total bandwidth) and status information (e.g. available bandwidth, delay) of the different Grid network links. It can be queried by any other management component (e.g. service managers, schedulers) to retrieve connection information (available bandwidth, delay, routing information, total bandwidth, etc.) between a source and destination network

node. If a connection query is received, the connection manager will ask ns-2 for the route from source to destination and, once routing information has been received, will inspect the status of all network links along this route. The connection manager will then construct a connection information object, containing info about all links along this route and available bandwidth/delay between source and destination. This connection information will be sent back (in XML) to the querying management component.

Next to an informative function, a connection manager is also responsible for setting up network resource reservations. Both end-to-end connections and service class' VPN reservations are supported (see network model in section 3.3.5.4). To this end, management components can ask the connection manager to reserve a certain amount of bandwidth between a source and destination node. The connection manager will inspect all network links on the route between source and destination, and, if bandwidth requirements are met on each link, will reserve the requested amount of bandwidth, grant the connection reservation and inform the requesting management component.

Note that reservations are not physically set up by the connection manager: if the bandwidth requirements of the requested connection reservation are not infringing previously guaranteed connection reservations' minimum bandwidth, the request is granted. If however this is not the case (due to the use of stale resource state information when assigning resources to jobs in the scheduling round), the connection reservation request is rejected and the job will be put back in the scheduler queue until the next scheduling round. The connection manager thus operates by bookkeeping all granted connection reservations and denying new reservations that would infringe on those previously granted reservations.

Listing 3.4: Connection manager interface

```
class ConnectionManager: public ManagementComponent {
public:
    void getConnections(string XMLQuery, string tclName, string requestId);
    void getLinksForJob(string tclName, string requestId);

    void testAndAllocate(string reservationId, string XMLConnectionReservation);
    bool setupVPN(pair<string,string> endpoints, double capacity, double delay, int service);
    bool setupConnection(string ID, pair<string,string> endpoints, double bandwidth, double
            delay, int service, double starttime, double timespan);
    void releaseReservation(string reservationId);

    void autoSetupServices();
    void addServiceClass(int service, double bandwidthPercentage);
    void removeServiceClass(int service);
};
```

Listing 3.4 shows the most important public methods (not including construc-

tors/destructors/getters/setters etc.) supported by the connection managers. In what follows we give a brief description of these methods:

- *void getConnections(...)*: retrieves end-to-end interconnection information (available bandwidth, delay, etc.), with endpoints and requirements described by the XML connection query. The *tclName* parameter denotes the component to which the answer for this query has to be sent.

- *void getLinksForJob(...)*: retrieves information regarding network links monitored by this connection manager.

- *void testAndAllocate(...)*: connection manager checks if connection reservation request can be granted and sets up the connection if possible.

- *bool setupVPN/Connection(...)*: sets up VPN/connection with specified capacity/bandwidth between two endpoints.

- *void releaseReservation(...)*: releases previously granted connection/VPN reservation.

- *void autoSetupServices()*: automatically partitions network resources amongst service classes based on service classes' average bandwidth requirements (see chapter 4).

- *void add/removeServiceClass(...)*: adds/removes service class average bandwidth requirements as calculated by the service manager (see chapter 4).

3.3.6.4 Service manager

Service managers receive information regarding service class characteristics from the service monitoring components. At regular intervals, the service manager queries the information services for all Grid resources. Once the answer to this query has been received, one of the service manager's resource partitioning algorithms is applied to the resource set, assigning each Grid resource to a particular service class. The resulting resource-to-service partitioning solution gets sent back to the information services, who in turn change the service class property of their registered resources. Resources that have a service class ID assigned to them will from then on only be returned by resource queries for jobs from that particular service class. If the service manager's partitioning algorithm also works in on network resources, the connection manager will be contacted to make service bandwidth reservations. Detailed information about the service manager and associated interface can be found in chapter 4.

3.3.6.5 Service monitor

Service monitors inspect job submission behaviour at the Grid portals (recall that a Grid portal acts as a job submission gateway for Grid users): each time a job is submitted, job requirements (service class, priority, needed input data sets and sizes, output storage sizes, computational requirements, etc.) are extracted and overall service class properties (e.g. average job interarrival time, average I/O data sizes, average processing requirements) are adjusted. When a service monitor has gathered adequate service class characteristics, it sends this collected service class' information to its known service managers, so as to allow them to have up-to-date service class characteristics for use by the resource-to-service partitioning algorithms. For detailed information about the service monitor and associated interface we also refer to chapter 4.

3.3.6.6 Grid scheduler

As described in section 3.3.4, multiple clients submit, according to their job submission configuration, jobs to Grid portals who in turn send them to a scheduler. Each time a job is received by a scheduler, it will query its known information services for resources that can be allocated to this unscheduled job. The resulting resource set is sent back to the scheduler, and, if the scheduler also wishes to receive resource interconnection information, it will proceed by contacting its connection manager and query the status (available bandwidth, delay, reservations) of the network links interconnecting the resources received for that particular job by the information services.

Once resource query results have been received for an unscheduled job, the scheduler applies one of its scheduling algorithms to the received set of resources (either instantly when the scheduling strategy is "as soon as possible" or, when scheduling in batch, at the time of a new scheduling round), and selects those that will be allocated to each job. Parameters that can be taken into account when scheduling a job include scheduling objective (minimization of job response times, priorisation of certain service classes, minimize cost, etc.), job deadlines, budget constraints, etc..

When a scheduling decision has been made, the scheduler contacts the computational, data and storage resources (and if bandwidth reservations are necessary the connection manager) to request the necessary resource reservations. If all resource reservation requests are granted (note that, due to stale resource status information, or due to scheduler competition, reservations can be rejected), the job is put into the scheduled state and is sent to the selected computational resource, which in turn will manage the job's input from the selected data resources and/or output to the selected storage resources (recall that all computational, storage and data resources will send up-to-date status information to the information services

with which they are registered once a reservation has been granted). If network resource reservations have been set up, job I/O data will be sent across the reserved connections.

The moment a job is finished, the computational resource responsible for processing this job notifies the scheduler, releases its computational reservation and sends up-to-date status information to its registered information services. The scheduler proceeds by notifying the client that its job has finished, and releases storage, data and (if necessary) network resources. Note that XML control messages and raw I/O data arrival times depend on the bandwidth / protocol / application that is used across each network link over which this data is sent (as simulated by the underlying ns-2 network layer).

In case insufficient resources are available at the time of scheduling or if resource reservation requests are rejected, jobs that do not get resources assigned to them are requeued for scheduling at a later time or (in case multiple schedulers exist) can be forwarded to a different scheduler. NSGrid supports advance reservation of all resource types.

Listing 3.5 shows the most important methods (not including constructors/destructors and most getters/setters) supported by the Grid schedulers. In what follows we give a brief description of these methods:

- *void setQueuePolicy/ScheduleInterval/ScheduleAlgorithm/ASAPRescheduleDelay(...)*: allow selection and configuration of the scheduling algorithm to be used by the Grid scheduler. Additionally, the time between scheduling rounds, queue policy (FIFO, priority rearranged, etc.) and the rescheduling delay (in case scheduling is impossible) when scheduling ASAP can be changed with these methods.

- *void addInformationService/ConnectionManager/ReplicaManager(...)*: adds management components so they can be queried by the Grid scheduler.

- *void submitJob(...)*: submits a job to the Grid scheduler.

- *void submitJobWithEndingCallback(...)*: submits a job to the Grid scheduler. Once the job has ended, the scheduler will inform the client of this event.

- *void endJob(...)*: ends job, cancelling all existing reservations for this job and removes it from the scheduling queue.

- *void getResources/Connections/LinksForJob(...)*: queries information service and connection manager components for computational, data, storage and network resource property and status information.

- *void queryCRsResult/querySRsResult/queryDRsResult(...)*: receives answers to information service queries.

Listing 3.5: Grid scheduler interface

```
class GridScheduler : public ManagementComponent {
public:
    void setQueuePolicy(string policy);
    void setScheduleInterval(double interval);
    void setScheduleAlgorithm(string algorithm);
    void setASAPRescheduleDelay(double delay);

    void addInformationService(string informationServiceTclName);
    void addConnectionManager(string connectionManagerTclName);
    void addReplicaManager(string gridsite, string replicaManager);

    void submitJob(string jobXML);
    void submitJobWithEndingCallback(string jobXML, string client);
    void endJob(JobId jobId);

    void getResourcesForJob(const JobId& jobId);
    void getConnectionsForJob(const JobId& jobId);
    void getLinksForJob(const JobId& jobId);
    void queryCRsResult(string requestId, string XMLCRList, string informationService);
    void querySRsResult(string requestId, string XMLSRList, string informationService);
    void queryDRsResult(string requestId, string XMLDRList, string informationService);
    void queryConnectionsResult(string requestId, string XMLConnectionList, string
        connectionManager);
    void queryLinksForJobResult(string requestId, string XMLLinkList, string connectionManager);

    void CRFailure(string CRTclName);
    void SRFailure(string SRTclName);
    void DRFailure(string DRTclName);
    void CRAllocationFailed(string CRTclName);
    void SRAllocationFailed(string SRTclName);
    void DRAllocationFailed(string DRTclName);
    void ConnAllocationFailed(string connectionId);
};
```

- *void queryLinksForJobResult(...)*: receives answers to connection manager queries.

- *void CRFailure/SRFailure/DRFailure(...)*: methods called by an information service when it detects a resource failure.

- *void CRAllocationFailed/SRAllocationFailed/DRAllocationFailed(...)*: resources call this method when a job reservation request is not granted.

- *void ConnAllocationFailed(...)*: a connection manager calls this method when a connection reservation request is not granted.

3.3.7 Dynamic resource model

Sudden failure of Grid resources (computational, data and storage resources) is detected by the information service components and the appropriate actions are taken to ensure that the jobs that were relying on the failing resource get rescheduled. Our dynamic model supports two notions of unavailability:

3.3.7.1 Resource failures

Unexpected Grid resource unavailability. These failures will be detected by the information service (who periodically sends a heartbeat message to each resource registered with it). If a resource does not reply to this message within a specified time-interval, failure is assumed. The information service proceeds to unregister the resource from its repository, and sends a notification message to the Grid scheduler(s) that had jobs running on the crashed resource. The affected schedulers then revert jobs that were utilizing the failing resource to the unscheduled state and put them back in the scheduling queue (see figure 3.9). Resource failures can be specified by means of two distribution-type parameters: “time before resource failure” and “time before resource restart”.

3.3.7.2 Resource unavailability

Each Grid resource may unregister itself at any time by sending a message to the information services it is registered with. The resource will then proceed by contacting the Grid schedulers that have jobs utilizing it (either for job processing or for retrieval/storage of I/O data). If checkpointing is enabled, the last checkpoint of jobs running on that particular resource will be sent to the scheduler responsible for allocating the job’s resources. When a scheduler is notified of the unavailability of a resource, it looks up which jobs were scheduled on that resource, cancels all resource reservations of those jobs, reverts the state of those jobs to their initial state (the “unscheduled” state with no work done) and readies the jobs for rescheduling. If a checkpoint is available, the job will continue processing from that checkpoint on.

3.3.8 NSGrid operation

At startup the simulator reads a Tcl script defining the Grid topology and location of the various resources, management components and clients (this script can be constructed manually (through a dedicated GUI as seen in figure 3.11) or generated by automatic topology generator tools based on GridG [30, 31]).

During simulation, each event can be logged (logging supports multiple log-levels, providing a filtering function for messages of lower importance) and either written to the screen or stored in an output file. A GUI parsing this output (see

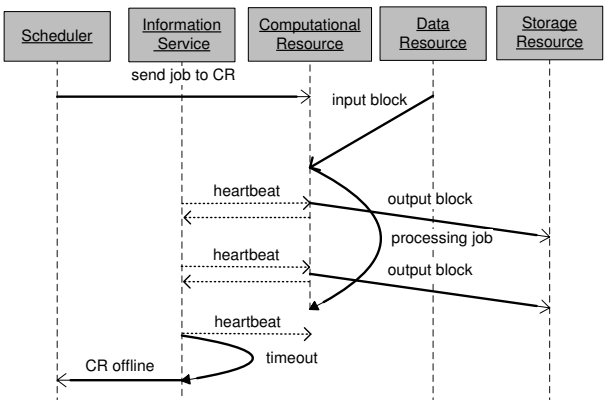


Figure 3.9: Computational resource failure

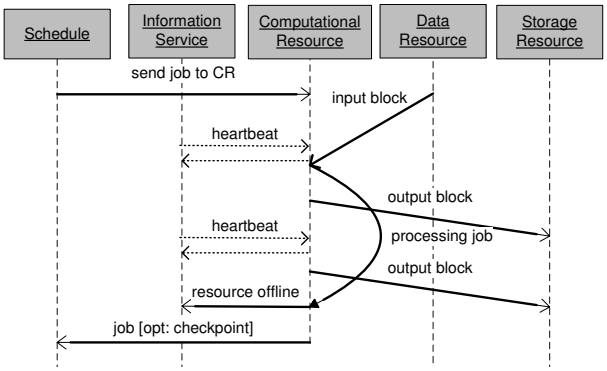


Figure 3.10: Computational resource unavailability

figure 3.12) allows for extensive filtering and sorting functions (e.g. in case one is only interested in the behaviour of a particular resource).

When a simulation has ended, NSGrid provides file output regarding job lifetime (arrival time, launch time, end time, execution speed, resources utilized, network usage, etc.) and management component performance (schedule computing time, resource-to-service partitioning times, etc.). Tools have been provided to parse this XML output to "comma separated value" files for easy use in spreadsheet programs.

3.4 Scheduling strategies

Grid scheduling algorithms attempt to allocate resources to jobs in such a way that a given criterium (e.g. average job response time, resource usage efficiency, cost)

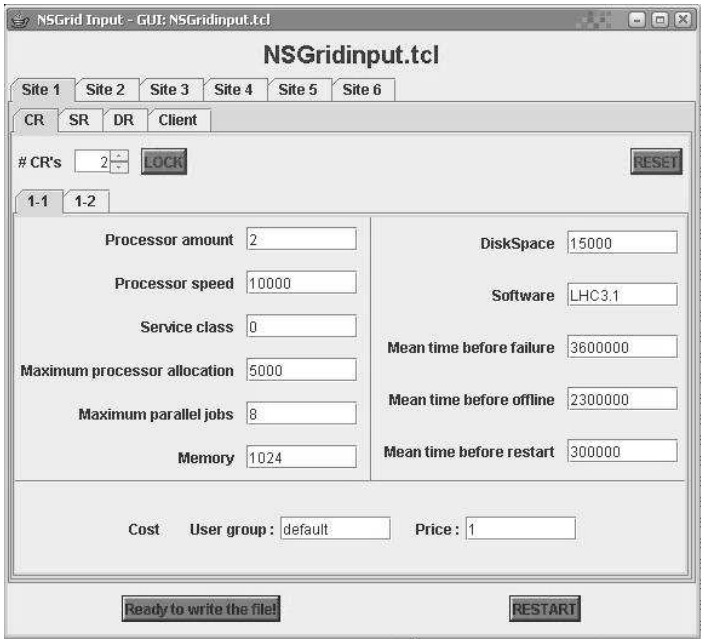


Figure 3.11: NSGrid Tcl input file generation

is optimized. In the optimal off-line case (i.e. the job requests that will arrive at the scheduler over time are known in advance), the schedule can in some cases (e.g. when job arrival times and job lengths are integer) be produced by solving an Integer Linear Program (we refer to [32] for an overview on this topic). In such an ILP, the scheduling constraints are formulated as linear (in)equalities.

Of course, in any operational Grid, only on-line scheduling strategies (meaning that at any time t, nothing is known about requests arriving at a time t' where $t' > t$) can be used. We distinguish between strategies that attempt to schedule a job as soon as it becomes available (ASAP), and strategies that schedule a batch of jobs at certain points in time (a batch then consists of previously unscheduled jobs and jobs that have arrived since the last batch was scheduled). ASAP strategies attempt to impose a minimal queue waiting time on each job, while batch strategies attempt to avoid suboptimal resource-to-job assignments by scheduling a set of jobs (instead of a single job) at the cost of extra waiting time for each job (at most equal to the interval between two scheduling rounds). However when scheduling in batch, one hopes to overcome this time loss by the possible gain created by an overall "better" scheduling decision.

In realistic scenarios, bursty job arrivals (e.g. high activity during office hours) will occur. We expect the quality of batch scheduling strategies, when compared to the schedule produced by strategies that schedule upon arrival, to rise when

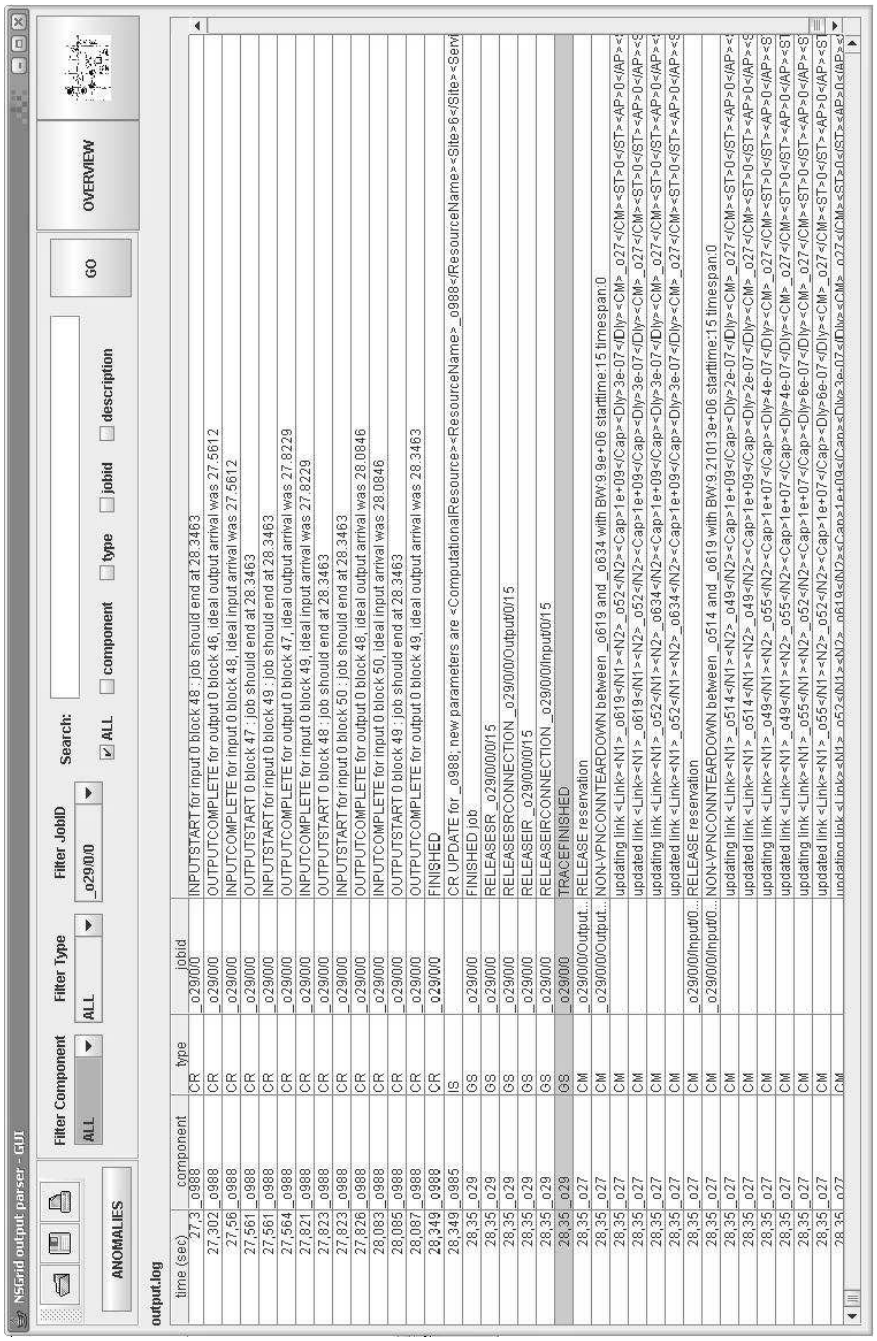


Figure 3.12: NSGrid output GUI

the average job arrival rate is higher than the rate at which resource status update information is disseminated through the Grid, while the scheduling rate is lower. If these assumptions hold, ASAP scheduling strategies can make wrong scheduling decisions for sequential jobs based on stale resource load information, resulting in an attempt to schedule a job on an already loaded resource.

In case batch scheduling is used, the (unscheduled) job queue can optionally be reordered prior to employing a scheduling algorithm (e.g. the job queue can be sorted based on service class priorities). Also, since batch scheduling can allocate resources to multiple jobs in one scheduling round (and resource reservation requests are only sent out once all jobs in the queue have been processed by the scheduling algorithm), the scheduler must keep track of "virtual" resource loads (i.e. the scheduler needs to modify resources' virtual state information each time a job gets resources allocated to it).

In [33], multi-site execution of divisible jobs is discussed. Jobs can be split into (communicating) subjobs which are then executed simultaneously on different computational resources. The network over which the subjobs communicate is not modelled directly; rather, it is assumed that the network's influence (bandwidth, delay) on the job's run time can be modelled by a single "overhead" parameter.

A similar job model is used in [34] and [35]. Here, the allocation of processors to rigid parallel applications on a purely space-shared (multi)cluster system is studied. Applications consist of a number of possibly communicating jobs, to be executed in parallel. Each job requires exactly one processor, which it occupies exclusively during its execution (i.e. no time-shared processors). Figures for the fraction of idle processors at a given point in time are deduced using statistical techniques, while the influence of a slow intercluster communication network is incorporated entirely in a slowdown factor α. This contrasts with our approach, as we study applications consisting of non-intercommunicating jobs, each of which can be executed on a single *time-shared* processor. In addition to computational resources, our work also treats other resources such as data and storage resources explicitly.

Scheduling work packets for collaborative computing efforts (e.g. SETI [36], MCell [37]) to computational elements is discussed in [38]. Because of the application's particular nature, the Grid can be modelled as a tree, with all work packets originating from the root node, which differs from our approach as we focus on generic Grids.

3.5 Scheduling algorithms

The scheduling algorithm used for assigning resources to jobs has a big impact on Grid performance, and influences overall job throughput, resource usage efficiency, average job response times, etc.. If the scheduler is unable to allocate the

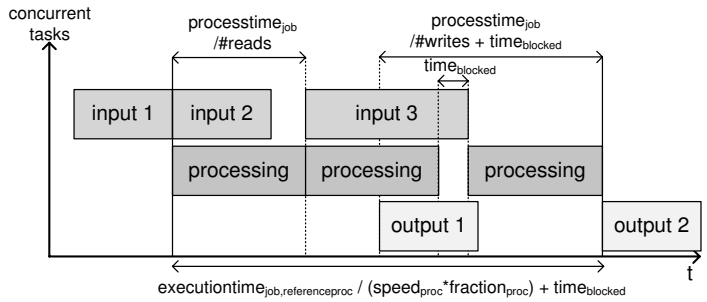


Figure 3.13: Job blocking on last input block

resources needed by a job, the job is queued for rescheduling in the next scheduling round (if an ASAP scheduling strategy is used, a rescheduling delay is imposed). The time between two scheduling rounds can be fixed, but it is also possible to set a threshold (e.g. maximum number of jobs in the scheduling queue) triggering the next scheduling round. In what follows we will explain the different scheduling algorithms available in NSGrid. During each scheduling round, every algorithm processes submitted yet unscheduled jobs from the job queue in a First-Come First Serve (FCFS) fashion (job reordering may have occurred as detailed in the previous section). Once a resource assignment has been made, our scheduler does not attempt to pre-empt jobs. All jobs run on a single processor which can be time-shared (i.e. serve multiple jobs simultaneously by allocating portions of its processing power to each such job). Intelligent allocation of these portions to jobs is necessary to prevent jobs from blocking when they depend on bandwidth-limited remote data access.

3.5.1 Network unaware

Network unaware scheduling will compute Grid job schedules based on the status of the computational, storage and data resources. Algorithms that use this kind of approach will not take into account information concerning the status of the network interconnecting these resources. The decision of which resources to use for a job will be based on the information acquired from the different information services (i.e. job execution speed and end time will be calculated based on the status of CR / DR / SR retrieved *for that job* from the different information services). In this case, our network unaware algorithm attempts to optimize execution time by minimizing the time a job spends on processing:

$$\frac{executiontime_{job,referenceproc}}{speed_{proc} * fraction_{proc}} \tag{3.1}$$

In this equation $executiontime_{job,referenceproc}$ is the execution time of a job on a reference processor, $speed_{proc}$ is the relative speed of a processor compared to the reference processor and $fraction_{proc}$ denotes the fraction of that processor that can be assigned to the job.

Because network unaware algorithms assume that residual bandwidth on network links is sufficient, job processing can block on input / output operations (see figure 3.13); their computational progress is no longer determined by the computational resource's processor fraction that has been allocated to it (which, together with the job's length and the computational resource's relative speed determines its earliest end time *if all I/O transfers complete on time i.e. before the start of the appropriate instruction block*), but rather by the limited bandwidth available to its I/O streams. Note that the fact that network information is discarded during the scheduling, implies no connection reservations (providing guaranteed available bandwidths) are made with the connection manager - these would allow to accurately predict the job's running time (see figure 3.14).

The time it takes for a job to complete since it has been submitted by the client can be broken up into:

- Sending the job to the scheduler
- Time spent in the scheduler's queue
- Time needed for the co-allocation of resources allocated to that job
- Transfer time for the first input data block(s)
- Time needed to process the job at its maximal execution speed
- Transfer time for the last output data block(s)
- Time during which the job is blocked on I/O operations

Pseudocode for the network unaware scheduling algorithm is shown in algorithm 3.5.1. For each job in the scheduling queue for which computational, storage and data resource queries have been answered by the information services, the scheduling algorithm inspects all possible resource triplets (CR,SR,DR) to see if they meet the job's requirements (this step is necessary because when scheduling in batch, virtual resource load changes - by already scheduled jobs in the same scheduling round - can make resources returned by the information services for a job, no longer meet the job's requirements; see section 3.4 for more information). If the job can be scheduled on the resource triplet, the algorithm calculates the time the job would spend processing on the selected computational resource and, if that time is less than the previously encountered optimum, this resource triplet is assigned the best resource triplet. Once the algorithm has checked all resource

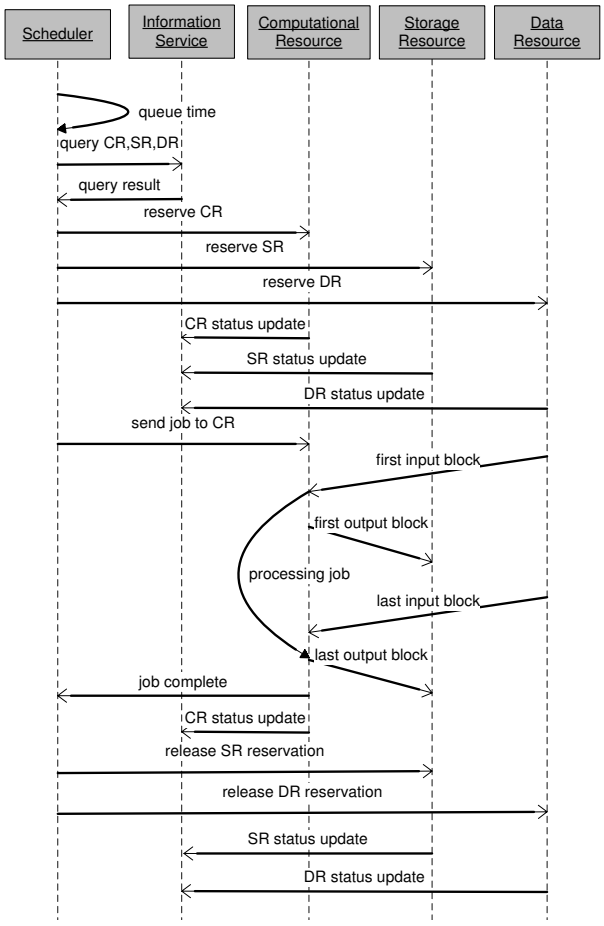


Figure 3.14: Network unaware scheduling

triplets, it selects the resource triplet that offered the best processing time (if one was found), schedules the job on those resources and updates those resources' virtual load information.

3.5.2 Network aware

Network aware scheduling algorithms will not only contact the information services for resources that adhere to the job's requirements, but will also query the connection manager for information about the status of the network links interconnecting these resources. In its turn, the connection manager will send the scheduler information about connections that can be set up between data / computational resource couples (necessary for job input retrieval) and computational / storage re-

Algorithm 3.5.1: NETWORK UNAWARE($Jobs$)

for each $j \in Jobs$
do {
  $BestResources \leftarrow [\,]$
  $LeastTime \leftarrow +\infty$
  for each $c \in CRs(j), s \in SRs(j), d \in DRs(j)$
  do {
    $Resources \leftarrow [c, s, d]$
    if $canSchedule(j, Resources)$
    then {
      $Time \leftarrow getTimeSpan(j, Resources)$
      if $Time < LeastTime$
      then {
        $LeastTime \leftarrow Time$
        $BestResources \leftarrow Resources$
      }
    }
  }
  if $BestResources \neq [\,]$
  then {
    $schedule(j, BestResources, LeastTime)$
    $updateResourceLoads(j, BestResources)$
  }
}

source couples (needed for job output storing)). Based on the answers from the information services and connection manager, the scheduling algorithm is able to calculate job execution speed and end time more accurately, by taking the speed at which I/O can be delivered to each available computational resource into account. For jobs with a single input stream and a single output stream, the best DR/CR/SR triplet is the one that minimizes the expected completion time of the job. This value is determined by the available processing power to that job on the computational resource (and its relative speed), the job's length, the job's total I/O data sizes and the residual bandwidth on the observed links from DR to CR and from CR to SR. Thus we are searching for the maximum speed at which the job can be processed, which (if we do not want blocking to occur) is the minimum of:

$\forall cr \in CR, \forall proc \in proccessors_{cr}, \forall dr \in DR, \forall sr \in SR$ calculate $speed_{proc} * fraction_{proc}$ with $speed_{proc} * fraction_{proc}$ the minimum of:

$$speed_{proc} \times fraction_{proc} \tag{3.2}$$

$$\frac{executiontime_{job,referenceproc} \times bandwidth_{dr,cr}}{8 \times blocksize_{input}} \tag{3.3}$$

$$\frac{executiontime_{job,referenceproc} \times bandwidth_{cr,sr}}{8 \times blocksize_{output}} \tag{3.4}$$

In these equations, $bandwidth_{dr,cr}$ is the bandwidth available between data and computational resource, $bandwidth_{cr,sr}$ denotes the bandwidth available between computational and storage resource, $blocksize_{input}$ is the block size of an input block and $blocksize_{output}$ denotes the block size of an output block. Equation 3.2 describes the rate at which computational processing of the job can occur (based

on CR properties returned by the information services and not taking into account I/O bandwidth restrictions), while equation 3.3 and equation 3.4 denote the speed at which processing can occur when limited by bandwidth from data resource to computational resource and from computational resource to storage resource respectively.

As explained, for some (DR/CR/SR) triplets, due to bandwidth constraints, this duration may be significantly higher than the value calculated from the job's length and the CR's relative speed, even if job execution and data transfer occur simultaneously. The scheduler selects the optimal DR/CR/SR triplet and contacts the connection manager to perform the necessary connection setups. The job then gets transferred to the selected CR for processing and I/O is sent from/to the DR/SR over the reserved connections. If neither local nor remote resources satisfying the job's requirements can be found, or if no connections with sufficient bandwidth are available, the job will be queued and prepared for rescheduling.

The time it takes for a job to complete since it has been submitted by the client can be broken up into:

- Sending the job to the scheduler
- Time spent in the scheduler's queue
- Time needed for the co-allocation of resources (including network resources) allocated to that job
- Network transfer time for the first input data block(s)
- Time needed to process the job at its allocated execution speed

$$\frac{executiontime_{job,referenceproc}}{speed_{proc} \times fraction_{proc}}$$

- Network transfer time for the last output data block(s)

Each of these can be found in figure 3.15. Note that no job can become blocked because of bandwidth reservations with the connection manager, excluding the network from becoming an unexpected bottleneck.

The pseudocode for the network aware scheduling algorithm is shown in algorithm 3.5.2. For each job in the scheduling queue for which computational, storage, data and network resource queries have been answered by the information services and connection managers, the scheduling algorithm inspects all possible resource triplets (CR,SR,DR) and associated interconnections to see if the resource triplet/connection combination meets the job's requirements (as already noted, this step is necessary because when scheduling in batch, virtual resource load changes can make resources returned by the information services/connection manager for

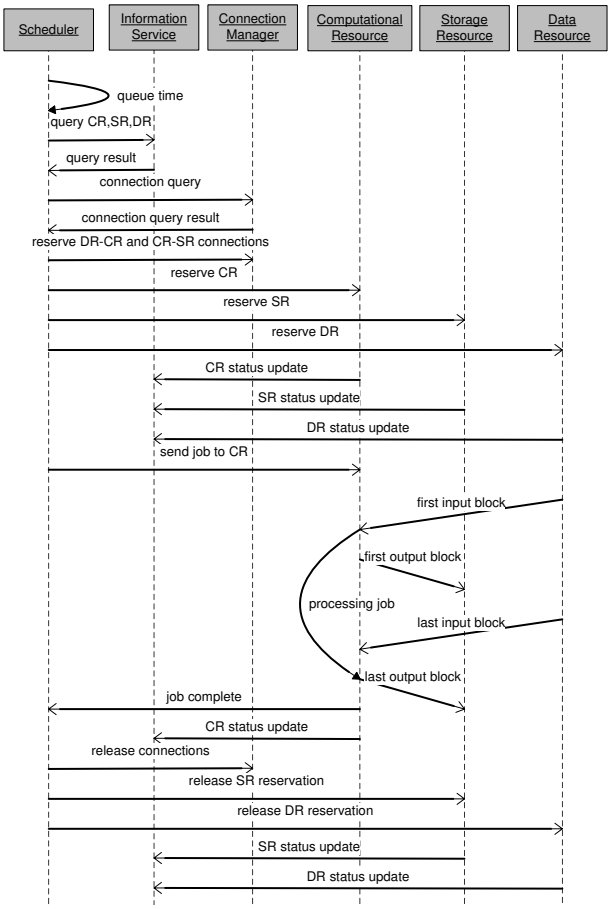


Figure 3.15: Network aware scheduling

a job, no longer meet the job's requirements). If the job can indeed be scheduled on the selected resources, the algorithm calculates the time during which the job can be processed (taking into account the speed at which input and output data blocks can be retrieved over the connections and the processing power available on the computational resource) and, if that time is less than the previously encountered optimum, temporarily stores this resource/connection combination as best scheduling solution. Once the algorithm has checked all resource triplets, it selects the resource triplet that offered the best processing time (if one was found), schedules the job on those resources and updates those resources' and their interconnections' virtual load information.

Algorithm 3.5.2: NETWORK AWARE($Jobs$)

for each $j \in Jobs$
do {
  $BestResources \leftarrow [\,]$
  $LeastTime \leftarrow +\infty$
  for each $c \in CRs(j), s \in SRs(j), d \in DRs(j)$
  do {
    $Conn \leftarrow getConn(c, s, d)$
    $Resources \leftarrow [c, s, d, Conn]$
    if $canSchedule(j, Resources)$
    then {
      $Time \leftarrow getTimeSpan(j, Resources)$
      if $Time < LeastTime$
      then {
        $LeastTime \leftarrow Time$
        $BestResources \leftarrow Resources$
      }
    }
  }
  if $BestResources \neq [\,]$
  then {
    $schedule(j, BestResources, LeastTime)$
    $updateResourceLoads(j, BestResources)$
  }
}

3.5.3 Resource locality preference

The "Local" scheduling heuristic is used solely for performance comparison. "Local" scheduling implies that a job submitted at a site's Grid portal, also needs to be processed on that site's computational, storage and data resources. Basically this means that each site executes its own jobs (the scheduler only queries the originating site's information services and thereby solely receives local resource information), and that no Grid functionality (namely remote processing and remote I/O retrieval/storage) is used in an attempt to overcome the site's restrictions. Both network aware and network unaware local scheduling heuristics are supported in NSGrid.

"PreferLocal" scheduling algorithms attempt to place a job on a site's local resources for processing, as we believe that, from an economic viewpoint, it can be assumed that remote resources are only used when necessary. When local processing is impossible (either because the job's requirements cannot be met locally, because the maximum computational load has been reached, or because I/O requirements are not met), the scheduler looks at the status of the remote resources and, if possible, selects a DR/CR/SR triplet (not necessarily all residing at one particular Grid site) meeting the job's requirements and prefers the triplet which allows for the earliest job end time (this job end time can be calculated in both a network aware or a network unaware fashion). The job is then transferred to the selected computational resource for processing and I/O is sent from/to the selected data and storage resource. If neither local nor remote resources satisfying the job's requirements can be found, the job gets queued for rescheduling during the next

scheduling round.

"Spread" algorithms do not prefer resources local to the job's originating site, but instead treat each Grid resource as equal.

3.5.4 Minimum hopcount

The "minimum hopcount" scheduling heuristic attempts to minimize the amount of network links (hops) input/output data needs to be sent over for a job in order to sparingly use available network resources. It chooses the DR/CR/SR triplet meeting the job's requirements and minimizing $hopcount(DR, CR) + hopcount(CR, SR)$ and does not take into account available network bandwidth on the links interconnecting the selected resource triplet.

Algorithm 3.5.3: MINIMUM HOPCOUNT($Jobs$)

for each $j \in Jobs$
do
- $BestResources \leftarrow [\,]$
- $MinHopCount \leftarrow +\infty$
- $Time \leftarrow +\infty$
- **for each** $c \in CRs(j), s \in SRs(j), d \in DRs(j)$
  - **do**
    - $Resources \leftarrow [c, s, d]$
    - **if** $canSchedule(j, Resources)$
      - **then**
        - $HopCount \leftarrow getHopCount(j, Resources)$
        - **if** $HopCount < MinHopCount$
          - **then**
            - $MinHopCount \leftarrow HopCount$
            - $Time \leftarrow getTimeSpan(j, Resources)$
            - $BestResources \leftarrow Resources$
- **if** $BestResources \neq [\,]$
  - **then**
    - $schedule(j, BestResources)$
    - $updateResourceLoads(j, BestResources)$

3.5.5 Service aware

A service aware heuristic dynamically classifies jobs in service class pools and attempts to schedule these service classes using an appropriate scheduling heuristic. Computationally intensive service classes are better of with a scheduling algorithm that focusses on allocating large and fast computational processor fractions than with a scheduling heuristic intent on optimizing network resource usage. Data intensive service classes on the other hand are better off with a scheduling heuristic that takes into account computational resource blocking times due to input/output

network delays. We refer to appendix B and appendix C for more information regarding service aware scheduling in Grids.

3.6 Simulation results

3.6.1 Simulation environment

All simulations performed in this section were run on a Mosix [39] cluster consisting of 14 AMD Athlon XP1700+ nodes with 1 GB RAM and Debian Woody [40] as operating system. NSGrid simulations were able to migrate to different cluster nodes depending on the load.

3.6.2 Simulated topology

A fixed Grid topology was used for the simulations presented here (topology and resources are depicted in figure 3.16). First, a Wide-Area Network (WAN) topology (containing 9 core routers with an average out-degree of 3) was instantiated using the *GridG* tool. Amongst the edge LANs of this topology, we have chosen 12 of them to represent a Grid site (each having its own computational, storage and data resource). Furthermore, we have homogenized the capacities of each WAN link, which we then treated as a parameter in our simulations. Each site has its own *information service* (storing resource properties and status) and local *Grid portal* (through which users can submit jobs). Local resources are connected through 1Gbps LAN links.

3.6.2.1 Job parameters

Two different job types were used in our simulations; one is more data-intensive (i.e. higher data sizes involved), while the other is more CPU-intensive. At each Grid site, two clients have been instantiated, one for each job type. Each client submits mutually independent jobs to its Grid portal. All jobs need a single data set stored on one of the data resources and write to a single storage resource. In the simulations where data is retrieved/stored in parallel with job processing, the number of blocks equalled 50. The ranges (uniformly distributed) between which the relevant job parameters vary have been summarized in table 3.2. Both job types make up 50 percent of the total job load; in each simulation, the job load consisted of 1200 jobs.

For each scheduling algorithm, we have chosen to use a fixed interval of $50s$ between consecutive scheduling rounds. From the arrival rates in table 3.2 (IAT) and the fact that multiple sites submit job simultaneously, it follows that we are likely to find multiple jobs in the queue at the start of each scheduling round.

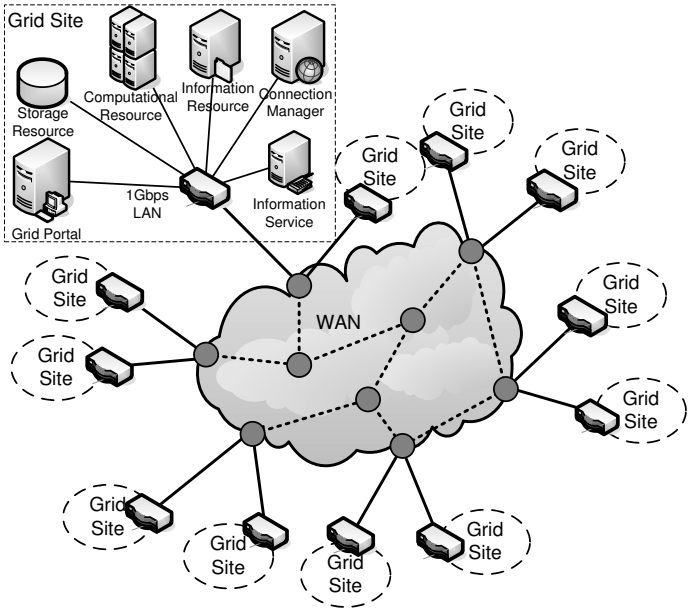


Figure 3.16: Simulated Grid topology

	CPU-Job	**Data-Job**
Input (GB)	0.01-0.02	1-2
Output (GB)	0.01-0.02	1-2
IAT (s)	100-200	100-200
Ref. run time (s)	400-1200	200-400

Table 3.2: Relevant job properties

3.6.2.2 Resource dimensions

We have assigned one computational resource to each Grid Site. To reflect the use of different tiers in existing operational Grids, not all computational resources are equivalent. The least powerful CR has two processors which operate at the reference speed. A second class of CRs has four processors, and each processor operates at twice the reference speed. The third - and last - CR type contains 6 processors, each of which operates at three times the reference speed. Conversely, the least powerful type of CR is three times as common as the most powerful CR, and twice as common as the middle one.

Since our focus is on determining the influence of the use of network resource status information on the optimality of the job schedule, we assumed that storage

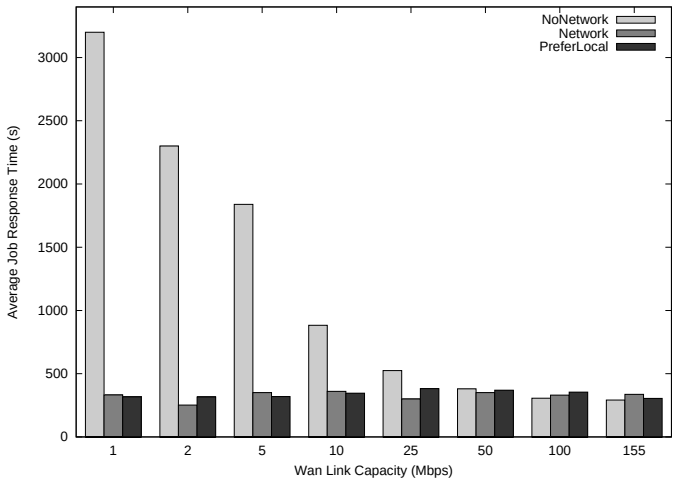


Figure 3.17: Job response time: parallel I/O

resources offer 'unlimited' disk space. Each site has at its disposal exactly one such storage resource.

Each site's data resource contains 6 out of 12 possible data sets. These data sets are distributed in such a way that 50 percent of the jobs submitted to a site can have local access to their needed data set.

3.6.3 Average job response time

We define the *response time* of a job as the difference between its end time and the time it is submitted to the scheduler.

In figure 3.17 we present this average job response time for algorithms we discussed earlier (network unaware spread scheduling, network aware spread scheduling and network aware preferlocal scheduling). In this particular simulation, simultaneous execution and data transfer were allowed (input was retrieved in 50 consecutive input blocks and output was stored in 50 consecutive output blocks); data connections were set up on a FCFS basis without upfront VPN dimensioning. Clearly, for low bandwidths, not taking the network status into account for schedule computation, incurs a severe penalty. When bandwidth grows, the importance of this network information degrades (for a constant job load) as the network no longer creates a bottleneck. In fact, for high bandwidths, it is possible for the network unaware algorithm to perform slightly better than the other algorithms; this is due to the conservative nature of our network-aware algorithms. For instance, these take for granted that the maximum data transfer rate is only 95 percent of the

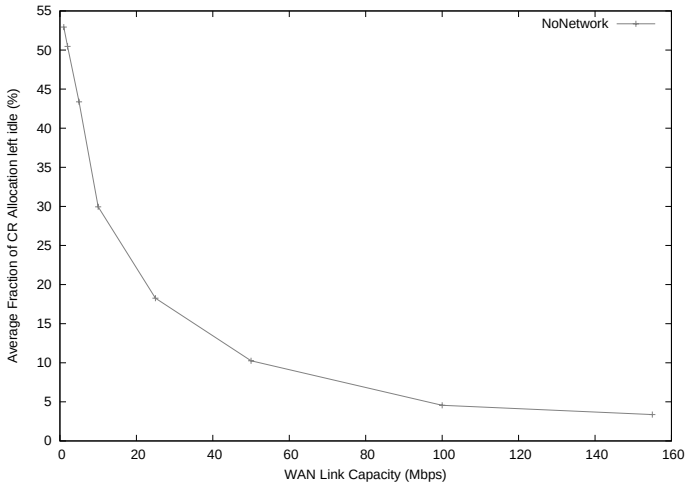


Figure 3.18: CR allocations: idle time

available bandwidth (i.e. 5 percent protocol overhead) and adjust their allocations accordingly.

In our simulations, no improvement is obtained by giving preference to local resources over remote ones. Intuitively, we expect better schedules using this strategy for data-intensive jobs as intra-site network links have high capacities. However, this improvement is neutralized by the asymmetry of the computational resources; jobs submitted at a site containing a slower CR, are now less likely to be executed on a faster one (which is of course the case if the best resource collection is selected for a job).

3.6.4 Computational resource idle time

If job execution and data transfer occur simultaneously, jobs can block, thereby inducing idle time on their time-shared CR within the processing power fraction allocated to that job. This happens if the job needs to wait for input data to arrive or output data to be written. Such a scenario is plausible when a network unaware scheduling algorithm is used; while available network bandwidth (in particular, between the job's CR and the DR providing it with input data) influences the minimum duration of a job on that CR, these algorithms do not take this bandwidth into account. This results in possible overallocation of the time-shared computational resource; a fraction of the resource is reserved uniquely for this job, but the job is unable to exploit its allocated computing power to its full extent. This means that - within its allocated fraction - a job induces idle time on the CR. Again, the

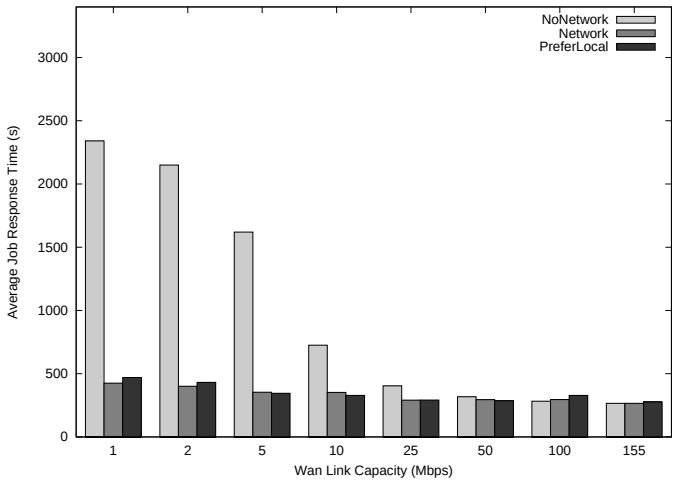


Figure 3.19: Job response time: pre-staged input

incurred penalty grows with lower bandwidths. Figure 3.18 shows the amount of idle time created by the network unaware algorithm in such cases.

In contrast, the network aware scheduling algorithms we discussed will be able to 'tune' their CR allocations with network bandwidth in mind, to ensure that no CR remains unnecessarily idle.

3.6.5 Influence of sequential data processing

In figure 3.19, we have plotted the average job response time again for the same job load. In this case however, jobs were not able to start execution while still downloading data, thus forcing the pre-staging of the entire input to the execution site. As the execution/transfer parallelism is lost, average response times for network aware algorithms increase (see also figure 3.17). However, this loss of parallelism does not influence the relative behaviour of the different algorithms (network aware or not) as discussed before. For low bandwidths, the network unaware algorithm produces better average response times when pre-staging data, as jobs cannot block their computational resource during processing in this case (rendering the computational resource utilisation more effective).

3.6.6 Influence of capacitated VPNs

If data connections are set up on demand using a pure FCFS scheme, it is likely that data-intensive jobs will quickly use up all of the available bandwidth, causing CPU-intensive jobs to remain queued for a longer period of time.

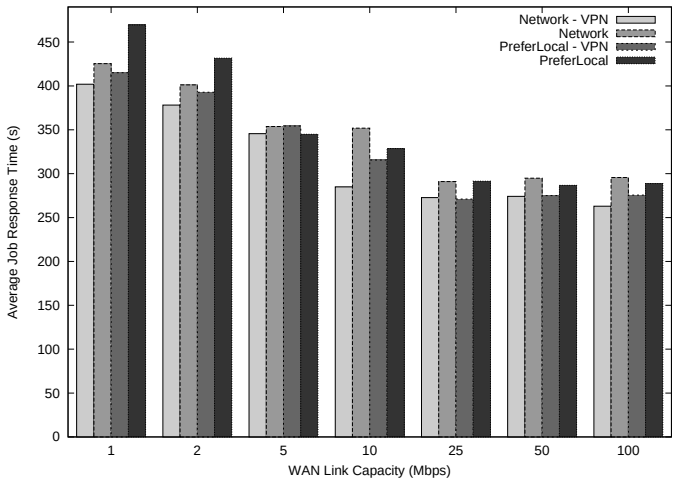


Figure 3.20: Job response time: upfront VPN reservations

The upfront reservation of bandwidth to each job type ensures that these CPU-intensive jobs will never be excluded from remote execution (i.e. process them on a faster CR). We have simulated the same job load, although this time VPNs were set up in advance for the two job classes (data-intensive vs. CPU-intensive). We reserved more bandwidth for the data-intensive jobs, using a $20 - 80$ percent ratio. The job response time for the different algorithms in this scenario is shown in figure 3.20. This approach visibly improves the response time; CPU-intensive jobs do not remain queued for an extraordinary long period of time, and as these jobs have long processing times, this has a significant impact on the average job response time.

Further improvement is possible if bandwidth is distributed more intelligently across the different job classes in a way that takes into account their respective processed data sizes and run times (see chapter 4).

3.7 Other simulations

We refer to appendix B for simulations concerning the use of application-specific hints in reconfigurable Grid scheduling algorithms. Using NSGrid we compare schedules that were produced by taking application-specific hints into account to schedules produced by applying the same strategy for all jobs (for both network aware as for network unaware scheduling). It is shown that when using application-specific hints in the scheduling process, average job response times in our simulated scenario improved by up to 30 percent, as some jobs are now

processed at a rate which is slightly lower than their preferred execution rate (as allowed by the hints), but finish sooner than if they were requeued for scheduling at a later time.

In appendix C we compare the performance of "network aware", "network unaware", "service" and "minhopcount" scheduling algorithms when scheduling cpu intensive and data intensive job classes (with dataset sizes between 15.6GB and 156GB per job - which can be expected of Grid jobs in the future) on a Grid topology in which sites are interconnected by means of high bandwidth ($2.5Gb/s$ - $5Gb/s$) optical links. It is shown that even when high-capacity network links are available, network unaware scheduling heuristics are outperformed by network aware scheduling algorithms in terms of average job response times and resource efficiency. Furthermore, if multiple application types are executed on the Grid, further improvements can be obtained by utilising service-oriented scheduling heuristics.

3.8 Conclusions

In this chapter we have presented NSGrid, a Grid simulator built on top of the ns-2 network simulator and capable of accurately modelling network traffic between different Grid resources. Computational, storage and data resource models were discussed, along with job (processing and I/O) models and the functionality & interoperation of the different management components: scheduler, connection manager, service manager, service monitor, information service and replication manager.

In order to demonstrate the usefulness of NSGrid, different Grid scheduling algorithms (some network aware while others network unaware) were detailed and their performance was evaluated on a sample Grid topology. The results show that whether data is pre-staged or accessed in parallel with the job's execution (i.e. streamed), accurate network status information allows to create significantly better schedules in terms of both job response time and computational resource efficiency. From our simulations, it follows that upfront reservation of bandwidth (between Grid resources) for different job types can improve the response time by avoiding that data-intensive jobs monopolize available bandwidth.

References

[1] *The Network Simulator - NS2*. `http://www.isi.edu/nsnam/ns`.

[2] L. Hall, A. Schulz, D. Shmoys, and J. Wein. *Scheduling To Minimize Average Completion Time: Off-line and On-line Algorithms*. In SODA: ACM-SIAM Symposium on Discrete Algorithms (Conference on Theoretical and Experimental Analysis of Discrete Algorithms), 1996.

[3] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. *Theory and Practice in Parallel Job Scheduling*. In Dror G. Feitelson and Larry Rudolph, editors, Job Scheduling Strategies for Parallel Processing, pages 1–34. Springer Verlag, 1997.

[4] Andrea Carol Arpaci-Dusseau. *Implicit coscheduling: coordinated scheduling with implicit information in distributed systems*. ACM Transactions on Computer Systems, 19:283–331, 2001.

[5] J. Liu, D.M. Nicol, B.J. Premore, and A.L.Poplawski. *Performance Prediction of a Parallel Simulator*. In Proc. of the Parallel and Distributed Simulation Conference (PADS'99), 1999.

[6] Jason Liu, L. Felipe Perrone, David M. Nicol, Michael Liljenstam, Chip Elliott, and David Pearson. *Simulation Modeling of Large-Scale Ad-hoc Sensor Networks*. In European Simulation Interoperability Workshop, 2001.

[7] A. Varga. *OMNeT++*. IEEE Network Interactive, 16(4), 2002.

[8] Andras Varga. *The OMNeT++ Discrete Event Simulation System*. In Proceedings of the European Simulation Multiconference (ESM'2001), 2001.

[9] Atsuko Takefusa, Satoshi Matsuoka, Henri Casanova, and Francine Berman. *A Study of Deadline Scheduling for Client-Server Systems on the Computational Grid*. In HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01), 2001.

[10] A. Takefusa, O. Tatebe, S. Matsuoka, and Y. Morita. *Performance Analysis of Scheduling and Replication Algorithms on Grid Datafarm Architecture for High-Energy Physics Applications*. In Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12), 2003.

[11] H. J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, Kenjiro Taura, and Andrew A. Chien. *The MicroGrid: a Scientific Tool for Modeling Computational Grids*. In Proc. of Supercomputing '00, 2000.

[12] Xin Liu, Huaxia Xia, and Andrew Chien. *Validating and Scaling the MicroGrid: A Scientific Instrument for Grid Dynamics*. Journal of Grid Computing, 2:141–161, 2004.

[13] *The Globus Alliance*. `http://www.globus.org/`.

[14] Arnaud Legrand, Loris Marchal, and Henri Casanova. *Scheduling Distributed Applications: the SimGrid Simulation Framework*. In CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid, 2003.

[15] J. Lerouge and A. Legrand. *MetaSimGrid : Towards realistic scheduling simulation of distributed applications*. ENS-LIP Research Report 2002-28, 2002.

[16] R. Buyya and M. Murshed. *GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing*. The Journal of Concurrency and Computation: Practice and Experience (CCPE), May 2002.

[17] Anthony Sulistio, Gokul Poduvaly, Rajkumar Buyya, and Chen-Khong Tham. *Constructing A Grid Simulation with Differentiated Network Service Using GridSim*. In Proc. of the 6th International Conference on Internet Computing (ICOMP'05), 2005.

[18] Hung-Ying Tyan and Chao-Ju Hou. *JavaSim: A component-based compositional network simulation environment*. In Proc. of Western Simulation Multiconference - Communication Networks And Distributed Systems Modeling And Simulation, 2001.

[19] John A. Miller, Andrew F. Seila, and Xuewei Xiang. *The JSIM Web-Based Simulation Environment*. Future Generation Computer Systems (FGCS), Special Issue on Web-Based Modeling and Simulation, 17:119–133, 2000.

[20] K. Ranganathan and I. Foster. *Identifying Dynamic Replication Strategies for a High Performance Data Grid*. In Proc. of the International Grid Computing Workshop, 2001.

[21] K. Ranganathan and I. Foster. *Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications*. In Int. Symposium of High Performance Distributed Computing, 2002.

[22] *Parsec : Parallel Simulation Environment for Complex Systems*. `http://pcl.cs.ucla.edu/projects/parsec`.

[23] William H. Bell, David G. Cameron, Luigi Capozza, A. Paul Millar, Kurt Stockinger, and Floriano Zini. *Simulation of Dynamic Grid Replication Strategies in OptorSim*. In GRID '02: Proceedings of the Third International Workshop on Grid Computing, pages 46–57, 2002.

[24] David G. Cameron, Ruben Carvajal-Schiaffino, A. Paul Millar, Caitriana Nicholson, Kurt Stockinger, and Floriano Zini. *Evaluating Scheduling and Replica Optimisation Strategies in OptorSim*. In 4th International Workshop on Grid Computing (Grid2003), 2003.

[25] *Java*. `http://java.sun.com/`.

[26] *The DataGrid Project*. `http://eu-datagrid.web.cern.ch/eu-datagrid/`.

[27] I. Foster K. Ranganathan. *Simulation Studies of Computation and Data Scheduling Algorithms for Data Grids*. Journal of Grid Computing, 1:53–62, 2003.

[28] *Tcl/Tk*. `http://www.tcl.tk`.

[29] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publication Company, 2000.

[30] D. Lu and P. Dinda. *Synthesizing Realistic Computational Grids*. In Proceedings of ACM/IEEE Supercomputing 2003 (SC 2003), 2003.

[31] D. Lu and P. Dinda. *GridG: Generating Realistic Computational Grids*. ACM SIGMETRICS Performance Evaluation Review, 40(4), 2003.

[32] L. Hall, A. Schulz, D. Shmoys, and J. Wein. *Scheduling To Minimize Average Completion Time: Off-line and On-line Algorithms*. In SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms), 1996.

[33] C. Ernemann, V. Hamscher, A. Streit, and R.Yahyapour. *Enhanced Algorithms for Multi-Site Scheduling*. In Proceedings of Grid2002, LNCS 2536, 2002.

[34] A.I.D. Bucur and D.H.J. Epema. *An Evaluation of Processor Co-Allocation for Different System Configurations and Job Struxctures*. In Proceedings of SBAC-PAD, 2002.

[35] A.I.D. Bucur and D.H.J. Epema. *The Influence of the Structure and Sizes of Jobs on the Performance of Co-Allocation*. In Proceedings of JSSPP6, 2000.

[36] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. *SETI@home: An Experiment in Public-Resource Computing*. Communications of the ACM, 45:56–61, 2002.

[37] H. Casanova, T. Bartol, J. Stiles, and F. Berman. *Distributing MCell Simulations on the Grid*. The International Journal of High Performance Computing Applications, 14:243–257, 2001.

[38] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. *Bandwidth-centric allocation of independent tasks on heterogeneous platforms*. Technical report, Technical Report 4210, INRIA, 2001.

[39] *MOSIX Cluster and Grid management*. `http://www.mosix.org/`.

[40] *Debian*. `http://www.debian.org/`.

4

Grid Service Management

4.1 Introduction

In chapter 2 we discussed the importance of accurately monitoring Grid state and presented a scalable Grid monitoring architecture capable of outperforming the current leading Grid monitoring platforms. We continued in chapter 3 by describing NSGrid, a Grid simulator capable of modelling a large variety of Grid topologies and resource configurations (providing network, computational, data and storage resource models along with models for various management components and jobs). The work presented in this chapter combines these efforts and discusses how monitored job, application and resource characteristics can be put to use by a Grid service management architecture to improve among others Grid management scalability and resource efficiency. NSGrid allows us to easily evaluate the effectiveness of different service management algorithms on a wide range of Grid configurations.

As more and more application types are ported to Grid environments, an evolution is noticed from purely computational and/or data Grid offerings to full-scale service Grids [1] (e.g. the Enabling Grids for E-Science in Europe (EGEE) project [2]). A 'service Grid' denotes a Grid infrastructure capable of supporting a multitude of *application types* with varying QoS levels (i.e. our definition of service Grid is not limited to web-service enabled Grids). We use the term 'service class' as a classifier for user-submitted Grid jobs that exhibit similar resource requirements (processing requirements, I/O data requirements, priority, etc.). The

architectural standards for service Grids are provided by the Global Grid Forum's Open Grid Service Architecture (OGSA) [3], and (to a lesser extent) the Web Service Resource Framework [4], building on concepts of both Grid and Web Service communities.

Widespread Grid adoption also increases the need for automated distributed management of Grids, as the number of resources offered on these Grids rises dramatically (hence the scalability of these Grids becomes very important). Automated self-configuration and self-optimization of Grid resource usage can greatly reduce the cost of managing a large-scale Grid system, and at the same time achieve better resource efficiency, scalability and QoS support [5, 6].

The distributed service management architecture proposed in this chapter can be described as a distinct implementation of the OGSA 'Service Level Manager' concept. Service Level Managers are, according to the OGSA specification, responsible for setting and adjusting policies, and changing the behaviour of managed resources in response to observed conditions.

Our main goal is to automatically and intelligently assign Grid resources (both network, computing and data/storage resources) to a particular service class for exclusive use during a specified time frame (i.e. partitioning the pool of Grid resources into distinct service class-assigned resource pool subsets). The decision to assign a resource to one particular service will be based on the resources available to the Grid and monitored service class resource usage characteristics and requirements. Once resource partitioning has been performed, dedicated management components (i.e. scheduler, information service, etc.) will be associated to a service class's assigned resources, effectively constructing multiple self-managing 'Virtual Private Grids'. These Virtual Private Grids in turn improve Grid management scalability, as their management components only need to take into account the state of their partition-assigned resources along with the state and requirements of jobs from the service class they are responsible for.

In order to compare the performance of a service managed Grid versus a non-service managed Grid we use NSGrid (detailed in chapter 3). More specifically, we evaluated Grid performance (in terms of average job response time and resource efficiency) when different partitioning strategies are employed, and this both in case network aware as when network unaware scheduling is used.

This chapter is structured as follows: section 4.2 summarizes related work in this area, while section 4.3 continues with an overview of the service management architecture and its interaction with other Grid components. Section 4.4 elaborates on the different resource partitioning strategies, while the evaluation of those partitioning strategies in a typical Grid topology is compared to a non-resource partitioned situation for varying job loads in section 4.5. Finally, section 4.6 presents some concluding remarks.

4.2 Related work

Considerable work has already been done in the area of distributed scheduling for Grids [7]. Grid scheduling taking into account service specific requirements has been dubbed *application-level scheduling*. Most notable application-level research projects include AppLeS [8] and GrADS [9].

In AppLeS, service-class scheduling agents interoperable with existing resource management systems have been implemented. Essentially, one separate scheduler needs to be constructed per application type. Our service management architecture differs from this approach in that it operates completely separated from the Grid scheduling components, working in on service-exclusivity properties located at the information services (responsible for storing resource properties and answering resource queries from e.g. the different schedulers).

GrADS on the other hand is a project to provide an end-to-end Grid application preparation and execution environment. Application run-time specific resource information comes from the Network Weather Service [10] and MDS2 [11]. For each application; a performance (i.e. computational, memory and communication) model needs to be provided by the user. This differs from our service monitor approach, which actively monitors application behaviour and deduces service characteristics at run-time (see section 4.3.2).

The General purpose Architecture for Reservation and Allocation (GARA) project [12] provides Globus with end-to-end Quality of Service guarantees for applications. Both advance and immediate resource reservations are supported. GARA does not offer dynamic automated resource-to-service partitioning but can instead be seen as a technology enabling the work proposed in this chapter.

IBM's Tivoli Intelligent Orchestrator (TIO) and Provisioning Manager (TPM) [13] can improve service response times by monitoring registered resources and requirements for anticipated peak workloads and, if necessary, can automatically re-allocate resources in accordance with business priorities. TIO and TPM are focused on automated data center resource-to-service allocations, and require users to predefine 'optimal resource utilization' plans for each supported service class. Our service management architecture focuses on the needs of generic computational / data / service Grids, and tries to automatically (i.e. without user interaction) deduce optimal resource utilization from monitored Grid job submission behaviour.

Optimally assigning resources to services has been the subject of research in [14]. In this study however, resource selection occurs each time a job is submitted to a Grid portal (i.e. service aware scheduling). This differs from the work proposed in this chapter in which resources are pre-assigned to service classes based on service class characteristics (i.e. prior to the job scheduling process).

In contrast to the above mentioned research projects, our contribution focuses

on distributed, automated and intelligent resource-to-service partitioning in a Grid environment (based on monitored service class characteristics/requirements) along with the dynamic deployment of service class exclusive management components (effectively constructing multiple Virtual Private Grids).

4.3 Service management concept

Recall that we regard a Grid as a collection of Grid sites interconnected by WAN links. Each Grid site has its own resources (computational, storage and data resources) and a set of management components, all of which are interconnected by means of LAN links. Every Grid resource in our model is given a service class ID property (stored in the information service with which the resource is registered) which denotes the service class the resource is associated with. If no service management components are instantiated in the Grid, all resources' service class ID equals '0', meaning these resources can be used by *any* job (i.e. belonging to *any* service class). If however a resource's service class ID is set to the ID of one particular service class, only jobs from that particular service class will be able to *start* utilising that resource (note that we say *start* since jobs already assigned to a resource the moment a change in that resource's service class ID occurs, can continue utilising that resource).

4.3.1 Resource-to-service partitioning

Our goal is to intelligently and automatically assign service class IDs to each resource so they can be used exclusively for jobs spawned from that service class. This classification of Grid resources in a per-service resource pool with its own dedicated scheduler and information service has multiple benefits:

- resource efficiency and average job response times improve (as will be shown in section 4.5)
- allows for faster scheduling decisions and resource information lookups
- service class priorities can be given by assigning more resources to high-priority service classes
- locally offered service classes can be prioritized over foreign Grid site service classes
- reduced infrastructure costs: by allocating job loads to resources more efficiently, the number of resources can be reduced
- improved scalability with dynamic deployment of dedicated VPG management components

- support for new business models
- service class dedicated management components can be finetuned to the needs of their particular service
- security can be enforced on a service class's resource pool basis

As we will see in section 4.5, resource efficiency (and average job response times) can be improved by limiting resource availability to service classes that can make efficient use of that particular resource (e.g. taking into account service class data locality). Note that resource efficiency describes the amount of time a resource is reserved *and* this reservation is fully utilised - as opposed to resource reservations where reservation time is spent on idling while waiting for job I/O data to arrive. In addition, the number of job resource query results returned by the information services to the scheduler will be less than when there is one common resource pool, allowing for faster scheduling decisions (as we are in fact utilising the resources' service class ID assignation as an advance reservation mechanism).

Of course, one has to be very careful when automatically assigning resources to service classes, as it creates the risk that certain service classes are (involuntarily) left starving for resources on which to run, while other resources are assigned to a service class for which there are no job submissions at that time (and are thus unnecessarily left idle). One also has to take into account service class necessities when making resource partitioning decisions, in order to avoid excluding a service class from access to a critical resource (e.g. prohibiting a service class access to mandatory data resources).

The same way computational, storage and data resources can be partitioned amongst different service class resource pools, network resources can also be split up by performing per-service bandwidth reservations (e.g. VPN technology). This can prevent data-intensive service classes from monopolising network bandwidth usage and thereby hampering the performance of jobs from other service classes. Instead, each service class should automatically receive a certain bandwidth and be able to use this bandwidth without having to worry about the network usage of other services' jobs.

With combined network and resource partitioning, a Grid can be modelled as a dynamic collection of overlay Grids or *Virtual Private Grids* (VPG), with one VPG for each service class offered in the Grid (see figure 4.2). These VPGs are not static structures in that they do not have resources assigned to them in a permanent way, but react to monitored changes in service characteristics (e.g. additional service offerings can lead to the construction of new VPGs and reallocation of resources across existing VPGs). Resource reallocation can stem from important changes in monitored service class characteristics (e.g. higher job submission rates for a service class), a change in service class priorities or, as already mentioned, the addition of new service classes.

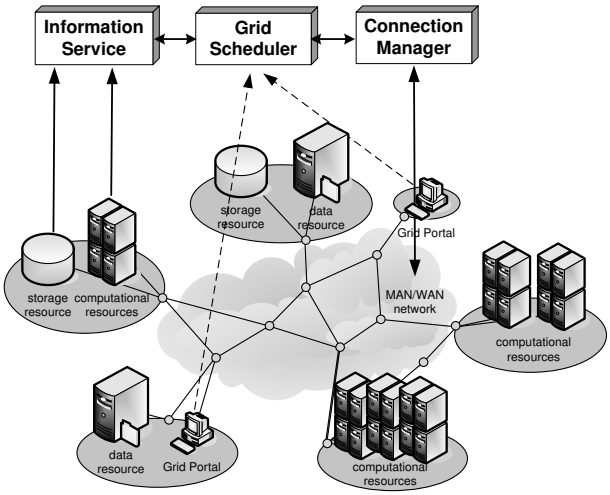


Figure 4.1: Standard Grid

4.3.2 NSGrid implementation

In NSGrid, a distributed service management architecture was implemented in order to evaluate the effectiveness of different resource-to-service partitioning strategies and Virtual Private Grid deployments. Each Grid site can have a local *service manager*, which interacts with the local information service (IS), connection manager (CM) and *service monitor*.

4.3.2.1 Service monitor

The service monitor inspects job submission behaviour at the Grid portals (recall that a Grid portal acts as a job submission gateway for Grid users): each time a job is submitted, job requirements (service class, priority, needed input data sets and sizes, output storage sizes, computational requirements, etc.) are extracted and overall service class properties (e.g. average job interarrival time, average I/O data sizes, average job computational needs, needed input datasets) are adjusted. When the service monitor has gathered adequate service class characteristics (either when service class properties remain relatively stable over a fixed period of time, or when an information dissemination timer has run out), the service monitor sends the collected service class' characteristics to its known (local and foreign Grid site) service managers, so as to allow them to have up-to-date service class information for use by the resource-to-service partitioning algorithms. The service monitor keeps a record of the info that was submitted to the service managers, and, if substantial changes (w.r.t. a configurable threshold) in service class properties are monitored (e.g. detection of new service classes, increased service class job

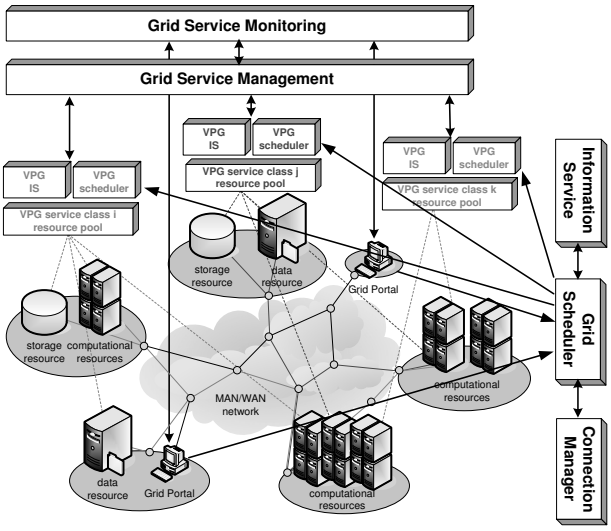


Figure 4.2: VPG partitioned Grid

interarrival times, change in priority, higher job response times, etc.), sends up-to-date service class information to the service managers (see figure 4.3).

Each service monitor has a moving time window (of configurable length), such that the properties of a job that was submitted at a time before the time window's beginning are no longer taken into account when calculating service class' characteristics. In doing so, service classes that spawn no jobs during a period of time equal to the time window's length are discarded: the service monitor will inform the service manager of this occurrence, which in turn will free resources allocated to that particular service class and (if necessary) repartition.

Listing 4.1 shows the most important public methods (not including constructors/destructors and most getters/setters) supported by the service monitors. In what follows we give a brief description of these methods:

- *void setBeginPartitionTime(...)*: sets time at which collected service class information needs to be sent to the service managers that are interested in it (i.e. information dissemination timer). Note that the service monitor can choose to distribute service class characteristics at an earlier time if these characteristics remain relatively stable over a period of time.

- *void setNewServiceJobNr(...)*: sets number of jobs (belonging to a new service class) that need to be monitored before the service monitor informs the service managers of the existence of a new service class.

- *void setProcDiffIAT(...)*: sets percentual service class' interarrival time dif-

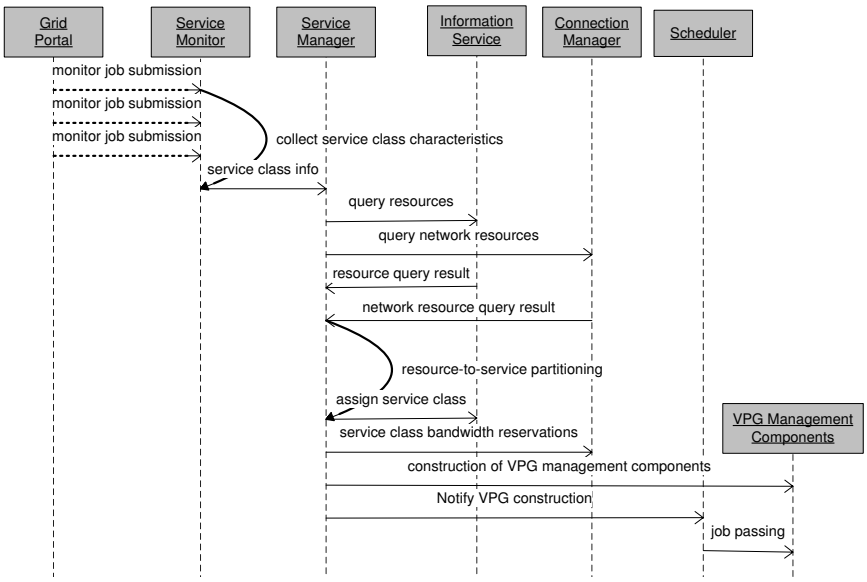


Figure 4.3: VPG partitioning messages

ferentiation that can be monitored before the service monitor informs the service managers of this change in service class' characteristics.

- *void setProcDiffReq(...)*: sets percentual service class' processing requirements differentiation that can be monitored before the service monitor informs the service managers of this change in service class' characteristics.
- *void setServiceMonitorWindow(...)*: sets length of service monitoring window.
- *void setProcDiffInput/Output(...)*: sets percentual service class' input/output requirements differentiation that can be monitored before the service monitor informs the service managers of this change in service class' characteristics.
- *void monitorJob(...)*: job submission monitored by the service monitor. The service monitor extracts information from the job's XML description and updates the job's service class' characteristics.
- *void addInitialServiceClass(...)*: utilised in case one wants to seed the service monitor with initial service class' characteristics.
- *void addServiceManager(...)*: adds a service manager that will be contacted by the service monitor when distributing service class' characteristics.

Listing 4.1: Service monitor interface

```
class ServiceMonitor : public ManagementComponent {
public:
    void setBeginPartitionTime(double beginPartitionTime);
    void setNewServiceJobNr(int newServiceJobNr);
    void setProcDiffIAT(double procDiffIAT);
    void setProcDiffReq(double procDiffReq);
    void setServiceMonitorWindow(double serviceMonitorWindow);
    void setProcDiffInput(double procDiffInput);
    void setProcDiffOutput(double procDiffOutput);

    void monitorJob(string jobXML);
    void addInitialServiceClass(int serviceClass, double MIReq, double inputReq, double outputReq
            , int priority, double IAT);
    void addServiceManager(string serviceManager);
};
```

4.3.2.2 Service manager

The service manager thus periodically receives information regarding local and foreign Grid site service class characteristics from the different service monitors. When the received information does not differ (with regard to a certain threshold) from the one used to partition the Grid resources in a previous partitioning run, no resource-to-service repartitioning will occur. If however the difference between the previous values and currently monitored service characteristics (average job IAT, processing length, I/O bandwidth necessities, etc.) is too large, or if no resource partitioning has yet been done, the service manager will query the information services for the characteristics of the resources in their local Grid site resource pool. Once the answer to this query has been received, one of the resource partitioning algorithms (detailed in section 4.4) is applied to the resource set, and the resulting resource partitioning solution is sent back to the information services, who in turn change the service class property of their registered resources. If the partitioning algorithm also works in on network resources, the connection manager will be contacted to make service bandwidth reservations (based on assigned computational resources, necessary input datasets and monitored service class' bandwidth requirements).

Once the partitioning algorithm has finished, resources will be assigned to service class resource pools, and (if this was not already done) dedicated Virtual Private Grid management components will be dynamically constructed and associated with the different Virtual Private Grids (in NSGrid these VPG management components are deployed at the Grid site where jobs from the VPG's service class are most common). A VPG information service will gather resource property and status information from all resources assigned to the VPG. This information service will in turn be queried by a dedicated VPG scheduler when the latter seeks

information on resources adhering to a job's requirements. Note that the global (central or distributed) Grid scheduling system continues to receive all jobs submitted to the different Grid portals, but, upon inspection of the service class of each arriving job, either tries to schedule the job itself, or, when a VPG is constructed for the job's service class, immediately sends it to the dedicated VPG scheduler.

Listing 4.2: Service manager interface

```
class ServiceManager : public ManagementComponent {
public:
    void addLocalServiceClass(int serviceClass, int submissionSites, double CPUReq, double
            inputReq, double outputReq, int priority, double IAT);
    void removeLocalServiceClass(int serviceClass);
    void addForeignServiceClass(string foreignServiceInfo);
    void removeForeignServiceClass(int serviceClass);
    void addLocalServiceClassIDNeed(int serviceClass, int ID);
    void addForeignServiceClassIDNeed(int serviceClass, int ID);

    void addInformationService(string informationService);
    void addServiceManager(string serviceManager);
    void addGridScheduler(string gridScheduler);
    void addConnectionManager(string connectionManager);

    void setPartitioningStrategy(string partitionStrategy);
    void setForeignPolicy(double foreignPolicy);

    void retrieveResources();
    void retrieveConnections();

    void boostPriority(int serviceClass, int boost);

    void submitServiceInfo(string serviceInfo);
    void deployVPGS();
};
```

Listing 4.2 shows the most important public methods (not including constructors/destructors and most getters/setters) supported by the service manager components. In what follows we give a brief description of these methods:

- *void add/removeLocalServiceClass(...)*: adds/removes local service class characteristics.
- *void add/removeForeignServiceClass(...)*: adds/removes foreign service class characteristics.
- *void addLocal/ForeignServiceClassIDNeed(...)*; adds data set ID frequently needed by a local/foreign Grid site service class.
- *void addInformationService/ServiceManager/GridScheduler/ConnectionManager(...)*: adds management component for querying/updating.

- *void setPartitioningStrategy(...)*: allows initialising/changing the employed resource-to-service partitioning strategy.
- *void setForeignPolicy(...)*: allows changing foreign service class policy ($\rho_{SC_{foreign}}$ as explained in section 4.4.1.2)
- *void retrieveResources()*: queries the information services for resource properties.
- *void retrieveConnections()*: retrieves network resource information from the connection managers.
- *void boostPriority(...)*: boosts the priority of a particular service class.
- *void submitServiceInfo(...)*: submits service class information (i.e. when service monitors have monitored a change in service class' behaviour).
- *void deployVPGS()*: deploys Virtual Private Grid management components and configures them accordingly.

4.3.2.3 Information service

Much in the same way as the service monitors can trigger a repartitioning of resources to services when substantial changes in service class characteristics are monitored, the information services are responsible for signalling changes in resource availability. Every time an existing Grid resource becomes unavailable (either because of failure or by policy), or conversely, when new resources become available to the Grid, the information services report this change to the service manager. The latter then decides if a resource-to-service repartitioning is necessary.

It is important to note that, while resources are assigned for exclusive use by a particular service, not one job using a service class reassigned resource will be interrupted (preventing jobs from being pre-empted when the CR it is running on is assigned to a different service class). The service assignment will thus only be effective for *new* jobs or jobs currently in the scheduler queue. At the time of scheduling, queries will be sent to the information services for resources adhering to the job's requirements, and these information services will return only those resources that are assigned to that particular job's service class.

On another note, we have limited our research to discrete resource-to-service allocations (i.e. a single service class assigned to a resource; note that in section 4.4.1.4 we allow network resources to be partitioned amongst multiple service classes, to prevent parts of the Grid topology of becoming isolated when network partitioning is performed). More advanced resource-to-service partitioning algorithms could allow resources to be reserved for a subset of the available service

classes (i.e. multiple service class IDs can be assigned to a single resource). This way, resources (or resource fractions) could be members of multiple Virtual Private Grids, allowing for stacked VPGs.

4.4 Partitioning strategies

Recall that we are trying to partition resources into service class resource pools. A solution in this case is a mapping from resource to a particular service class ID, and this for all resources returned from the service manager - information service queries. A resource can also be assigned service class ID '0', meaning it can be used by jobs from every service class. Exhaustively searching for an optimal partitioning (by evaluating the fitness of a solution by means of a cost function) quickly becomes infeasible, as the amount of solutions that needs to be evaluated is $(\#serviceclasses + 1)^{\#resources}$. In our attempts to find a suitable solution in reasonable time, we have used Genetic Algorithm based heuristics to obtain a resource-to-service mapping. Note that Pieter Thysebaert has developed a heuristic based on Divisible Load Theory (DLT [15]) for tackling the resource-to-service partitioning problem. For a thorough discussion of this heuristic we refer to [16].

In order to provide a better understanding of the results of employing one of the different resource-to-service partitioning heuristics discussed in the next sections, we introduce a sample Grid with 3 Grid sites in figure 4.4. We introduce two service classes: the first (SC 1) is cpu-intensive, needs access to the data set with ID 1 and can run at four times the reference processing speed when receiving/sending I/O 100Mbps. The second service class (SC 2) is data-intensive, needs access to the data set with ID 2 and can run at twice the reference processing speed when receiving/sending I/O at 1Gbps. Both service classes account for 50 percent of the job load and have equal priority. As shown in table 4.1, the first and second Grid site launch jobs from service class 1 and 2, while the third Grid site only launches jobs from service class 2. We have provided site 1 with 2 fast computational resources (who can run jobs at 4 times the reference speed) and 1 slow computational resource (running at the reference speed), while site 2 and 3 have a single fast CR and 2 slow CRs. Each Grid site has a client portal, information service, connection manager, service monitor and service management component, while a single *network aware* Grid scheduling component is instantiated (the different management components are not shown to avoid cluttering the figure).

If no resource-to-service partitioning occurs (see figure 4.4), the scheduler will assign both service class 1 and 2 jobs to the fastest computational resources, as these resources provide for both service classes the fastest processing speeds. However, once these fast CRs are fully loaded (with both SC1 and SC2 jobs), the scheduling algorithm will have to assign computationally intensive SC1 jobs

	site 1	site 2	site3
fast CRs (4x ref.speed)	2	1	1
slow CRs (ref.speed)	1	2	2
Service classes	1-2	1-2	2
Data sets	1	2	2

Table 4.1: Sample Grid site properties

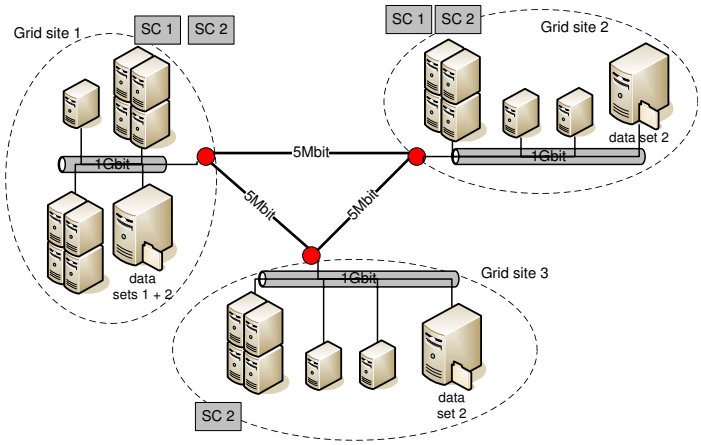


Figure 4.4: Grid example - no resource-to-service partitioning

to CRs with low processing speeds (processing of these SC1 jobs on the slow CRs will be four times slower than on a fast CR). A better approach would have been to assign the data-intensive SC2 jobs to those slower computational resources that have local access to dataset 2, keeping the fast computational resources fully available for processing jobs from the computationally intensive service class. The latter behaviour is exactly what the resource-to-service partitioning heuristics will try to enforce.

Note that due to the use of a Genetic Algorithm different solutions can be found when employing one of our resource-to-service partitioning algorithms on the described problem. Our example will only highlight one of those possible solutions.

4.4.1 Genetic Algorithm heuristics

The resource class assignment can easily be encoded into an n-tuple of service class IDs, where n equals the number of resources. These *chromosomes* can then be fed to a Genetic Algorithm (GA) which evaluates the fitness of each chromosome (i.e. possible service class assignment) w.r.t. a cost function $f(x)$ (see algo-

rithm 4.4.1). Different cost functions will be described in section 4.4.1.1, 4.4.1.2 and 4.4.1.3.

Algorithm 4.4.1 starts with an initial population size of m randomly generated tuples (each tuple b consisting of n service class ID slots). While the stop-condition is not fulfilled, the GA applies a proportional selection, after which a two-point crossover and a mutation step occur. The proportional selection selects tuples based on their fitness (with fitter solutions more likely to be selected and carried over to the next generation). In the next step, a two-point crossover operation is applied (for each two consecutive tuples the crossover probability ρ_C determines if all service class IDs between the randomly selected $pos1$ and $pos2$ are switched). Finally, the mutation operation is performed for each tuple, with mutation probability ρ_M determining which of the n service class ID slots needs to be mutated to a random service class ID.

Depending on how much time is available between partitioning runs (which in turn depends on the stability of the different service characteristics), parameters of this GA can be tuned in such a way that feasible search times can be attained (i.e. search time $<<$ time between partitioning runs).

In the next sections we provide details on some implemented partitioning strategies (and accompanying cost functions $f(x)$): section 4.4.1.1 and section 4.4.1.2 describe computational resource partitioning based on the processing requirements of respectively local and global service classes. Taking into account the site locality of much needed service class' input datasets is discussed in section 4.4.1.3. Finally, partitioning of network resources based on data requirements of the different service classes is discussed in section 4.4.1.4. We assume that the service manager has received both up-to-date local and foreign Grid site service characteristics from the service monitors and resource properties from the information services.

4.4.1.1 Local Service CR partitioning

The first (and simplest) partitioning strategy only takes into account the computational processing needs and priority of the different *local* service classes. The service manager queries the information services for all local computational resources and calculates average service class' requested processing power as the average processing time of that service class (as measured on a CR running at reference speed) divided by the average interarrival time of that SC (the higher job interarrival times, the less processing power will be needed) and multiplied with the number of sites that submit jobs from this SC.

$$\forall SC \cdot ppower_{req_{SC}} = sites_{SC} \times \frac{ptime_{ref_{SC}}}{IAT_{SC}}$$

Algorithm 4.4.1: GENETIC ALGORITHM($resources$)

$population_{initial} \leftarrow (b_{(1,0)}, ..., b_{(m,0)}),\ t \leftarrow 0$
while $stopcondition$ **false**
do
- **comment:** proportional selection
- **for** $i \leftarrow 1$ **to** m
  - **do**
    - $x \leftarrow rand[0,1]$
    - $k \leftarrow 1$
    - **while** $k < m$ **and** $x < \sum_{j=1}^{k} \frac{f(b_{j,t})}{\sum_{j=1}^{m} f(b_{j,t})}$
      - **do** $k \leftarrow k + 1$
    - $b_{i,t+1} \leftarrow b_{k,t}$
- **comment:** two-point crossover
- **for** $i \leftarrow 1$ **to** $m - 1$ **step** $i + 2$
  - **do**
    - **if** $rand[0,1] \leq \rho_C$
      - **then**
        - $pos1 \leftarrow rand[1,n]$
        - $pos2 \leftarrow rand[1,n]$
        - **if** $pos1 > pos2$
          - **then** $switch(pos1, pos2)$
        - **for** $k \leftarrow pos1$ **to** $pos2$
          - **do** $switch(b_{i,t+1}[k], b_{i+1,t+1}[k])$
- **comment:** mutation
- **for** $i \leftarrow 1$ **to** m
  - **do**
    - **for** $k \leftarrow 1$ **to** n
      - **do**
        - **if** $rand[0,1] < \rho_M$
          - **then** $b_{i,t+1}[k] \leftarrow rand[0, \#SC]$
- $t \leftarrow t + 1$

In this equation, $ptime_{ref_{SC}}$ is the average processing time of a service class SC job on a reference computational resource, $sites_{SC}$ denotes the amount of Grid portals launching service class SC's jobs and IAT_{SC} is the average service class SC's job interarrival time. The relative processing power assigned to a service class (sum of processing power of computational resources assigned to that SC) can be found from

$$\forall SC \cdot ppower_{asg_{SC}} = \sum_{\forall CR \in SC} \frac{speed_{CR}}{speed_{CR_{ref}}} \times ptime_{ref_{SC}}$$

In this equation $speed_{CR}$ is the processing speed of the selected computational resource (as stored in the information services) while $speed_{CR_{ref}}$ is the processing speed of a reference computational resource. Once CR query answers have been received, the GA (as shown in algorithm 4.4.1) will be started with cost function

$f(x)$ described in algorithm 4.4.2.

Algorithm 4.4.2: $f_{CRpart_{local}}(x)$

$result \leftarrow \frac{ppower_{asg_0}}{2}$
$maxAlloc_{over} \leftarrow 0$
$maxAlloc_{under} \leftarrow 0$
for $i \in SC_{local}$
do {
  $aux \leftarrow ppower_{req_i} - ppower_{asg_i}$
  if $aux < 0$
    then { **if** $-aux > maxAlloc_{over}$ **then** $maxAlloc_{over} \leftarrow -aux$; $aux \leftarrow ppower_{asg_i}$
    else { **if** $\frac{aux}{ppower_{req_i}} > maxAlloc_{under}$ **then** $maxAlloc_{under} \leftarrow \frac{aux}{ppower_{req_i}}$; $aux \leftarrow ppower_{asg_i} - aux$
  $result \overset{+}{\leftarrow} \frac{priority_i}{(\sum_{j \in SC_{local}} priority_j)} \times aux$
$result \overset{-}{\leftarrow} maxAlloc_{over} + maxAlloc_{under}$
return $(result)$

In this cost function (which is to be maximized), the objective is to donate to each *local* service class the same amount of processing power *relative* to their requested processing power (giving a higher cost function impact factor to service classes that have a high priority). Right at the start we penalize assigning all processing power to service class ID '0' (i.e. we only take into account half of the processing power when a resource can be used by *any* service class), as the objective of our algorithm is exactly to assign resources for exclusive use by a single service class, so service ID '0' assignations should only be used when conflicting service class requirements are found (e.g. two service classes need jobs to run on a single computational resource). The $maxAlloc_{over}$ and $maxAlloc_{under}$ parameters assure an even spread of processing power to services (both in case insufficient processing power is available as when sufficient processing power is available), as they keep track of the maximum amount of overallocated/underallocated processing power (compared to that service class' requested processing power) and penalize the cost function result accordingly.

If we take a look at our example (see figure 4.5) we notice that Grid site 1 has reserved one of the fast CRs for the computationally intensive service class (SC1), while Grid site 2 has reserved its only fast CR for processing SC1 jobs. The third Grid site only takes into account its local service class requirements, and assigns

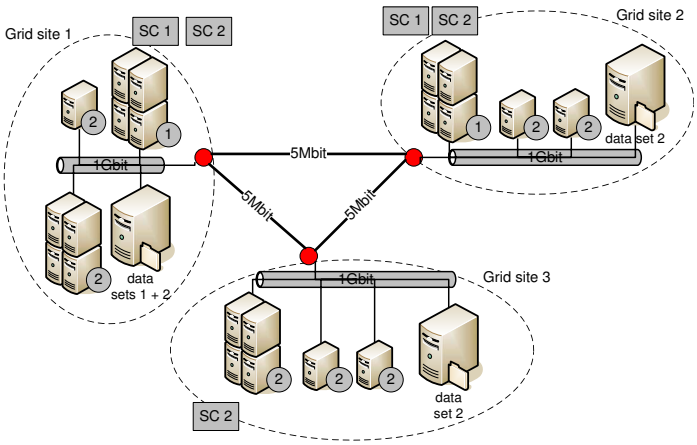


Figure 4.5: Grid example - local service CR partitioning

all of its resources to the data intensive service class. The scheduler can now only assign SC1 resources to SC1 jobs, preventing data intensive jobs from running on most fast computational resources (these data intensive jobs spend half of their *fast* computational reservation idling while I/O data is being sent/retrieved).

4.4.1.2 Global Service CR partitioning

The second partitioning strategy adds support for services offered at foreign Grid sites. The cost function impact factor of assigning resources to foreign service classes can be adjusted by the local service manager by tuning the foreign service policy $\rho_{SC_{foreign}}$. Support for foreign service classes can range from no impact at all on the cost function ($\rho_{SC_{foreign}} = 0$) to an impact equal to that of local service classes ($\rho_{SC_{foreign}} = 1$) or any value in between. The resulting cost function is stated in algorithm 5.3. In this algorithm we include both local and global services in the processing power allocation loop and, if we assign processing power to a foreign service class, multiply the assigned processing power with the foreign service policy $\rho_{SC_{foreign}}$, effectively manipulating the cost function impact factor of assigning processing power to foreign service classes according to the local policy.

Returning to our example (see figure 4.6), we notice that the third Grid site has taken into account the global need for computational power by service class 1 jobs and has assigned its fast computational resource to this service class.

Algorithm 4.4.3: $f_{CRpart_{global}}(x)$

$result \leftarrow \frac{ppower_{asg_0}}{2}$
$maxAlloc_{over} \leftarrow 0$
$maxAlloc_{under} \leftarrow 0$
for $i \in SC_{local} \cup SC_{foreign}$
do {
  $aux \leftarrow ppower_{req_i} - ppower_{asg_i}$
  if $aux < 0$
    then {
      if $-aux > maxAlloc_{over}$
        then $maxAlloc_{over} \leftarrow -aux$
      $aux \leftarrow ppower_{asg_i}$
    }
    else {
      if $\frac{aux}{ppower_{req_i}} > maxAlloc_{under}$
        then $maxAlloc_{under} \leftarrow \frac{aux}{ppower_{req_i}}$
      $aux \leftarrow ppower_{asg_i} - aux$
    }
  if $i \in SC_{foreign}$
    then $aux \leftarrow aux \times \rho_{SC_{foreign}}$
  $result \overset{+}{\leftarrow} \frac{priority_i}{(\sum_{j \in SC} priority_j)} \times aux$
}
$result \overset{-}{\leftarrow} maxAlloc_{over} + maxAlloc_{under}$
return $(result)$

4.4.1.3 Input Data Locality Penalization

Resource partitioning based solely on the processing needs of the different services can lead to bad performance. In case of data-intensive services in particular, one wants these services to be processed on computational resources located near input data that is generally requested by those service classes. In order to provide this functionality, the service manager queries the information services for both computational and data resources and constructs a list of which CRs have local access (i.e. accessible from the local Grid site) to which input sets. We adjust the cost function to include this notion and penalize assigning a computational resource that has *no local access to an input dataset much-needed by the assigned service*. The actual penalty depends on the input data intensiveness of the service class i ($\frac{InputReq_i}{IAT_i}$) when compared to the total input data requirements of all service classes ($\sum_{\forall j \in SC} \frac{InputReq_j}{IAT_j}$):

$$cost_{CR \in SCi} = \frac{\frac{InputReq_i}{IAT_i}}{\sum_{\forall j \in SC} \frac{InputReq_j}{IAT_j}} \times \frac{\rho_{cost}}{\#CR_{assigned_i}}$$

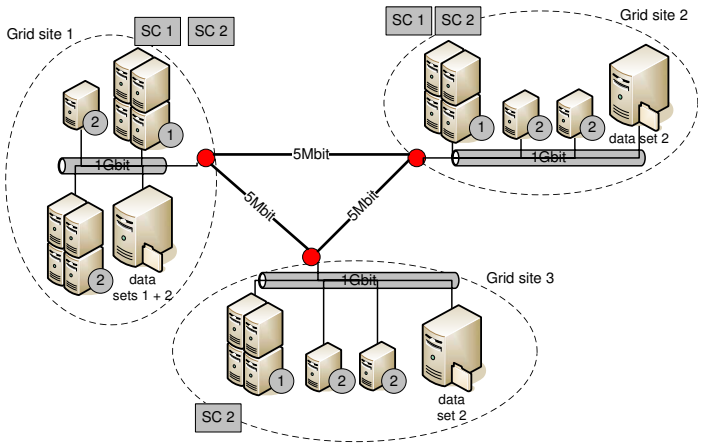


Figure 4.6: Grid example - global service CR partitioning

In this equation $InputReq$ is the average service class's input size requirement, $\#CR_{assigned}$ denotes the amount of computational resources assigned to the service class and ρ_{cost} describes the data non-locality penalty factor. An additional (yet larger) penalty is given when, amongst all computational resources assigned to a particular service, *not one of them* has access to a needed dataset, as it can be considered best practice that at least one computational resource can access a needed input set locally. This cost is only charged once for each service class.

$$cost = \frac{\frac{InputReq_i}{IAT_i}}{\sum_{\forall j \in SC} \frac{InputReq_j}{IAT_j}} \times \rho_{cost}$$

Both costs can be used as a penalty for the cost function in algorithm 4.4.2 and 4.4.3. Algorithm 4.4.4 shows the cost function for global service CR partitioning with input data locality penalisation.

Our example in figure 4.7 shows that all computational resources at the first Grid site have been assigned to service class 1. This is exactly because the first Grid site is the only site with local access to data set 1. Service class 2 is in turn assigned a fast computational resource at Grid site 3, which has local access to data set 2.

4.4.1.4 Network partitioning

Since the service monitor keeps track of I/O data characteristics of each service, data intensiveness relative to the other services can be calculated. This in turn can be used to perform per-service network bandwidth reservations. We have implemented a proof-of-concept network partitioning strategy, in which the service

Algorithm 4.4.4: $f_{CRpart_{globalIDLP}}(x)$

$result \leftarrow \frac{ppower_{asg_0}}{2}$
$maxAlloc_{over} \leftarrow 0$
$maxAlloc_{under} \leftarrow 0$
for $i \in SC_{local} \cup SC_{foreign}$
do {
  $aux \leftarrow ppower_{req_i} - ppower_{asg_i}$
  if $aux < 0$
    then {
      if $-aux > maxAlloc_{over}$
        then $maxAlloc_{over} \leftarrow -aux$
      $aux \leftarrow ppower_{asg_i}$
    }
    else {
      if $\frac{aux}{ppower_{req_i}} > maxAlloc_{under}$
        then $maxAlloc_{under} \leftarrow \frac{aux}{ppower_{req_i}}$
      $aux \leftarrow ppower_{asg_i} - aux$
    }
  if $i \in SC_{foreign}$
    then $aux \leftarrow aux \times \rho_{SC_{foreign}}$
  for $c \in CR_{assigned_i}$
    do {
      if $noLocalAccesToDataNeeded(c, i)$
        then $aux \stackrel{-}{\leftarrow} \frac{\frac{InputReq_i}{IAT_i}}{\sum_{\forall j \in SC} \frac{InputReq_j}{IAT_j}} \times \frac{\rho_{cost}}{\#CR_{assigned_i}}$
    }
  if $noCRsAssignedWithLocalAccessToDataNeeded(i)$
    then $aux \stackrel{-}{\leftarrow} \frac{\frac{InputReq_i}{IAT_i}}{\sum_{\forall j \in SC} \frac{InputReq_j}{IAT_j}} \times \rho_{cost}$
  $result \stackrel{+}{\leftarrow} \frac{priority_i}{(\sum_{j \in SC} priority_j)} \times aux$
$result \stackrel{-}{\leftarrow} maxAlloc_{over} + maxAlloc_{under}$
return $(result)$

manager calculates average data requirement percentages for each service class i:

$$bw_{req_i} = \frac{\frac{bw_{input_i} + bw_{output_i}}{IAT_i}}{\sum_{\forall j \in SC} \frac{bw_{input_j} + bw_{output_j}}{IAT_j}}$$

In this equation bw_{input} is the average service class's input bandwidth need i.e. $\frac{speed_{CR}}{speed_{CR_{ref}}} \times \frac{InputReq}{ptime_{ref}}$ while bw_{output} denotes the average service class's output bandwidth need: $\frac{speed_{CR}}{speed_{CR_{ref}}} \times \frac{OutputReq}{ptime_{ref}}$. The service manager sends the calculated information to the connection manager, who in turn will make service class bandwidth reservations on all network links for which it is responsible. Network partitioning can be applied to all previously mentioned partitioning algorithms.

If we employ network partitioning on our example (see figure 4.8) we notice

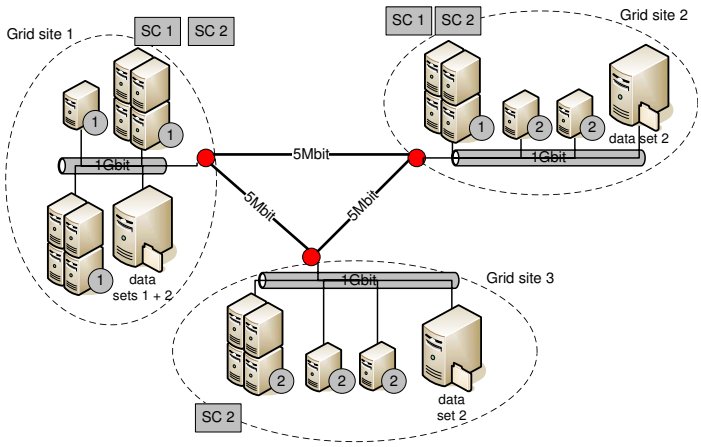


Figure 4.7: Grid example - global service CR partitioning IDLP

that all network links have been partitioned amongst service class 1 (which has 10 percent of the I/O needs of the data-intensive service class 2) and service class 2. This way, computationally intensive jobs assigned to the fast CR at Grid site 2 will not suffer from fully congested network links due to I/O from service class 2 jobs travelling over network links on their I/O retrieval/storage route, but instead will always have a minimum of 10 percent of the total network link capacity assigned to them. As our network partitioning algorithm only acts as a proof-of-concept heuristic, a variety of improvements still exist, most notably non-global percentual network link partitioning. More advanced network partitioning heuristics could take into account the computational resource-to-service assignations, Grid network topology and the location of different needed data sets to provide service classes with point-to-point advance connection reservations tailored to their needs.

4.5 Performance evaluation

4.5.1 Resource setup

A fixed Grid topology (see figure 4.9) was used for all simulations (run on an LCG-2.6.0 Grid [17] comprised of dual Opteron 242 1.6Ghz worknodes with 2 GB RAM per cpu, and operating under Scientific Linux 3). First, a WAN topology (containing 9 core routers with an average out-degree of 3) was instantiated using the *GridG* tool [18]. Amongst the edge LANs of this topology, we have chosen 12 of them to represent a Grid site. Each site has its own resources, management components and Grid portal interconnected through 1Gbps LAN links, with Grid site interconnections consisting of dedicated 10Mbps WAN links. A single ser-

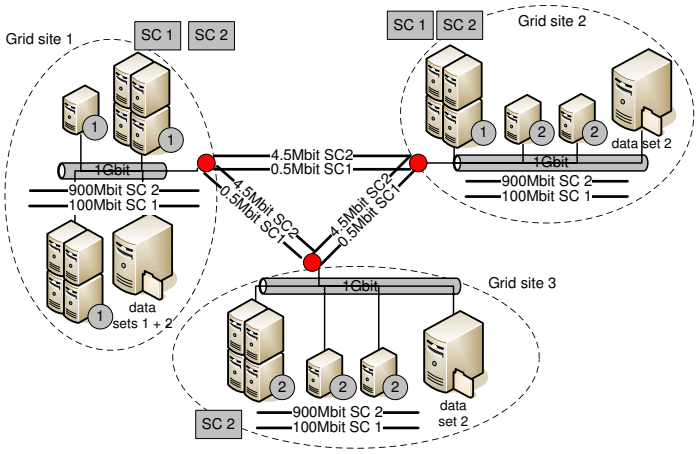


Figure 4.8: Grid example - Global service CR partitioning IDLP with network partitioning

vice manager was instantiated, and was given access to the different Grid sites' information services.

We have assigned 3 computational resources to each Grid site (for a total of 36 CRs). To reflect the use of different tiers in existing operational Grids, not all CRs are equivalent: the least powerful CR has two processors (which operate at the reference speed - note that speed in our computational model refers to the actual speed at which computational resources can process jobs i.e. computational resource performance). A second class of CRs has four processors, and each processor operates at twice the reference speed. The third - and last - CR type contains 6 processors, each of which operates at three times the reference speed. Conversely, the least powerful type of CR is three times as common as the most powerful CR, and twice as common as the middle one (for a total of 18 reference CRs, 12 four-processor CRs and 6 of the most powerful CRs deployed in our simulated topology).

We have assumed that storage resources offer "unlimited" disk space, but are limited in terms of access/write speed by the bandwidth of the link connecting the resource to the Grid site. Each site has at its disposal exactly one such SR. Each site's data resource contains 6 out of 12 possible data sets. These data sets are distributed in such a way that 50 percent of the jobs submitted to a site can have local access to their needed data set.

4.5.2 Job parameters

We have used two different, equal-priority service classes (each accounting for half of the total job load) in our simulations; one is more data-intensive (i.e. higher

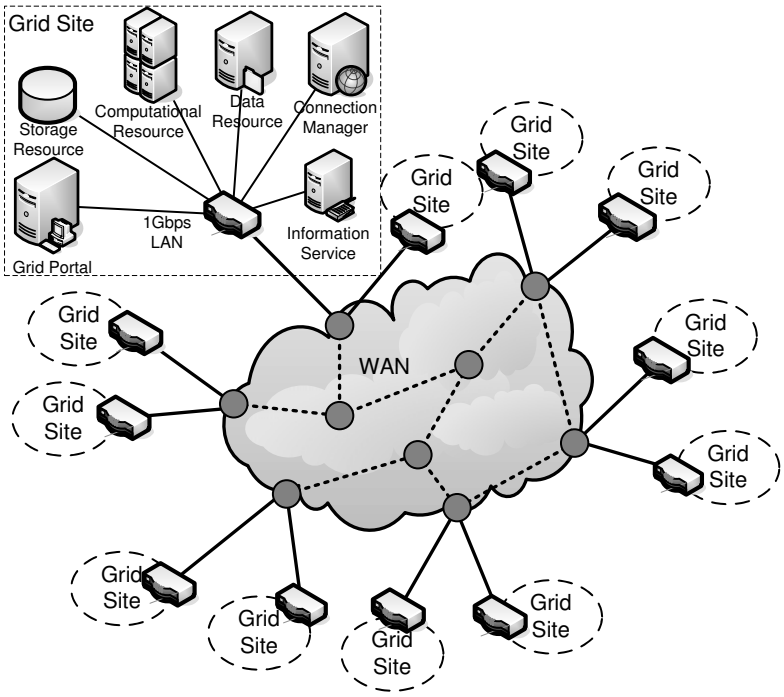


Figure 4.9: Simulated multi-site Grid topology

data sizes involved), while the other is more cpu-intensive. At *each* Grid site, two "clients" have been instantiated, one for each job type. Each client submits mutually independent jobs to its Grid portal. All jobs need a single data resource and a single storage resource. The ranges between which the relevant job parameters vary have been summarized in table 4.2. In each simulation, the job load consisted of 2784 jobs. For each scheduling algorithm, we chose to use a fixed interval of $20s$ between consecutive scheduling rounds. From the arrival rates in table 4.2 and

	CPU-Job	**Data-Job**
Input(GB)	0.01-0.02	1-2
Output(GB)	0.01-0.02	1-2
IAT(s)	30-40	30-40
Ref. run time(s)	100-200	40-60

Table 4.2: Relevant service class properties

the fact that multiple sites submit job simultaneously, we are likely to find multiple jobs in the queue at the start of each scheduling round.

4.5.3 GA-partitioning performance

The main drawback of GA-based partitioning is the time needed to complete a GA run (with reasonable results); on our sample scenario, a naive stop condition of 100 generations takes on average $2632s$ ($26.32s$ per generation but it should be noted that this time is not exclusive for GA solution calculation, but is also spent on all other simulation tasks during partitioning) as can be seen in figure 4.10. More reasonable GA calculation times (with an average of $1123.4s$ can however be obtained when using a more intelligent stop condition (i.e. stop when over a period of 15 generations the cost function optimum changes by less than 0.5 percent). For the GA approach, we used Grefenstette's settings [19], with a population of 30 per generation, $\rho_C = 0.9$ and $\rho_M = 0.01$. In case faster partitioning times need to be attained, one can either tune GA parameters (smaller population sizes, faster stopping condition, etc.) or deploy a service monitor/service manager at every Grid site, who are then responsible for communicating with the foreign site's service monitor components and partitioning the resources at their assigned site (as described in section 4.3.2).

Figure 4.11 shows the trend of the cost function optimum for different GA generations (partitioning occurred on the topology discussed in section 4.5.1). The cost function used is the one discussed in section 4.4.1.1 (Local Service CR partitioning with Input Data Locality penalization). It is important to note that during the calculation of a resource-to-service partitioning, Grid operation does not stall

but continues as normal, as the service management components do not block any other management components.

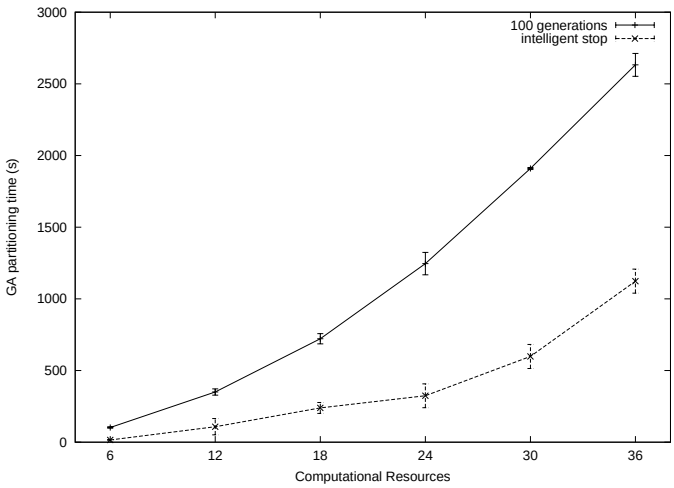


Figure 4.10: GA solution calculation time

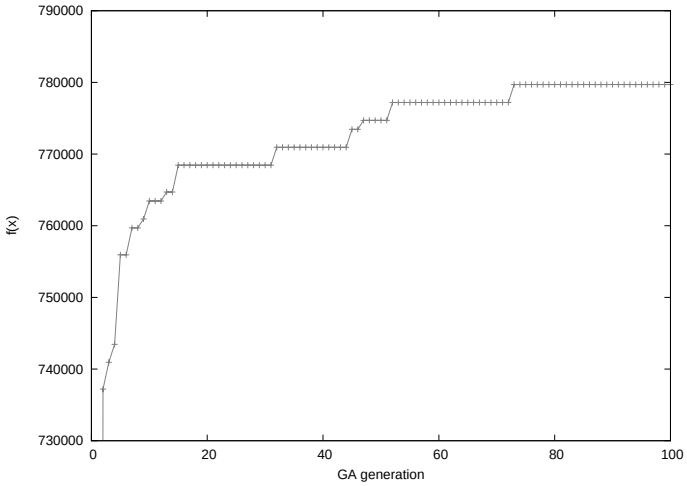


Figure 4.11: GA optimal fitness trend

4.5.4 Job response time

We define the *response time* of a job as the difference between its end time (time at which the job's final output block has been sent to the scheduler-assigned stor-

age resource) and the time it is submitted to the scheduler. If we compare the performance of the different GA-based partitioning heuristics (see figure 4.12 - in this figure IDLP is short for Input Data Locality Penalization), the results show that average job response times can be improved significantly (by 29 percent when network unaware scheduling is used and by 30.5 percent when network aware scheduling is employed) by employing a resource partitioning algorithm prior to scheduling. This behaviour can be explained because resources are reserved for exclusive use by a service class. It is this service-exclusivity that forces the scheduler to not assign jobs to less-optimal resources (e.g. non-local access to needed input data, low processing power available), but to keep the job in the scheduling queue until a service-assigned resource becomes available. Note that when network unaware scheduling is employed, no connection partitioning results are shown, due to the fact that the network unaware scheduling algorithm does not take into account the connection reservation system.

When scheduling network unaware, the best results are attained when using computational partitioning taking into account input data locality, as data intensive jobs can be run on computational resources reserved physically near resources that store much needed I/O data, leading in turn to less computational stalling, as I/O data suffers from less network bottlenecking. When network aware scheduling is employed, one is best off using a heuristic that partitions both computational and network resources. Network partitioning assures that service classes with high I/O requirements do not consume all bandwidth (thereby preventing computationally intensive service classes from retrieving their I/O), but instead force them to only use a predefined percentage of bandwidth. It is interesting to note that, since the average runtime of the computational and data intensive service class jobs is 150 and 50 seconds respectively (only taking into account computational requirements) and taking into account that the fastest computational resource in our simulated topology can process a job at three times the reference speed, optimal average job response time is 33.33 seconds.

4.5.5 Resource efficiency

Using the same job load, the average hopcount over which data was transferred by data-intensive jobs (with hopcount equalling the amount of hops between data resource and computational resource added to the amount of hops between computational resource and storage resource) is shown in figure 4.13. We notice that average hopcount dropped by 4.8 percent when network unaware scheduling was employed (i.e. computational resource partitioning with data locality versus non-service partitioned resources), and by 5.5 percent when a network aware scheduling heuristic was used (i.e. network partitioning with data locality compared to the non-service managed case), due to the fact that input/output data was located at re-

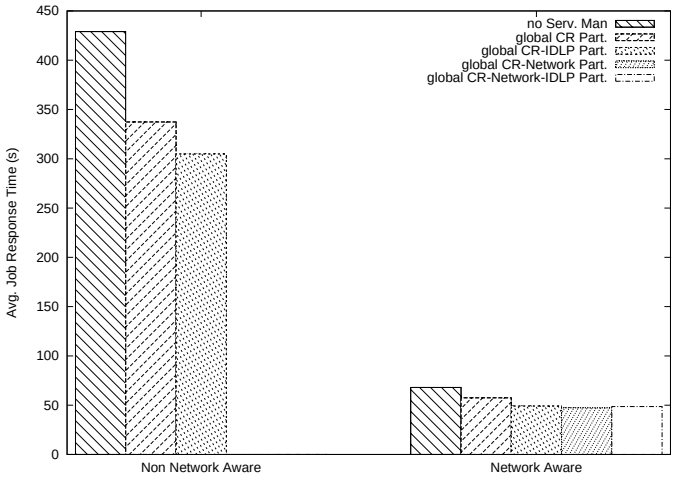


Figure 4.12: Job response times for GA-based partitioning heuristics

sources closer to the job's service class' assigned CRs. Network resources are thus used most sparingly when computational and network resource partitioning with input data locality is employed together with a scheduling algorithm that takes into account the state of the network links interconnecting the job's resources.

Furthermore, we calculated the average computational resource utilization:

$$\frac{\sum_{j \in Jobs_{CR}} Load_j}{Makespan \times speed_{CR}}$$

In this equation $\sum_{j \in Jobs_{CR}} Load_j$ is the total amount of processing that needs to be done for the jobs accepted on this resource, *Makespan* denotes the total amount of time these jobs spend on this resource, while $speed_{CR}$ is the amount of work that can be processed per time-unit on that particular computational resource (note that $\frac{\sum_{j \in Jobs_{CR}} Load_j}{speed_{CR}}$ represents the minimum amount of time for processing these jobs on that particular computational resource, if no time is lost waiting for I/O data to arrive). The improvement obtained by employing resource-to-service partitioning when using network unaware scheduling equals 17 percent, whereas in the case where network aware scheduling is used, it is 14.6 percent. Indeed, the fastest (and rarest in our topology) computational resources were automatically reserved for processing computationally complex jobs, disallowing data intensive jobs from cluttering these resources and using their full processing potential for those computationally intense jobs. The slower computational resources were then assigned to the data intensive service classes, that (because of their large I/O needs) benefit more from having fast (i.e. LAN) access to much needed data.

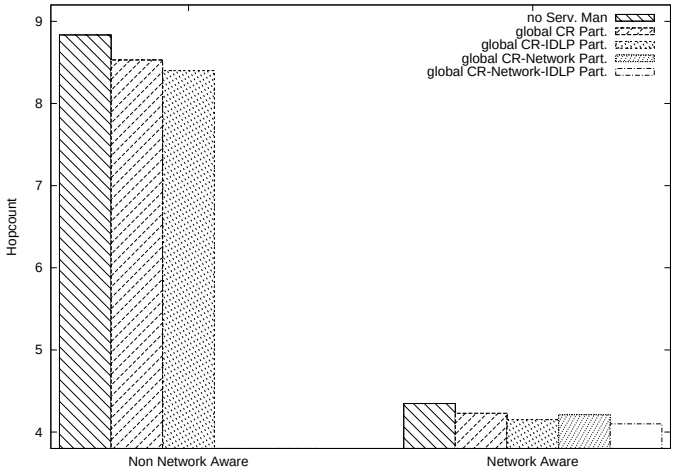


Figure 4.13: Network resource efficiency

4.5.6 Scheduling

We measured the time it takes to calculate a scheduling decision and noticed a decrease in scheduling time of 28.17 percent when comparing the service managed Grid to the non-service managed Grid in case network aware scheduling is used (i.e. from an average $7.88s$ in the non service managed case to 5.66s in the service managed Grid). Note that this is the time measured to schedule *all* jobs that are present in the scheduling queue, as we are scheduling in batch with a scheduling round time interval of $20s$. This behaviour can be explained by the fact that a scheduler queries the information services for resources adhering to a job's requirements *and* assigned to either the job's service class or service class 0. When resources are partitioned amongst services, less results will be returned to the scheduler, allowing for faster schedule making decisions.

4.5.7 Priority - service class QoS support

In another experiment, we gave the cpu-intensive jobs higher priority than the data-intensive jobs (by informing the service manager of the higher service class priority before resource-to-service partitioning started) and let the service manager construct a Virtual Private Grid (dedicated resource pool, scheduler and information service) for each service class. Due to the high priority of the cpu-intensive class, its cost function impact factor becomes higher which leads to more (and/or better) resources being assigned to the prioritized class. Also, during deployment of the VPG schedulers, the service manager configures the dedicated cpu-intensive scheduler to schedule those prioritized jobs as soon as possible, using a network

aware scheduling algorithm (the data intensive jobs were also scheduled using a network aware scheduling algorithm, but were by default queued until the next scheduling round). The results are shown in figure 4.14: the average job response time of the computationally intensive service class is substantially improved (due to more/better resources assigned to this service class and the ASAP scheduling policy enforced by the VPG scheduler), while the data intensive service class's average response time gets worse (prioritizing service classes over other service classes can not lead to win-win situations: the non-prioritized service classes' performance will deteriorate).

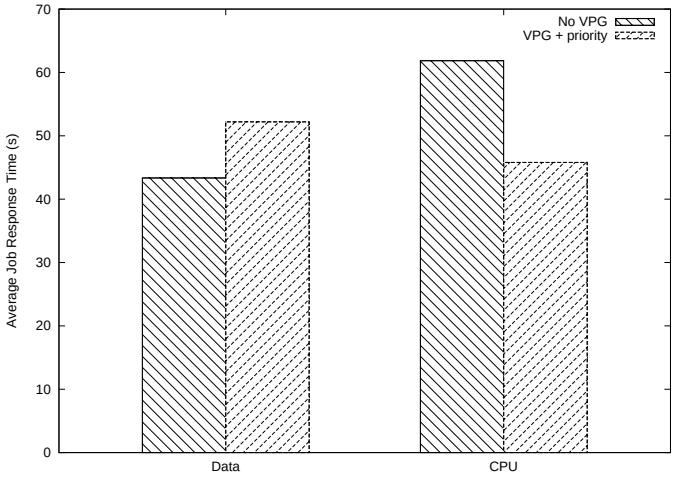


Figure 4.14: VPG service class priority support

4.6 Conclusions

We proposed the use of a distributed service management architecture, following the OGSA 'service level manager' concept, capable of monitoring service characteristics at run-time and partitioning Grid resources amongst different priority service classes. This partitioning, together with the dynamic creation of per-service management components, lead to the introduction of the Virtual Private Grid concept. A variety of resource-to-service partitioning algorithms (based on Genetic Algorithm heuristics) were discussed and we evaluated their performance on a sample topology using NSGrid. Our results show that the proposed service management architecture improves both network and computational resource efficiency and job turnaround times, eases the process of making scheduling decisions, and at the same time offers service class QoS support. Management complexity

and scheduling / information service scalability is improved due to the automated deployment of service class dedicated management components.

References

[1] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke. *Grid services for distributed system integration*. IEEE Computer, 35(6):37–46, 2002.

[2] *Enabling Grids for E-Science in Europe*. `http://egee-intranet.web.cern.ch`.

[3] I. Foster and al. *The Open Grid Services Architecture, Version 1.0*. draft-ggf-OGSA-spec-019 `http://forge.gridforum.org/projects/ogsa-wg`.

[4] K. Czajkowski and al. *The WS-Resource Framework Version 1.0*. `http://www.globus.org/wsrf/specs/ws-wsrf.pdf`.

[5] J.O. Kephart and D.M. Chess. *The Vision of Autonomic Computing*. IEEE Computer, 36(1):41–50, 2003.

[6] A.G. Ganek and T.A. Corbi. *The dawning of the autonomic computing era*. IBM Systems Journal, 42:5–18, 2003.

[7] I. Foster K. Ranganathan. *Simulation Studies of Computation and Data Scheduling Algorithms for Data Grids*. Journal of Grid Computing, 1:53–62, 2003.

[8] F. Berman et al. *Adaptive Computing on the Grid Using AppLeS*. IEEE Transactions on Parallel and Distributed Systems, 14:369–382, 2003.

[9] H. Casanova H. Dail, F. Berman. *A Decoupled Scheduling Aproach for Grid Application Development Environments*. Journal of Parallel and Distributed Computing, 63-5:505–524, 2003.

[10] R. Wolski, N. Spring, and Jim Hayes. *The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing*. Journal of Future Generation Computing Systems, 15(5-6), 1999.

[11] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. *Grid Information Services for Distributed Resource Sharing*. In Proc. of the 10th IEEE International Symposium on High-Performance Distributed Computing, 2001.

[12] V. Sander I. Foster, A. Roy. *A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation*. In Proceedings of the Eighth International Workshop on Quality of Service (IWQoS 2000), 2000.

[13] A. Rodger. *Analyst report: Butler Group Subscription Services: Technology Infrastructure - IBM Tivoli Intelligent Orchestrator and IBM Tivoli Provisioning Manager*. `ftp://ftp.software.ibm.com/software/tivoli/analystreports/ar-orch-prov-butler.pdf`, 2004.

[14] H.L. Lee and al. *A Resource Manager for Optimal Resource Selection and Fault Tolerance Service In Grids*. In Proceedings of Cluster Computing and the Grid (CCGrid 2004), 2004.

[15] D. Yu and T.G. Robertazzi. *Divisible Load Scheduling for Grid Computing*. In Proceedings of the IASTED 2003 International Conference on Parallel and Distributed Computing and Systems (PDCS), 2003.

[16] P. Thysebaert, B. Volckaert, M. De Leenheer, F. De Turck, B. Dhoedt, and P. Demeester. *Resource partitioning algorithms in a programmable service Grid architecture*. published in Lecture Notes in Computer Science, Proceedings of the 5th Intern. Conf. on Computational Science ICCS 2005, 3516:250–258, 2005.

[17] *LHC Computing Grid project*. `http://lcg.web.cern.ch/LCG`.

[18] D. Lu and P. Dinda. *Synthesizing Realistic Computational Grids*. In Proceedings of ACM/IEEE Supercomputing 2003 (SC 2003), 2003.

[19] J.J. Grefenstette. *Optimization of control parameters for genetic algorithms*. IEEE Trans. Systems, Man, and Cybernetics, 16-1:122–128, 1986.

5
Media Grids

5.1 Introduction

In this chapter, we introduce a typical Grid use case based on the recent trend of employing Grid technology in audio/visual production and distribution companies. In this media environment, typical audio/video tasks impose heavy network resource QoS requirements. We discuss how NSGrid, the network aware Grid simulator presented in chapter 3, was extended to incorporate typical audio/visual production company profiles, user profiles and task profiles in order to allow quick and accurate simulations for studying the effectiveness of introducing Grid technology in the media production/distribution environment.

Much in the same way as other businesses, the media industry has been confronted with an increasing complexity in both the technical and the business domain. Up until now, a broadcaster was an umbrella organization for different kinds of in-house activities like media production, distribution and play-out, etc.. More and more however, business drivers such as cost reduction (a.o. reduced infrastructure investments by sharing resources), added value management, partnerships, global sourcing, and business componentisation are forcing these companies to become more agile, find partnerships and evolve to dynamically extending organizations, with business models based on business services available within the media market. These parameters combined with possible future mergers, acquisitions and fusions drive the media businesses to become more agile.

Furthermore, exponential decrease of harddisk costs [1] ignited a paradigm

shift in the production of audiovisual media from tape to file based. Current cost per byte of harddisk based storage systems rivals that of tape based systems and is expected to go below the stagnating prices of the latter. Although today's architectures promise *democratization* of data access, i.e. inexpensive, non-mediated, and shared access to centrally-managed storage, this promise is only partially met by existing installations. On a software level, generic (Grid-enabled) applications are tuned towards typical ICT related requirements and are not yet fitted for the specific challenges induced by a file based media production and archiving platform.

In the long term, one wants to allow automated interaction between several audio/video media production sites, and share centralized storage, computational and specialized (e.g. capturing devices, broadcasting equipment) resources with several independent corporate users in a controlled manner. It is in this domain that media production environments can benefit from Grid technology to both improve media handling/processing times and provide a means for securely sharing and utilizing distributed resources and applications amongst multiple virtual organizations by employing specialized Grid middleware.

Due to the specific scenario however, current Grid technology can not be introduced in a straightforward way. The high bandwidth, reliability and short response time requirements when handling audio/video streams (as will be illustrated in section 5.4) imply the need for special care in the design of the overall architecture and in particular in the scheduling and resource control process. Media handling can take place at local sites before streaming them to a remote site or can be performed at a remote site: the scheduling, resource control and QoS management components of the Grid will have a high impact on the achieved application performance. Furthermore, the software architecture of the management platform will need to exhibit high performance and reliability to meet the specific application requirements. The MediaGrid architecture presented in this chapter has been developed to cope with these challenges, and will make it possible for media partners to evolve to extended organizations where partnerships, media communities and commercialization of media services are omnipresent.

Advantages of Grid-enabling the audiovisual media production/distribution companies would be:

- Ability to distribute media files among different companies within an environment with high reactivity requirements and various levels of Quality of Service (QoS)
- Ease the exchange of media resources/assets (rendering farms, specialized media capture devices, etc.) allowing for a.o. distributed computing
- Integration of broadcast media exchange standards (e.g. the EBU's P/Meta standard [2]) in a Grid services environment to provide interoperability between different media content providers

- Migration from special purpose resources and applications to conventional IT hard/software

- Stimulate the growth of media community Virtual Organization (VO) setups supporting advanced collaborative working

This chapter continues as follows: first we give an overview of the related work in section 5.2, and continue by discussing the MediaGrid (Micro/MacroGrid) architecture in section 5.3. An overview of the different media production/distribution company profiles, along with the typical characteristics of their associated job classes is presented in section 5.4. MediaNSG, a MediaGrid simulator is discussed in section 5.5, while simulation results are shown in section 5.6. Finally, we give some concluding remarks in section 5.7.

5.2 Related work

For an overview of current Grid enabling technologies we refer to the related work of the previous chapters. Here we focus on Grid technology specifically tuned to the needs of audio/visual production and distribution companies.

GridCast [3, 4] is a research project being undertaken by the BBC and the Belfast eScience Center aiming to develop a prototype media Grid, running on Globus middleware, that will manage the sharing of program content between distributed sites. The objectives are to effectively manage the distribution of broadcast media files, permit distributed processing and provide security and network resilience within a highly reactive environment requiring high levels of Quality of Service (QoS). The GridCast project has similar objectives as our work, but focusses more on the specific BBC topology (with regional BBC departments interacting with the main BBC production house), whereas we try to provide a general framework and focus on the accurate simulation of a multitude of collaboration setups between audiovisual companies.

The FIPA project [5] (File based Integrated Production Architecture), is an IBBT project aiming at the development of an IP based architecture to share storage and computing power on single or multiple sites. Application areas are digital media production, e-security, e-health, etc. Apart from the storage, processing and management of the data, accessibility is also a crucial architectural issue, especially since more and more companies tend to share their data with business partners and freelance international employees. The work contained in this chapter is based on and an extension to some of the research work performed in the FIPA project.

A scalable solution for digital media post production networks is offered by Force10 Networks [6]. They mainly focus on the interconnectivity of rendering

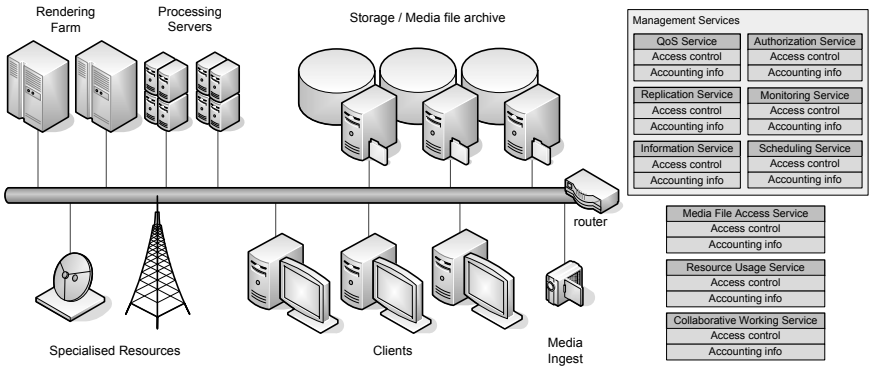


Figure 5.1: Typical MicroGrid scenario

farms with several hundreds of cluster nodes through Ethernet LANs and offer no full Grid solutions.

SGI [7] is known to be able to deploy an IT storage, computing and networking hardware infrastructure tuned to broadcast media environments.

MediaNSG, the MediaGrid simulator detailed in this chapter, was developed on top of NSGrid (detailed in chapter 3), our in-house developed network aware Grid simulator. An overview of other Grid simulators has been given in chapter 3.

5.3 MediaGrid architecture

The MediaGrid architecture consists of two main components: the MicroGrid concept on one side and the MacroGrid concept on the other. The MicroGrid is very similar to NSGrid's *Grid site* concept from chapter 3 while a MacroGrid shares similarities with NSGrid's *Grid* concept. We introduce these new terms in order to have a clear distinction between NSGrid's generic simulation structure and the concrete use case at hand (i.e. Grid computing technology for audio/visual production companies). In what follows we will explain the MediaGrid constituents in detail.

5.3.1 MicroGrid

As explained above, a MicroGrid (see figure 5.1) denotes a Grid set up at a local audiovisual media processing facility, interconnecting the different local resources and providing the tools to manage, access and control local (self-owned) resources. Resources can be storage/data resources, providing diskspace for storing and retrieving media files, or computational resources, which in turn provide the computational power required for processing the different user submitted tasks. In

a media production company, one typically discerns computational resources located in terminals (with a high degree of interaction between the job and user e.g. editing terminals) and computational resource farms (focussed on fast processing of computationally intensive tasks e.g. rendering). The MicroGrid can also allow Grid access to specialized resources (capturing devices, broadcasting equipment, etc.).

Each MicroGrid, like our local Grid site described in chapter 3, needs a set of Grid management components to be able to tackle issues such as job scheduling, Quality of Service, etc.. Required management components are: a scheduling system (responsible for the allocation of resources to jobs according to a certain objective e.g. minimize job turnaround time or meet specific deadlines), information service (storing registered MicroGrid resource's properties and characteristics), monitoring system (monitoring the status of computational, storage and data resources), connection manager (responsible for monitoring the status of network resources and setting up network connection reservations), service monitor (monitoring QoS requirements of jobs and collecting service class information) and a service manager (which reserves resources for service classes in order to provide them with specific QoS guarantees). Other notable management components include an accounting component, an authorisation/security component (for restricting MicroGrid resource access) and a data transfer/replica manager (responsible for replicating/caching frequently accessed media files).

A MicroGrid can thus be seen as a provider of a set of Grid services, and these services can be advertised not only to the local company users, but if wanted also to 3rd party media companies with which one wants to collaborate (see section 5.3.2). Each offered service is accompanied by extensive access control (describing which user/userclass can use that service and to what extent) and accounting agreements (e.g. service usage pricing information for different user classes), allowing MicroGrid managers full control over how local resources may be utilized by *MediaGrid* users.

5.3.2 MacroGrid

A MacroGrid is a collection of interconnected MicroGrid sites. In such a MacroGrid, resources can be shared amongst the different constituent MicroGrids (while taking into account the access policies of each MicroGrid's resource usage service). This way, jobs that originate at one MicroGrid site, can be migrated to another MicroGrid for processing (e.g. in case insufficient processing power is available at the originating site or if the job needs to have access to specialized resources not available at its originating site). The MicroGrid schedulers query the different MicroGrid sites' information services for resources adhering to a user job's requirements and decide whether it is beneficial (e.g. faster processing times)

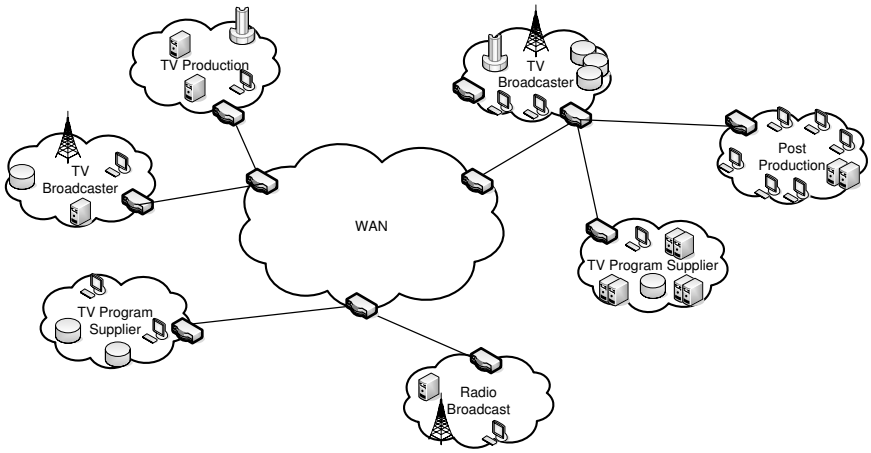


Figure 5.2: Physical MacroGrid

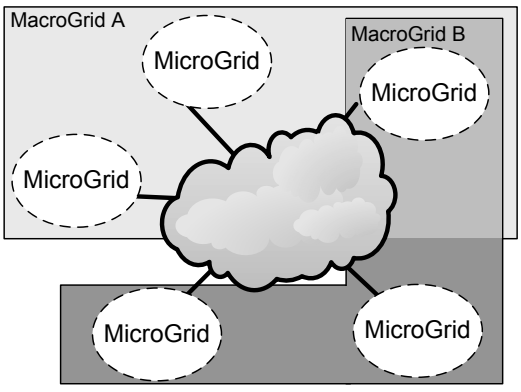


Figure 5.3: Logical MacroGrid

or necessary (e.g. specialized resources unavailable locally or when having to cope with local resource failures) to utilize remote resources, while taking into account the possible downsides of using remote resources (price tag, larger network transfer times due to remote location, etc.).

On top of resource sharing, MacroGrids also enable media file sharing between different MicroGrids. In this case, users are able to utilise/retrieve/store remote media files, again taking into account any access restrictions that have been specified (speed/ratio limitations in order to not deteriorate internal MicroGrid performance, clearance levels depending on the content/copyrights of media files, etc.).

Another important benefit of sharing MicroGrid services across company boundaries is the ability for users to work on projects collaboratively. The accounting managers of each MicroGrid can be used to keep track of resource and media file service usage by users/MicroGrids allowing economic gains by charging money and/or bartering for external resource usage compensation.

It is important to note that a MacroGrid does not have exclusive access to a MicroGrid's services: one MicroGrid can be included in multiple MacroGrids (and each membership can come with different resource/media file usage policies and access control configurations), with each MacroGrid representing a different Virtual Organization (see figure 5.2 and figure 5.3). Also, MacroGrids are not necessarily static structures, in that MicroGrids can join or leave this Virtual Organization at any time by changing service access policies.

5.4 Audiovisual application/user/company profiles

Together with partners from the media industry (more specifically the Flemish Radio- and Television Network [8] and Video Promotion [9], a company active on the broadcast television market), we studied the characteristics and requirements for the audiovisual applications that are to be supported by the MediaGrid architecture. This resulted in task, user and company profiles that have been implemented in the MediaNSG simulator (see section 5.5) and that can readily be used in simulations.

5.4.1 Application profiles

Audiovisual application classes show large differences in their processing, network and storage requirements. In table 5.1 we give an overview of average task class/application requirements of the most typical tasks/applications in a media centered company. The Quality of Service parameter can be used by MediaNSG while scheduling and during service management to ensure priorities are given to high QoS tasks. Table 5.2 shows the network and storage requirements for different resolution audio and video streams.

Following important tasks were identified:

- Ingest: deals with bringing media files onto the storage/archive system, extracting keyframes and constructing metadata about the ingested media.
- Quality checking, HiRes Browse: tasks from this class inspect the quality of media files in high resolution to see if it's fit for playout.
- LoRes browse: mainly used to rapidly shuffle through different archived media files in low resolution when trying to find specific or suitable source material.
- LoRes rough EDL: construction of a rough Edit Decision List (EDL). This Edit Decision List is a list of events that include the sources to be recorded from and information about transitions (cuts, dissolves, wipes), transition durations, etc.. Once an EDL has been processed, the result will be a newly constructed media file.
- Send to/Restore from archive: fetching data from the archive or storing new media files mainly stresses the available network resources.
- Craft editing: high quality finegrained editing and jog shuffling of multiple audio/video streams.
- Rendering, conforming, transcoding: this task involves rendering graphics, conforming of media to different video standards and transcoding of audio/video data to different qualities/resolutions/standards.
- Playout: Viewing multiple audio/video streams and sending one of those to playout equipment (e.g. broadcast equipment).
- Audio editing: Editing of multiple audio streams (possibly in conjunction with a video stream that needs to have the associated audio stream edited)
- Graphic creation: The creation of Computer Generated Imagery (CGI), custom scene transitions, etc.

5.4.2 User profiles

Now that we have discussed the different application/task classes and their requirements, we can look at the different user classes of typical audiovisual companies, with each user class showing widely differing characteristics regarding which applications they use:

- Ingester: This profile includes tasks like quality checking and low resolution browsing, besides the actual ingesting of media onto the storage archive.

	Bandwidth	CPU	Storage	QoS	no
Ingest	Lo- or HiRes A/V	Low	0,65-25,7GB/h	High	1
Quality checking, HiRes browse	HiRes A/V	Low	25,7GB/h	Low	2
LoRes browse	LoRes A/V	Medium	0,65GB/h	Low	3
LoRes rough EDL	LoRes V, Lo- or HiRes A	High	0,5GB/h; 0,15-0,7GB/h	Medium	4
Send/Restore archive	Lo- or HiRes A/V	Low	0,65-25,7GB/h	Medium	5
Craft editing	5-10 HiRes A/V	High	5-10 25,7GB/h	High	6
Rendering, conforming, transcoding	HiRes A/V	High	25,7GB/h	Low	7
Playout	1-40 HiRes A/V	Low	1-40 25,7GB/h	High	8
Audio editing	Lo- or HiRes A/V	High	0,65-25,7GB/h	Medium	9
Graphic creation	HiRes V	High	25GB/h	Low	10

Table 5.1: Typical audiovisual application requirements

Streams	Bitrates	Storage
HiRes Video	20-50 Mb/s	25 GB/h
LoRes Video	1 Mb/s	0.5 GB/h
HiRes Audio	1.5 Mb/s	0.7 GB/h
LoRes Audio	256 kb/s	0.15 GB/h
HD HiRes Video	200Mb/s	100GB/h

Table 5.2: Network and storage requirements of typical audio/video streams

- Video journalist: The main tasks of a journalist are low resolution browsing, low resolution rough EDL construction (Edit Decision List) and rendering, conforming and transcoding.
- Audio/Video editor: an audio editor deals with mixing and editing multiple audio tracks, while video editing includes quality checking, craft editing, rendering/conforming/transcoding and graphic creation.
- Producer/Director: involved at different stages of media production, mainly doing low resolution browse tasks, with the occasional sending to/restoring from archive and some quality checking and/or high resolution browsing.
- Playout: tasks include quality checking, low resolution browsing and playout.
- Archivist: an archivist mainly performs low resolution browsing and sending to/restoring from archive.

This information, together with the user/task workflows (presented in figure 5.4) and average application characteristics presented in table 5.1, has been used to construct accurate media user profiles for use in MediaNSG simulations.

5.4.3 Company profiles

Finally, profiles have been provided for typical audiovisual companies (mainly describing the average amount of users from each userclass working simultaneously). The most important profiles are:

- Television production: an example of television production is news program production. In these organizations tens (regional) or hundreds (national) of video journalists gather information that has to be ingested, edited, archived and played out.
- Television post production: in a post-production facility the same user classes are present, along with producers / directors managing the studio work.

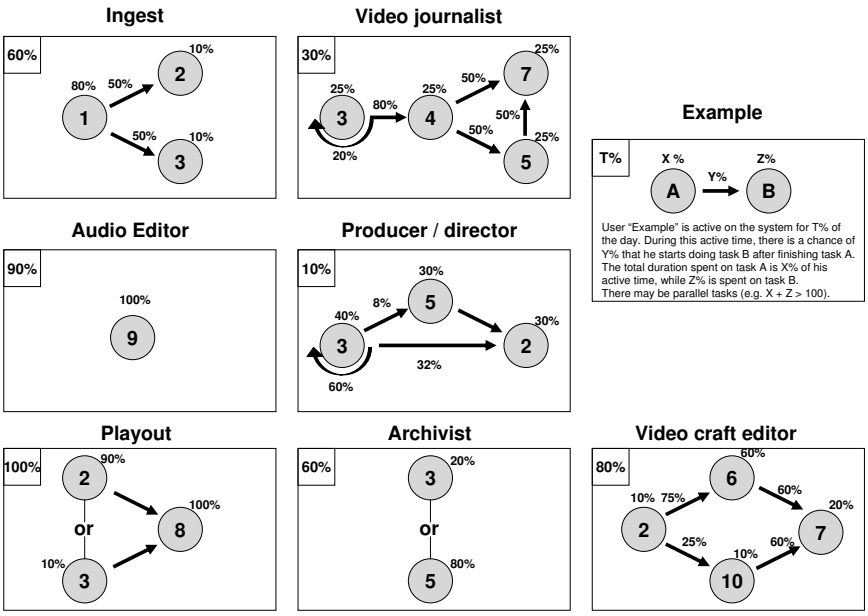


Figure 5.4: Task workflow of typical audiovisual company user classes

- Television broadcast: television broadcast companies are not involved in (post) production. The focus is more on playout than on editing.
- Television program supplier: these companies combine individual items into finished programs and send these to television broadcasters. Editors and producers/directors are the most important user classes in this type of organization.
- Video on Demand: companies delivering Video on Demand services mainly focus on indexing of the available material, user and channel dependent encoding of the streams and play out.
- Radio broadcast: similar to television broadcast, but with different requirements (e.g. no buffering or delays allowed).

5.5 MediaGrid simulation

If we wish to develop MediaGrid suitable scheduling/service management algorithms, or wish to evaluate the performance of different network/computational/storage resource configurations, we either have to construct a testbed and measure

	ingest	video journ.	audio ed.	video ed.	prod./director	playout	archivist
regional TV prod.	2	30-50	2-3			2	2
national TV prod.	3	300-500	20-30			3	4
TV post prod.	1	10-50	1-3	5-20	10	1	1
TV broadcast	1	5-10	1-5			1	1
TV program supplier			25		25		
Video on demand	2		5			2	1
Radio prod./ broadcast			30		20	50	

Table 5.3: Audiovisual company typical user class representation

task/resource performance, or we can simulate the MediaGrid's behaviour. Due to the size and the amount of resources involved in setting up a MediaGrid testbed each time a new scenario needs to be evaluated, accurate simulation of MediaGrid scenarios is likely to be more efficient.

MediaNSG, a MediaGrid specific extension to NSGrid has been developed allowing users to simulate typical task submission behaviour of different media company organizations and experiment with scheduling and service management architectures. MediaNSG supports the simulation of both Micro- (single site) and MacroGrid behaviour (Grid comprised of different interconnected Micro Grids), and provides the user with output data regarding job execution statistics (job response time, time spent in scheduling queue, data transfer size/speed, etc.) for the different tasks, resource (computational, storage and network resources) and management component (scheduler, information service, etc.) usage statistics, bottlenecks, etc..

5.5.1 User model

Each user belongs to a particular user class (with the latter describing the characteristics of job types that are to be submitted) and is modelled as a job submitting entity. Every time a user wishes to launch a job, it constructs a job from one of its user class's registered job types (job characteristics are generated from distributions specified in the job type's definition) and waits until the job has been scheduled and processed by the Media Grid. When the job is finished, the user class's workflow (as seen in figure 5.4) is inspected to see from which job type a new job is to be generated (i.e. a user class's workflow determines (by probability) which job types will be executed after a particular job has finished).

5.5.2 MediaNSG operation

In order to set up a simulation, users must provide both a company and resource topology description. The default organizations that are discussed in section 5.4.3, are all readily available through the MediaNSG frontend. Each organization is modelled as a collection of users belonging to different user types (see section 5.4.2), with each user in turn being modelled as a task submitting entity (i.e. users submit tasks according to the task workflows discussed in section 5.4). The default tasks described in section 5.4.1, along with all their properties (CPU utilization, storage needs, bandwidth, QoS, etc.) have been supplied, and all task characteristics can be modified through the GUI (see figure 5.5). New media organization profiles can be added, and existing profiles can be modified to include additional users and/or jobs.

Currently, simulated MicroGrid topologies deploy one central data storage / retrieval resource by default, with each client submitting jobs from a dedicated

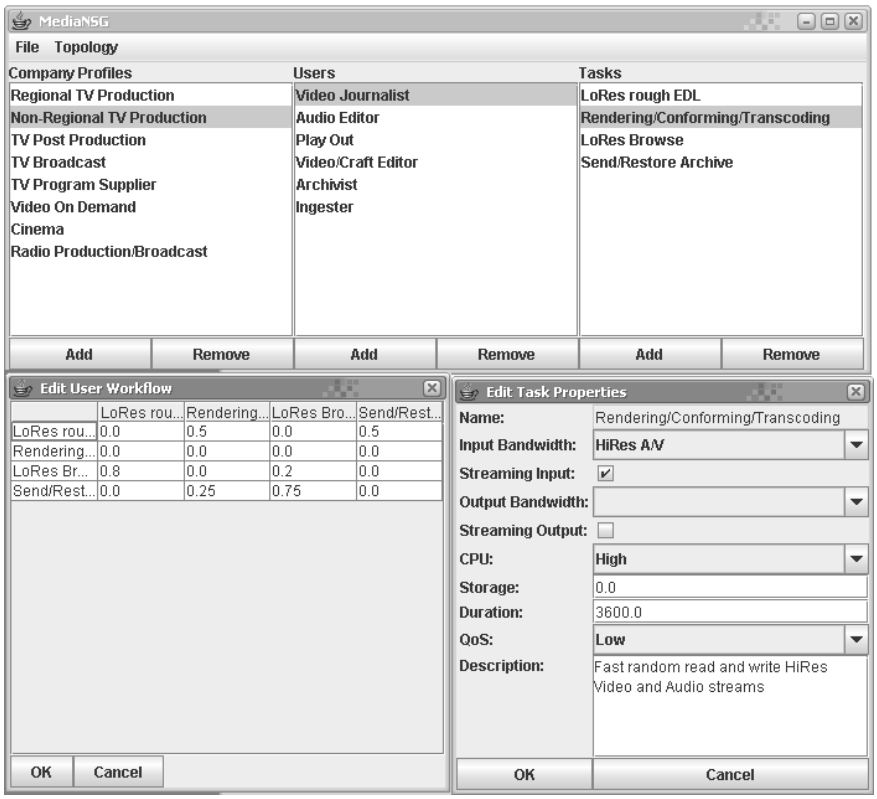
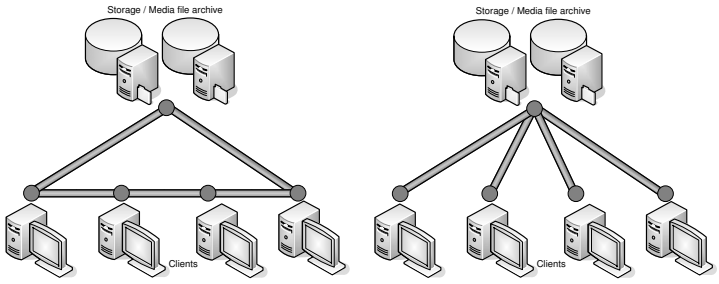


Figure 5.5: MediaNSG frontend

computational resource (which is used to provide processing power for the user-submitted jobs) connected to this storage resource by means of TCP/IP network links. Furthermore, each MicroGrid site can have a computational resource farm, offering processing power to computationally intensive tasks (e.g. rendering tasks). Different MicroGrid network topologies can automatically be generated by MediaNSG: for now point-to-point topologies, in which clients are directly connected to the data storage resources, and ring topologies are supported. Other network topologies can easily be added manually or one can use GridG [10, 11], a tool to generate realistic Grid topologies (which is supported by the NSGrid simulation core). Also, additional storage/computational resources can be added to the topology to allow simulation of dedicated storage/computational server farms (e.g. used for rendering). A multitude of task scheduling algorithms (e.g. network aware scheduling, service aware scheduling, application level scheduling) are available from NSGrid and can be employed to schedule user submitted tasks on the MicroGrid/MacroGrid resources. Also, advanced management components such as replica managers, checkpointing, service class managers (providing QoS support) can be instantiated and used in Micro/Macro Grid simulations. Once a suitable media company profile has been selected/constructed, users can automatically generate a Tcl script describing the scenario's topology, together with the different company profiles to NSGrid. The NSGrid simulator itself has been *gridified* in that it is able to run in a Grid environment (all NSGrid simulations described in section 5.6 were run on an LCG-2.6.0 Grid [12] comprised of dual Opteron 242 1.6Ghz worknodes with 2 GB RAM per cpu, and operating under Scientific Linux 3).

5.6 Simulation results

In what follows, MediaNSG will be used to construct realistic MediaGrid topologies and simulate some proof-of-concept MediaGrid situations. In all simulations presented here, each user was associated with a computational resource, with all computational resources having equal reference processing capabilities. We limited storage resource/archive access to a maximum read/write throughput of 5600 Mbps (which is realistic as a proof-of-concept storage element array interconnected by fiber channel technology [13] and attaining these speeds is deployed in the FIPA project). User tasks with a duration of 100 percent were mapped to a simulated duration of 3600 seconds (e.g. from figure 5.4 we can see that each Quality Checking/HiRes browse task by a producer/director takes 1080 seconds to complete on a reference processing element).



(a) MicroGrid ring topology (b) MicroGrid point-to-point topology

Figure 5.6: MicroGrid topologies

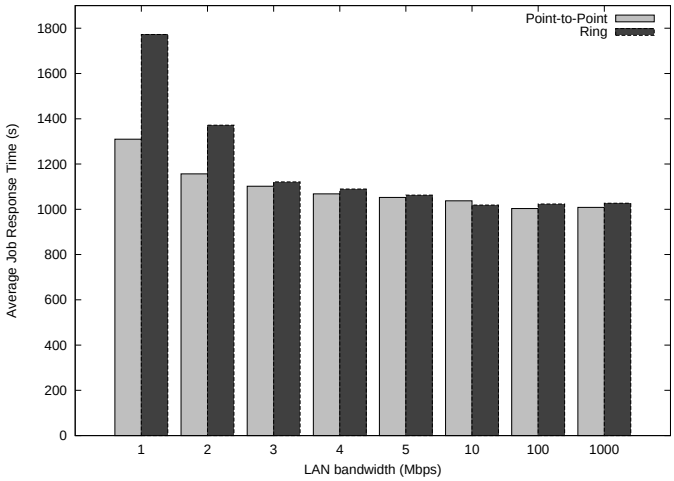


Figure 5.7: Influence of topology on average job response time - network aware scheduling

5.6.1 MicroGrid topology

In a first batch of simulations, we constructed a Television Broadcasting MicroGrid, and parametrized LAN network bandwidth (LAN interconnections are in this case the network connections between the different clients and the central MicroGrid storage/archiving element containing I/O data for the different tasks) from 1 Mbps to 1000 Mbps. We mapped low, medium and high CPU usage to respectively 10, 50 and 90 percent processor utilization on a reference processor in this simulation. In this first case a point-to-point connection between clients and storage element was provided and network aware scheduling was employed. We measured the average job response times (we define job response times as the difference between the time the job ends, and the time it was submitted to the scheduling service) and notice (see figure 5.7) that tasks experience serious delays when MicroGrid LAN bandwidth is less than 2 Mbps (due to the fact that data input/output retrieval/storage is blocked by network congestion, thereby stalling computational progress). It is also interesting to note that the difference between 10 Mbps and 1 Gbps interconnections is relatively small (on average tasks took 29 seconds longer to run when 10 Mbps network technology was used) as the network is no longer posing a bottleneck.

In the next batch of simulations, we changed the point-to-point network connection from clients to the storage/archiving element to a ring topology. The results show that at least 3 Mbps interconnections are needed if tasks are to execute without substantial delays. This can be explained because of the ring configuration of the network topology, network congestion on one link will influence more than one client's job response times (as opposed to the point-to-point network). It is interesting to note that when network bandwidths of 5 Mbps or more are available, the difference between point-to-point connections and ring connections is virtually non-existent, while a ring topology has the benefit of protecting client machines from single link failures.

The average job hopcount (number of hops storing job output data from computational to storage resource, plus the number of hops for retrieving job input data from data resource to computational element) when a ring topology was used is 7.88, while hopcount when using point-to-point connections is 2.

5.6.2 MacroGrid resource sharing

In this simulation, we connected the Television Broadcasting MicroGrid to a remote computational resource provider, offering 5 additional processing elements capable of running at twice the reference processor speed (point-to-point connected to a gateway). We again parametrized MicroGrid LAN bandwidth (and used a point-to-point topology), and low, medium and high task processing requirements in this simulation were respectively set at 20, 100 and 180 percent

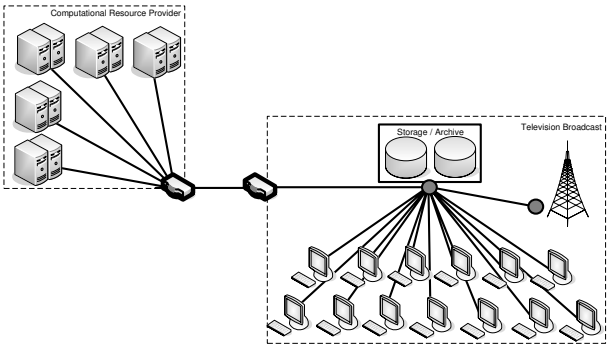


Figure 5.8: Simulated MacroGrid topology

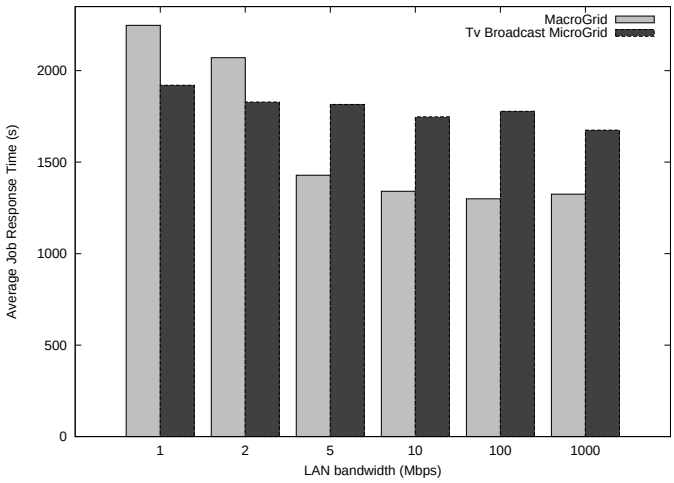


Figure 5.9: Influence of MacroGrid resource sharing on avg. job response time

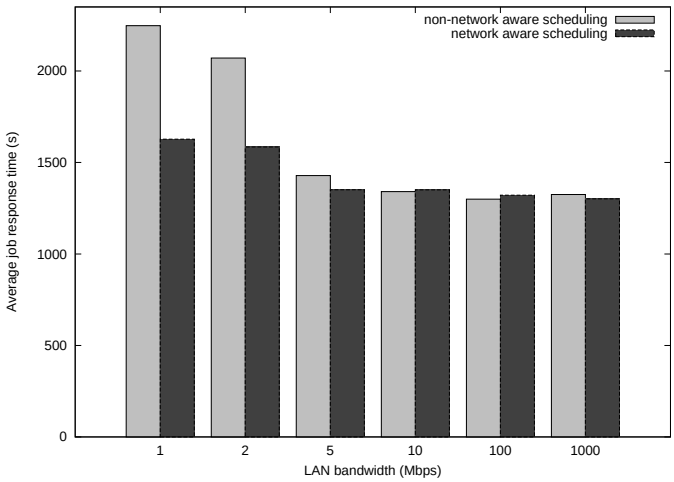


Figure 5.10: Network unaware vs. network aware scheduling

of a reference processor. Jobs running at 180 percent can be processed on the Television Broadcasting MicroGrid, but will run at 0.55% of their normal speed. The link connecting the remote resource provider to the Television Broadcasting MicroGrid is a dedicated link, and in each simulation, it was given the same bandwidth as the Television Broadcast company's LAN bandwidth. The television broadcast's scheduling service (utilizing a network unaware scheduling algorithm) queried both its own information service and the computational resource provider's information services for resources adhering to the job's requirements, and both returned status information regarding the provided resources.

From figure 5.9 we can see that the average task response times drop significantly when the Television Broadcast MicroGrid is given access to the computational resource provider's assets *if the interconnecting link bandwidth does not go below 5 Mbps*. Indeed, since network unaware scheduling is used, the scheduler looks at the state of available computational and storage resources, but does not take into account the state of the network links interconnecting these resources. If the available link bandwidth between MicroGrid sites drops, it can be detrimental to schedule jobs for processing on remote resources (as can be seen on figure 5.9 for bandwidths of 1 Mbps and 2 Mbps), since the network links connecting these resources to the job's originating site's storage element will become a bottleneck.

If the network does not hamper computational processing (5 Mbps or more interconnections), average job response times were up to 30.37 percent better when MacroGrid's resource usage services were being used (in this case allowing migration of jobs from the TV broadcast company to the computational resource provider).

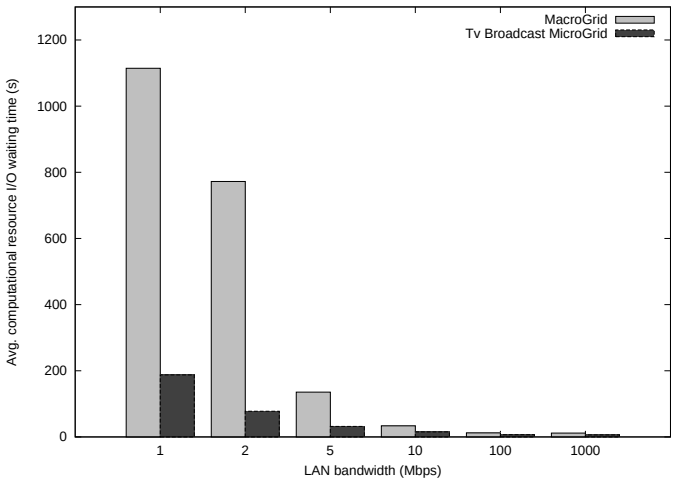


Figure 5.11: Computational resource blocking times

5.6.3 Network aware versus network unaware scheduling

To overcome the problem described in the previous simulation, a network aware scheduling algorithm needs to be employed. We simulate the same MacroGrid topology as in section 5.6.2 and compare average job response times when scheduling jobs by using network aware on one hand and network unaware scheduling algorithms on the other. From figure 5.10 we see that at low bandwidths, the average job response time no longer exhibits bad performance when using a network aware scheduling algorithm. This is due to the fact that the network aware algorithm will schedule jobs for processing on remote resources only if the network links connecting these resources to the required storage resource/archive support transferring the job's I/O data at sufficient speeds so as to not waste reserved processing time.

5.6.4 Resource efficiency

From the simulations performed in section 5.6.2 we calculated the average amount of time during which a computational resource was reserved for a job without being able to continue processing (i.e. 'idling'), because job processing was blocked while waiting for necessary input/output data to be received/sent. From figure 5.11 we can see that utilizing remote resources (the MacroGrid case) is not efficient if available connection bandwidth drops below 5 Mbps. At 1 Mbps local MicroGrid task processing capabilities are being hindered by network bottlenecks towards the local storage element/archive.

5.7 Conclusions

Since media production/broadcast companies are more and more evolving to file based media handling instead of tape based, and since these companies tend to have high requirements regarding Quality of Service, the need to integrate state-of-the-art IT technology in this domain is becoming mandatory. A second important evolution in this field is the need for collaboration amongst media companies, not only on application level, but also to share resources and media repositories. Grid computing can offer a solution in this case, with support for resource/data sharing and advanced collaborative virtual organizations crossing media company boundaries. We therefore propose the use of a MediaGrid architecture consisting of MicroGrid and MacroGrid components. The development of suitable scheduling and service management algorithms in a MediaGrid context, can only happen thoroughly if multiple collaboration scenarios are investigated. To this end, we have developed MediaNSG, a media company specific Grid simulator built on top of NSGrid. Typical task, user and media company profiles were presented, and a set of proof-of-concept simulations was discussed.

References

[1] S. Gilheany. *Projecting the Cost of Magnetic Disk Storage Over the Next 10 Years*. http://www.archivebuilders.com/whitepapers/22011p.pdf, 2001.

[2] *EBU Project Group P/Meta Metadata Exchange Scheme, V. 1.0*. http://www.ebu.ch/en/technical/trev/trev_290-hopper.html.

[3] T.J. Harmer and al. *GridCast - Using the Grid in Broadcast Infrastructures*. In Proceedings of UK e-Science All Hands Meeting (AHM03), 2003.

[4] T.J. Harmer et al. *GridCast - A Service Architecture for the Broadcasting Media*. In Proceedings of UK e-Science All Hands Meeting (AHM04), 2004.

[5] *FIPA - File based Integrated Production Architecture Project*. https://projects.ibbt.be/fipa/.

[6] *Force10 Networks White Paper - Building Scalable Digital Media Post Production Networks: The Role of Ethernet*. http://www.force10networks.com.

[7] *SGI White Paper - Broadcast Media Management in a Data-Centric Workflow*. http://www.sgi.com.

[8] *VRT - The Flemish Radio- and Television Network*. http://www.vrt.be/vrt_master/vrt_en_homepage/index.shtml.

[9] *Video Promotion*. http://www.videopromotion.be/.

[10] D. Lu and P. Dinda. *Synthesizing Realistic Computational Grids*. In Proceedings of ACM/IEEE Supercomputing 2003 (SC 2003), 2003.

[11] D. Lu and P. Dinda. *GridG: Generating Realistic Computational Grids*. ACM SIGMETRICS Performance Evaluation Review, 40(4), 2003.

[12] *LHC Computing Grid project*. http://lcg.web.cern.ch/LCG.

[13] Tom Clark. *Designing Storage Area Networks - A practical reference for implementing fibre channel and IP SANs*. Addison-Wesley Professional, 2003.

6
Overall Conclusion

In this work we have investigated the use of network status and application monitoring information when assigning resources to jobs and/or applications on a Grid. To this end we have developed:

- a scalable and performance tuned Grid monitoring architecture
- NSGrid, a Grid simulator focussed on modelling accurate application control and data transfers
- network and service aware Grid scheduling algorithms allowing faster average job response times and more efficient Grid resource useage
- a distributed Grid service management architecture with multiple resource-to-service partitioning heuristics

The main conclusion of the work comprised in this thesis can be stated as follows: when up-to-date and accurate Grid application and resource status monitoring information is available, the Grid infrastructure can be utilised more efficiently by taking into account the state of the network and the characteristics of the different Grid service classes when scheduling Grid jobs and/or managing these service classes. By differentiating between service classes and allocating Grid resources based on these classes' characteristics, a relatively easy to implement advance reservation mechanism can be deployed in a Grid.

To reach this conclusion, we first presented a well-performing, scalable and portable Grid monitoring framework. Performance was obtained by using C++ as

base implementation language, together with caching mechanisms at key locations (e.g. producers caching sensor data, eliminating the need for producers to contact sensors directly); portability then dictated the use of appropriate middleware for which we chose the Adaptive Communication Environment, the GTop libraries for monitoring resource status and Java based consumers. Scalability was achieved by using a "Grid Monitoring Architecture"-compliant architecture consisting of sensors, producers, consumers and a decentralized directory service. Multiple ready-to-use consumers (e.g. network usage prediction/failure detection, real-time visualization, archiving) have been implemented, and the information service offers a fast resource matchmaking portal for use by management components.

We compared the functionality, performance and intrusiveness of our monitoring framework to that of Globus MDS2 and its successor, the GT3.2 web services based information service. Satisfying results were presented in terms of query throughput and response times, both for our producers and directory service. Monitored resource properties (e.g. typical processing capabilities, network bandwidths) and state information data (e.g. failure probabilities) allow construction of realistic Grid topology descriptions for simulation purposes.

Noting the absence of a Grid simulator providing accurate network resource modelling, we developed NSGrid, a Grid simulator built on top of the ns-2 network simulator and capable of accurately modelling network traffic between different Grid resources and management components. Computational, storage and data resource models were discussed, along with job models and the functionality and interoperation of the different management components: scheduler, connection manager, service manager, service monitor, information service and replication manager.

In order to demonstrate the usefulness of a network aware Grid simulator like NSGrid, different Grid scheduling algorithms (some network aware while others network unaware) were detailed and their performance was evaluated on a sample Grid topology. The results showed that whether data was pre-staged or accessed in parallel with the job's execution (i.e. streamed), accurate network status information allowed to create significantly better schedules in terms of both job response time and computational resource efficiency. With Grid site interconnection bandwidths of 10 Mbps we measured average job response times that were 61 percent better than when no network information was included in the scheduling process. For the same scenario, we showed that computational resource reservations spent on average 30 percent of their time idling, while waiting for I/O data to arrive/be sent (which can be avoided when taking into account the state of network resources at the time of scheduling). Our simulations also showed that upfront reservation of bandwidth (between Grid resources) for different service classes can improve the average job response times by avoiding that data-intensive service classes monopolized available bandwidth.

Additional research into the area of Grid service class differentiation yielded a distributed service management architecture capable of monitoring service characteristics at run-time and partitioning Grid resources amongst different priority service classes. This partitioning, together with the dynamic creation of per-service management components, lead us to the introduction of the "Virtual Private Grid" concept. A variety of resource-to-service partitioning algorithms (based on genetic algorithm heuristics) were discussed and we evaluated their performance on a sample topology using NSGrid. The results showed that the proposed service management architecture can improve both computational and network resource efficiency (with average improvements of 17 and 5.5 percent respectively), combined with lower average job response times (when employing a network aware scheduling algorithm this can lead to 30.5 percent better job response times) and that it became possible to automatically enforce service class priorities. Management complexity and scheduling / information service scalability was improved due to the automated deployment of service class dedicated management components.

We concluded our Grid research contributions by presenting a use case based on a recent trend in Grid computing: the deployment of Grid technology in the audio/visual production industry. As media production/broadcast companies are more and more evolving to file based media handling instead of tape based, and since these companies tend to have high requirements regarding Quality of Service, the need to integrate state-of-the-art IT technology in this domain is becoming mandatory. A second important evolution in this field is the need for collaboration amongst media companies, not only on application level, but also to share resources and media repositories. Grid computing can offer a solution for these problems, with support for resource/data sharing and advanced collaborative virtual organizations crossing media company boundaries. We therefore proposed the use of a MediaGrid architecture consisting of MicroGrid and MacroGrid components. The development of suitable scheduling and service management algorithms in a MediaGrid context, can only happen thoroughly if multiple collaboration scenarios are investigated. To this end, we have developed MediaNSG, a media company specific Grid simulator built on top of NSGrid. Typical task, user and media company profiles were presented, and a set of proof-of-concept simulations was discussed.

In the future, we expect work to continue on extensions to some of the key aspects discussed in this PhD thesis. In particular, decoupling the NSGrid Grid modelling layer from the underlying ns-2 network layer and providing a generic interface to discrete event network simulators (e.g. Dartmouth SSF) would allow for more scalable Grid simulations. It would also be interesting to implement the presented network and service aware Grid scheduling algorithms in a real-life Grid middleware scheduling system as this would allow us to accurately determine the

error margins when simulating Grid behaviour with NSGrid, which in turn would allow us to tune resource and management component delays to better reflect the behaviour of a particular Grid middleware solution. A third notable extension would be the conception, implementation and evaluation of additional resource-to-service partitioning algorithms (particularly with regard to network resource partitioning) for the presented Grid service management architecture.

A

Grid Computing: The Next Network Challenge!

B. Volckaert, P. Thysebaert, M. De Leenheer, F. De Turck, B. Dhoedt, P. Demeester

published in The Journal of The Communications Network (Proceedings of FITCE 2004), 43rd European Telecommunications Congress, 2004, Vol. 3, pp. 159-165.

Abstract *For the last five years, grid computing has been a very hot and fruitful research theme resulting now in the deployment of the first operational grid systems. The main motivation for this new computing paradigm lies in the observation that the demand for computational and storage resources is ever growing while on the other hand vast resources remain underused. The grid paradigm aims at solving this mismatch by offering its users computational and storage resources transparently, making abstraction of the exact geographic location of the physical resource (this approach has appealing similarities to the power grid, hence the term "grid computing"). Despite the current deployment of operational grid systems, important challenges still lay ahead. New applications, opening the grid also for commercial exploitation, impose new requirements in terms of e.g. security, scaling behaviour, Quality of Service and robustness. In particular the geographic spread of grid users in combination with these new requirements will certainly have drastic consequences for the communication infrastructure. In this*

paper, an overview of current grid systems, application taxonomy and emerging trends will be discussed. The implications for the network will be analysed, taking current grid deployments as a starting point. The importance of co-management of computational/storage and network resources will be illustrated.

A.1 The concept of Grid computing

A.1.1 Historical background

Supercomputers and clusters have been the main workhorses to process computationally complex problems, often originating from the scientific community. However, problems are becoming increasingly demanding, challenging the capabilities of even the most powerful single supercomputer or cluster system. This observation led to the idea to join forces for solving these problems in a reasonable time frame, interconnecting these remote computational and storage resources into a single number crunching system: the Grid.

As a first step in realizing this concept, the maturation of the Internet in the nineties led to the first global distributed computing projects. Two projects in particular have proven that the concept works extremely well. The first project, distributed.net, used thousands of independently owned computers across the Internet to crack encryption codes. The second is the SETI@home project [1]. Over two million people have installed the SETI@home software agent since the project's start in May 1999. This project proved that distributed computing could accelerate computing project results while at the same time managing project costs (IBM's ASCI White supercomputer is rated at 12 TeraFLOPs and costs $110 million. SETI@home currently gets on average 15 TeraFLOPs and has cost $500K so far).

The term "Grid computing" suggests a computing paradigm similar to the operation of an electric power grid: a variety of resources contribute power into a shared pool for consumers to access on an as-needed basis. Although this ideal is still a few years off, key efforts are emerging to define standards allowing the easy pooling and sharing of all computing, storage, data and network resources in a way that can promote mass adoption of grid computing.

A.1.2 What's in a name: Grid computing or cluster computing?

Grid computing differs from cluster computing in a number of key aspects. First, due to the geographic distribution of Grid resources, a Grid does not have a central administration point (instead it uses resources across multiple administrative domains), whereas all cluster resources can usually be administered from one loca-

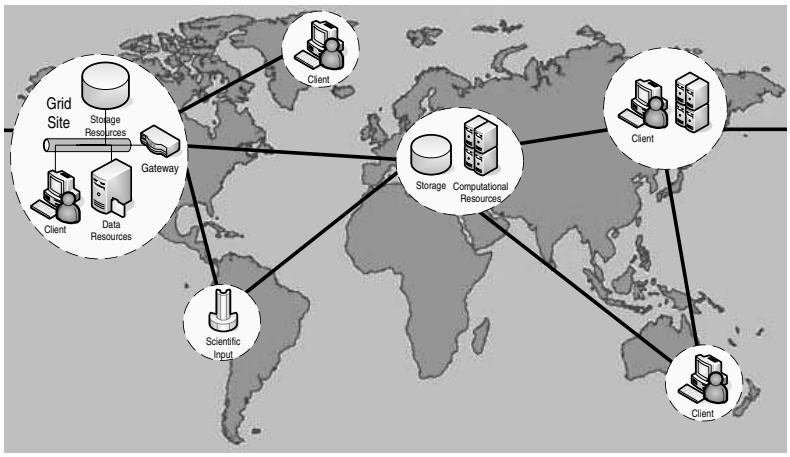


Figure A.1: Global computing Grid

tion. Second, the same geographic distribution usually entails drastically different resource usage policies and heterogeneity of equipment: a variety of resources will be connected by a wide range of network technologies, whereas a cluster will usually consist of a large collection of homogeneous resources interconnected by a proprietary bus or high speed / short range network links. This again indicates an important distinctive Grid feature: communication links can be long haul, possibly subject to congestion, while the Grid topology itself is subject to frequent change, due to the possibly dynamic nature of resources and the decentralized authority over resource usage (this dynamic behaviour can easily be spotted in the case of SETI@home type grids, based on desktop pc's donating unused CPU cycles).

A.1.3 No one-size-fits-all: different Grid systems for different application areas

Based on their main application area, grid systems can be divided in three classes: Computational grids, Data grids and Service grids. Computational grids can offer more processing power than any of its constituent machines. In this category, one can further distinguish "distributed supercomputing" and "high throughput computing" categories. The former class attempts to shorten execution time of a task by processing the task in parallel on multiple machines, while high throughput computing systems are tuned to process large job batches (for instance parameter sweep jobs). Cycle scavenging grids are special cases of computational grids: they allow desktop users to donate their idle CPU time to help scientific research (mostly global interest projects e.g. SETI@home, fightAIDS@home, etc.).

A second grid system class is coined the term "Data Grid". These synthesize

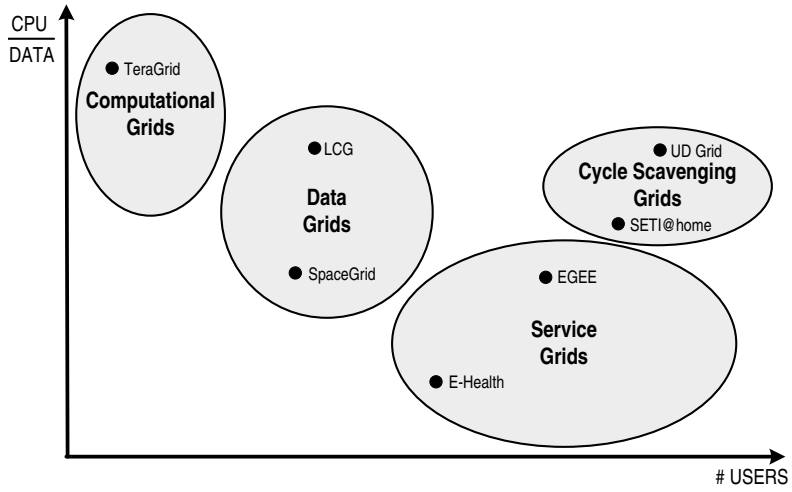


Figure A.2: Grid taxonomy

new information from distributed data repositories. The LCG project (i.e. the processing of data generated by CERN's Large Hadron Collider [2]) is a well-known example of this type of grids.

Service grids offer services that cannot be provided by a single machine. Services can range from collaborative working (enabling interaction between users and applications through a virtual workspace), to multimedia grids and "on demand" grids (enabling a user to dynamically increase the amount of machines processing on its jobs or even to dynamically select dedicated equipment to realize a possibly virtual experiment). A thorough look at Grid taxonomy can be found in [3].

A.1.4 Grid standardization

The Global Grid Forum (GGF [4]) acts as standardization body for the Grid, comparable in terms of philosophy as the IETF for Internet related matters. The GGF is a community-initiated forum of thousands of individuals from industry and research leading the global standardization effort for grid computing. The GGF's primary objectives are to promote and support development, deployment, and implementation of Grid technologies and applications through the creation and documentation of "best practices" - technical specifications, user experiences, and implementation guidelines. Similar to the Internet, adopting common standards and promoting the use of these standards will allow clients to connect transparently to multiple grids and eventually facilitate grid interworking.

A.2 The Grid today

A.2.1 Generic Grid management architecture

Just like an electrical power grid needs advanced control functions and infrastructure to assure proper operation, managing resources is a vital Grid function. A Grid management architecture consists of three fundamental building blocks: resource discovery, resource advertisement and a scheduler capable of assigning Grid resources to jobs in an intelligent manner. Grid applications need to formulate their resource needs (processing elements, storage elements and communication and data requirements) and their respective Quality of Service requirements in job requests. The resource management architecture contains an admission control component that decides if an incoming job / resource request can be accepted. In order to be able to schedule jobs on Grid resources in an intelligent way, the Resource Monitoring component provides resource information (both static properties and dynamic status) to the scheduler. Other notable management components are the naming service, which provides a distinct name to each Grid resource, and the reservation manager that keeps track of all resource reservations. Once a job has been scheduled, the execution manager is responsible for starting the Job on the assigned resources. The job monitoring component prevents a job from violating its allocated resource utilization contract and the resource policy (enforced by the policy manager). This allows the resource management architecture to offer Quality of Service to Grid applications. The Job Monitor is also responsible for reporting the job's resource usage to the external Accounting/Billing components.

Besides having a Grid application interface, the resource management system also has an interface with the Grid resources' native operating system (in order to be able to execute jobs on resources and provide job and resource monitoring), with the Security Manager (the resource managers must authenticate and authorize all resource requests with the Grid security manager), and with the accounting manager (responsible for issuing Grid resource usage bills to consumers).

A.2.2 Glueing Grid resources: Grid middleware systems

Grid middleware refers to a service layer that operates between the Grid resources on the one hand and applications on the other: it attempts to transparently connect networks, computational resources and data resources into one Grid that can encompass different architectures, operating systems and physical locations, and provides the tools for applications to communicate and collaborate effectively. The Globus toolkit [5] is one of the most advanced and widely deployed Grid middleware suites, offering a set of Grid services and software libraries to construct computational, data- or service grids. Globus implements most of the Global Grid Forum's specifications and provides components for resource management (Grid

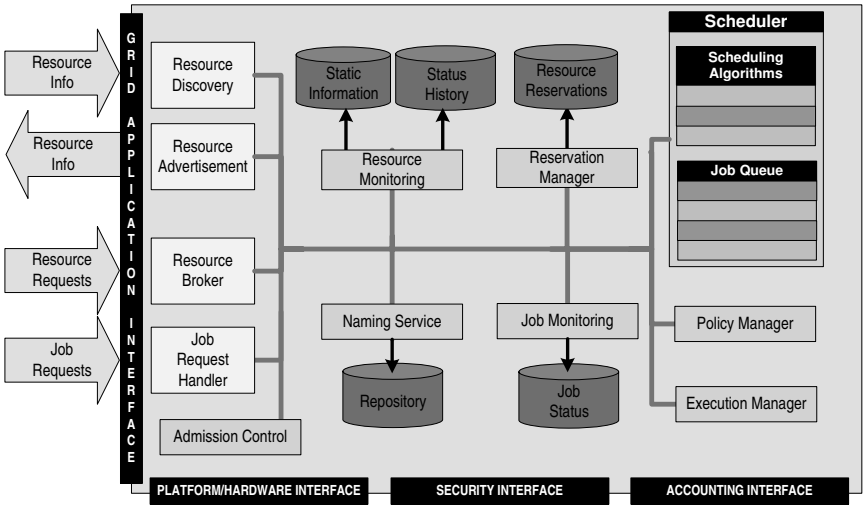


Figure A.3: Generic Grid management architecture

Resource Allocation & Management Protocol), information services (Monitoring and Discovery Service), security services (Grid Security Infrastructure) and data movement/management (Global Access to Secondary Storage and GridFTP). Grid developers typically pick the Globus services they need and incorporate those in their Grid-enabled application. An important example of Globus' usage in current scientific grids is the Large Hadron Collider Computational Grid middleware (LCG). The LCG middleware is built on services provided by Globus and components developed by the European DataGrid project, and provides a Grid capable of processing large amounts of data (further details are given in the LCG case study).

Another notable middleware suite, Legion, provides an object-based framework in which resources can be combined into a computational Grid. Legion is not a "sum of services" architecture (as Globus is), but rather a completely integrated architecture, allowing for fast Grid deployment.

A.3 Grid applications: case studies

A.3.1 Cycle Scavenging Grids

United Devices Grid [6] offers desktop users the ability to donate their unused processing power and thereby help achieve scientific progress in finding new cures for life-threatening diseases. Global participation has passed the 2.5 million mark in 2004. UD Grid is now the largest public grid in operation, helping the Cancer

Project	Grid Category	CPU	Storage	Network
EGEE	Service	1000	10TB	2.5-10 Gbps
LCG	Data & Comput.	1075	15TB	10Gbps(core)
TeraGrid	Comput. & Data	6174 + 1.1TF	1PB	40Gbps(core)

Table A.1: Grid case studies

Research Project, the Anthrax Research Project and the current PatriotGrid project, designed specifically to identify new leads for cures to bioterrorism diseases. Due to the participation of numerous users, UD Grid projects have achieved record levels of speed and success in processing data. Other notable cycle scavenging grid projects are SETI@home and fightAIDS@home.

A.3.2 Dedicated scientific computing and data Grids

TeraGrid [7] is a US-based virtual computing infrastructure, built from five individual clusters, offering 1 Petabyte of storage, and linked by a cross-country 40Gbps core network. The goal of TeraGrid is to provide scientists with a dedicated computing and storage infrastructure delivering the power needed to cope with large data set analysis and high resolution simulations and, in the process, speed up scientific discovery.

Another major example of this type of Grid is the Large Hadron Collider computational Grid project (LCG). In this project, a worldwide computational grid will be deployed to cope with the massive data flow that will come from CERN's Large Hadron Collider (in 2007 the LHC will roughly generate between 12 to 14 Petabytes of data). Data analysis requirements are in the order of 70.000 of today's fastest PCs.

A.3.3 Service Grids

The Enabling Grids for E-Science project in Europe (EGEE [8]) aims to develop a dedicated service Grid infrastructure in Europe, available to scientists 24 hours-a-day. The project focuses on building a consistent, robust and secure grid network (built on the GANT research network), offering a large variety of production-quality services to the user. For the moment, the two main service interests lie in the area of Biomedical Science and High Energy Physics, but more services will be added at a later stage.

A.3.4 Grid services meet web services

The Globus toolkit is, as of 2003, evolving towards a service-based Grid concept with the development of the Open Grid Services Architecture (OGSA), offering

complete virtualization of the Grid resources. The goal of OGSA is to standardize all the services one finds in a Grid application (job management services, resource management services, security services, accounting services, etc.) by specifying a set of standard interfaces for these services. The Web Services Resource Framework (WSRF) was chosen as the base of this architecture, due to the fact that web services offer a way for accessing diverse services/applications in a distributed environment. The WSRF standard is being developed partly by the Grid development community and partly by the Web Services community, as it allows Grid services to be treated as extended web services.

A.4 New Grid trends

A.4.1 Emergence of new applications

As the ability to tap into the Grid's power is getting closer and closer to the end-user's home, the need to support new (not necessarily scientific) Grid applications arises. In what follows we present some of these next-generation Grid applications and their associated requirements:

- **Multimedia Editing**: With the advent of high-definition video, and a growing interest in home video editing, end-users need processing capabilities exceeding those of the regular personal computer. Computational Grids can offer end-users the possibility to use industry level video editing capabilities.
- **Collaborative Working**: Collaborative working focuses on bringing Grid benefits as on-demand processing power and data availability together in an integrated environment, giving people at different locations all the necessary tools to be able to work on a common project.
- **Online Gaming**: Massively Multiplayer Online games, where thousands of players can simultaneously interact with each other in a virtual world, are becoming increasingly popular. In the near future, as the amount of players grows, the resource limitations of the game provider will be reached. Grid technology can offer a cost-effective scalable solution, increasing computational power by assigning more resources during gaming rush hours, and solving the network bottleneck by distributing the workload.
- **Virtual Reality**: New advances in screening technologies allow people to experience realistic 3D environments through a variety of Virtual Reality applications (e.g. entertainment, exploring new architectural structures, simulator training, etc.). The more realistic the scenario, the more processing power involved (especially when rendering complex environments in real-time), giving an incentive to move to Grid computing.

Application	CPU	Data	Netw.	Robust	Users	Secure	Realt.
Online Gaming	++	++	++	+	+++	+	+++
Virt. Reality	+++	++	++	+	+	+	+++
Collab. Working	++	++	++	++	++	+++	+++
MM Editing	+++	+++	+++	++	+	+	+
Data Mining	+++	+++	+++	++	++	+++	+

Table A.2: New Grid trends and requirements

- **Data Mining**: Effective data mining is being hampered by the massive amounts of data available in today's digitized age. Filtering this data requires huge amounts of processing and storage, unavailable to most end-users. Since data mining is becoming critical for launching effective marketing campaigns, companies are more and more interested in the benefits (faster results, cost-effective processing on an as-needed basis, etc.) of Grid technology.

A.4.2 Telecom oriented Grid management

In [9], the Grid is defined as follows: "A computational Grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities". This definition was later extended to include usage policy issues, but the message remains clear; when broad adoption of grids becomes a reality, we should treat computing and storage not as an asset, but instead as service. Typically, this service will be purchased, on demand, from one or more service providers. This might be your neighbour who shares his spare CPU cycles, or it might be a dedicated service provider. These Grid Service Providers will usually house powerful computational and storage resources, and charge customers for the used capacity and time. Obviously this scenario poses several challenges for Grid related research. For instance, how should a provider dimension his network, and, for a given setup, how should prices be determined and varied over time?

Additional difficulties arise if we consider the fact that a paying customer will not tolerate degraded service levels. First and foremost, a user-submitted job should be executed until completion, even in case of failing resources. This implies that we need a resilient Grid environment, which can handle problems quickly and efficiently with minimal (preferably even no) intervention. This requirement has implications on Grid resource management strategies and components (resource allocation, reservation management, job scheduling), since they must be able to react to network link failures, malfunctioning hardware, software errors, etc. A natural consequence of this service-oriented view of grids is the introduction of

service classes. This gives providers the opportunity to offer added value on top of their basic service, which essentially is the successful completion of a user-submitted job. Which service classes the providers will offer remains to be seen, but an obvious example is a deadline guaranteed service in which the user's job will be completed before a specified deadline. Offering QoS on the Grid also poses several scientific challenges, since it must be based on the independently provided QoS levels of the network and the resources.

A.5 Grid network importance

A.5.1 Network aware Grid scheduling

Traditionally, computing jobs have been scheduled on multiprocessor or cluster systems using algorithms that ignore network parameters such as bandwidth and delay. The rationale behind this approach is that (i) the interconnection bandwidth between processing elements is high, (ii) input data is readily available at the processing site and (iii) the overall time spent transferring input and output data is negligible in comparison with the total job duration, given its computational complexity. Given the distribution of resources in a Grid environment (resulting in greater network delays between two distant resources) and the size of the data to be moved around for a typical Grid job, it becomes clear that this approach is suboptimal: the time spent in transferring data can no longer be neglected, and if it is transferred between distant resources, network bottlenecks can severely block a job's computational progress.

It follows that in order to generate quality schedules (both from the end user's point of view - the experienced job turnaround time - and the provider's point of view - efficient use of the interconnecting network), data location and network status need to be incorporated. In particular, decisions must be taken regarding (i) the replication of input data sets to multiple locations and (ii) the allocation of available network bandwidth to the different jobs.

The latter requires that a Grid scheduler is able to query the status (available bandwidth) of network links, and make bandwidth reservations for jobs. In other words, it is necessary that the network can be treated as a manageable resource, which in turn requires the presence of a Network Management Infrastructure that implements all these operations.

If all resources (including network) are manageable, it becomes possible to partition the Grid's aggregate resources into subsets; each subset can be dedicated to a single class of applications, or can be reserved for users requesting a specific class of service.

In figure A.4 and figure A.5, we have visualized both user-experienced and provider-experienced improvements in schedule quality when scheduling jobs from

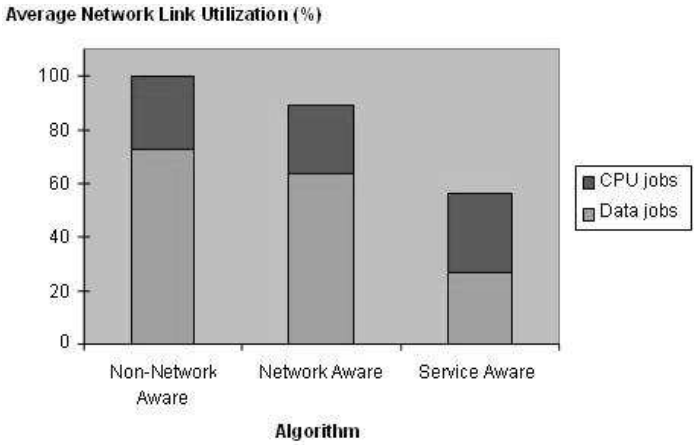


Figure A.4: Average network link utilization

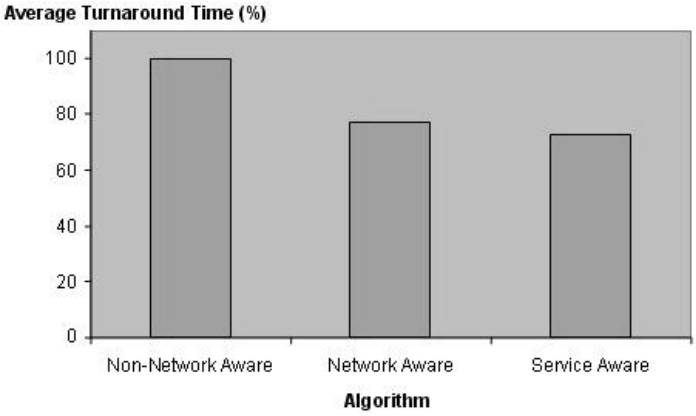


Figure A.5: Average turnaround time for data jobs and cpu jobs

different application classes using either traditional (network unaware) and more refined (network and possibly application class aware) heuristics. The results were obtained from the simulation of a Grid consisting of 5 different sites, each providing local resources and a number of end-users submitting jobs. It is clear that network-awareness in the Grid scheduler can improve the average job turnaround time significantly. In addition, when jobs fall into distinct application or service classes, this knowledge can be exploited by providers to create more efficient schedules in terms of resource utilization (as is shown for the network resources), while preserving the overall job turnaround time. A more in-depth discussion of these results can be found in [10–12].

A.5.2 Management and control in the optical transport layer

Incorporating network information in a scheduler not only implies that the scheduler will decide which computational and storage resources a job should use, but also how data should be transported through the network. In other words, the grid traffic is routed through the network by the scheduler. In order to exploit the grid's capacity to its full extent, resources should be connected through high capacity links. Advances in optical and fiber technology such as Dense Wavelength Division Multiplexing (DWDM), allow us to make these kinds of connections between grid resources. Since we expect optical technology to remain the dominant technique for building high-speed networks, an important challenge is to optimize the interface between grids and optical networks.

Broadly speaking, there are three methods to send data over an optical network. The first is called circuit switching, and works by reserving a dedicated wavelength on all links of the path between the sender and receiver. Although setting up a path takes a non-negligible time, the end points have exclusive access to the path once it is in place. Not surprisingly, this technique is very attractive when large datasets need to be transmitted. A second technique, called packet switching, sends small amounts of data in packets. These packets are sent independently of each other. A router, receiving a packet, has to inspect the header and make a routing decision. A third recently developed technique called Optical Burst Switching (OBS [13]), combines features of both packet and circuit switching. Using OBS, a control packet is sent first, which reserves a wavelength for a specified amount of time at each router it passes. The control packet is closely followed by the actual data (a burst), which does not wait for a confirmation of the reservation. Note that OBS is only usable for medium-sized datasets; in practice this means a burst length smaller than 100 ms (longer bursts are better transmitted by circuit switching), or about 10 MB on a 1 Gbps link.

One could think of assembling a complete grid job (code and data) in one burst. If the optical routers maintain some information about where resources

are located, and if this information becomes more precise when approaching the resources, an intermediate router should be able to guide the burst in the direction of the resources. The router could dynamically make a routing decision for each burst based on the available resource status information. Ultimately, this could lead to a large scale, self-organizing optical network.

A.6 Conclusions

Grid systems are reaching the maturity level necessary to initiate production grade roll-out initiatives, leading to an increasing user community in scientific and business sectors. Development of standards and wide spread usage of toolkits based on these standards are signs of this maturity level. Research programs like EGEE and TeraGrid indeed aim at large-scale deployment, opening up vast computational and storage resources to many. The success of these high-performance Grid systems will undoubtedly lead to adoption of Grid technology for small companies and even residential users (just like the Internet itself has evolved from a purely research infrastructure to a commodity service for everybody). As suggested above, development of Grid intensive applications for residential end users will lead to a paradigm shift towards service oriented Grid infrastructures. New requirements relating to e.g. service guarantees, resilience and security will arise.

It becomes increasingly clear that together with the expected growth of Grid usage, important network challenges will need to be handled. Co-scheduling of network resources and computational/storage resources has been presented as a promising avenue towards cost-effective Grid exploitation. On the longer term, a vision of a self-organizing optical network infrastructure designed specifically for massively scalable Grid operation has been presented.

References

[1] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. *SETI@home: An Experiment in Public-Resource Computing*. Communications of the ACM, 45:56–61, 2002.

[2] *LHC Computing Grid project*. `http://lcg.web.cern.ch/LCG`.

[3] K. Krauter, R. Buyya, and M. Maheswaran. *A Taxonomy and Survey of Grid Resource Management Systems*. International Journal of Software: Practice and Experience (SPE), 32:135–164, 2002.

[4] *Global Grid Forum*. `http://www.gridforum.org/`.

[5] *The Globus Alliance*. `http://www.globus.org/`.

[6] *United Devices Grid*. `http://www.grid.org`.

[7] *The TeraGrid project*. `http://www.teragrid.org/`.

[8] *Enabling Grids for E-Science in Europe*. `http://egee-intranet.web.cern.ch`.

[9] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[10] B. Volckaert, P. Thysebaert, M. De Leenheer, F. De Turck, B. Dhoedt, and P. Demeester. *Network Aware Scheduling in Grids*. In Proceedings of 9th European Conference on Networks & Optical Communications (NOC), pages 311–318, 2004.

[11] P. Thysebaert, B. Volckaert, F. De Turck, B. Dhoedt, and P. Demeester. *Evaluation of grid scheduling strategies through NSGrid: a network-aware grid simulator*. published in Neural, Parallel & Scientific Computations, Special Issue on Grid Computing, 12:353–378, 2004.

[12] B. Volckaert, P. Thysebaert, F. De Turck, B. Dhoedt, and P. Demeester. *Application-specific hints in reconfigurable grid scheduling algortihms*. published in Lecture Notes in Computer Science (LNCS 3038), Proceedings P3 of ICCS 2004, LNCS 3038:149–157, 2004.

[13] C. Qiao and M. Yoo. *Choices, Features, and Issues in Optical Burst Switching*. Optical Networks Magazine, 1:36–44, 2000.

B

Application-specific hints in reconfigurable Grid scheduling algorithms

B. Volckaert, P. Thysebaert, F. De Turck, B. Dhoedt, P. Demeester

published in Lecture Notes in Computer Science (LNCS 3038), Computational Science, Proceedings P3 of ICCS 2004, Springer-Verlag Berlin Heidelberg 2004, Krakow, 2004, Vol. LNCS 3038, pp. 149-157.

Abstract *In this paper, we investigate the use of application-specific hints when scheduling jobs on a Computational Grid, as these jobs can expose widely differing characteristics regarding CPU and I/O requirements. Specifically, we consider hints that specify the relative importance of network and computational resources w.r.t. their influence on the associated application's performance. Using our ns-2 based Grid Simulator (NSGrid), we compare schedules that were produced by taking application-specific hints into account to schedules produced by applying the same strategy for all jobs. The results show that better schedules can be obtained when using these scheduling hints intelligently.*

B.1 Introduction

Computational Grids consist of a multitude of heterogeneous resources (such as Computational, Storage and Network resources) which can be co-allocated for the execution of applications or jobs. The allocation of resources to particular jobs and the order in which these jobs are processed on the Grid are determined by the Grid's management infrastructure through the application of a *scheduling algorithm*. Most jobs will need access to different resource types during their execution, meaning job execution progress depends on the quality of service delivered to that job by *every* resource involved. The exact sensitivity of a job's computational progress w.r.t. the individual resources' performance depends on the "nature" of that job: jobs that require huge amounts of CPU power, but perform (relatively) little I/O operations, will only suffer lightly from a temporary degradation of e.g. available network bandwidth, but cannot withstand a sudden loss of CPU power. Conversely, the computational progress made by an I/O-bound job is influenced dramatically by the network bandwidth available to that job, and to a lesser extent by the variation in available computing power. This leads us to the observation that:

1. algorithms that schedule jobs on a Computational Grid ought to take into account the status of multiple different resource types instead of solely relying on e.g. the available computational power.

2. using the same scheduling algorithm with rigid constraints for all job types can be outperformed by applying different scheduling algorithms for different job types; each job-specific algorithm only performs rigid resource reservation with *critical* resources, but allows for relaxed resource availability constraints when dealing with non-critical resources.

This indicates that programmable architectures, where the job scheduling mechanism is provided (at least partly) by the application (and where the algorithms could even be adapted on the fly), are a promising avenue towards grids offering a wide variety of services (each having their specific service metrics and quality classes). In this approach, the grid infrastructure is coarsely managed by cross-service components, supplemented by on-the-fly configurable service specific management components. The latter components manage the resources allocated to the service on a fine grained level, optimizing job throughput and service quality simultaneously according to service specific attributes and metrics. In this paper we show how *scheduling hints* can be incorporated into job descriptions. The goal of these hints is to enable a Grid scheduler to estimate the critical level of the different resource types w.r.t. that job. Because hints are contained in the job description, they are available at each scheduler in the Grid to which the job is submitted or forwarded.

This paper continues as follows: section B.2 starts with a short description of the related work. In section B.3, we give an overview of the relevant simulation models used: the Grid, Resource, VPN, Job and Scheduling Hint models are explained in detail. In section B.4, we discuss the various algorithms that we compared; they differ from each other in (i) the types of resources they take into account and (ii) whether or not they treat all jobs equally. Our simulated scenario and corresponding results are presented in section B.5, leading to the conclusions in section B.6.

B.2 Related work

Well-known Grid Simulation toolkits include *GridSim* [1] and *SimGrid* [2]. The key difference with *NSGrid* [3] is that NSGrid makes use of a network simulator (ns-2 [4]) which allows for accurate simulation down to the network packet level.

Scheduling jobs over multiple processing units has been studied extensively in literature. Machine scheduling [5] [6] is concerned with producing optimal schedules for tasks on a set of tightly-coupled processors, and provides analytical results for certain objective functions. Jobs are commonly modelled as task graphs, or as continuously divisible work entities. As these models do not deal with "network connections" or "data transfers", they do not capture all the Grid-specific ingredients described in the previous section. Grid scheduling strategies which take both computational resource load and data locality into account are extensively discussed in [7]. The use of Application-specific scheduling hints is not considered however.

The *Metacomputing Adaptive Runtime System* (MARS) [8] is a framework for utilizing a heterogeneous WAN-connected metacomputer as a distributed computing platform. When scheduling tasks, the MARS system takes into account Computational Resource and Network load, and statistical performance data gathered from previous runs of the tasks. As such, the MARS approach differs from the scheduling model simulated by NSGrid, as NSGrid allows for *user preferences* to be taken into account as well.

Application-level scheduling agents, interoperable with existing resource management systems have been implemented in the *AppLeS* [9] work. Essentially, one separate scheduler needs to be constructed per application type. Our simulation environment allows the simulation of multiple scheduling scenarios, including those using a single centralized schedule as well as those having multiple competing schedulers (not necessarily one per application type).

B.3 Simulation model

B.3.1 Grid model

Grids are modelled as a collection of interconnected and geographically dispersed *Grid sites*. Each Grid Site can contain multiple *resources* of different kinds such as Computational Resources (CRs) and Storage Resources (SRs) interconnected by VPN links. At each Grid Site, resource properties and status information are collected in a local *Information Service*. Jobs are submitted through a *Grid Portal* and are scheduled on some collection of resources by a *Scheduler*. To this end, the scheduler makes *reservations* with the appropriate *Resource Managers*.

B.3.2 Grid resource models

Each Grid site can offer one or more CRs and/or SRs. A CR is a monolithic entity, described by its total processing power in MIPS, the maximum number of jobs that it can handle simultaneously and the maximum slice of processing power (in MIPS) that can be reserved for a single job. An SR on the other hand, serves the purpose of providing disk space to store input and output data. In our model, their basic properties include the total available storage space, the input data sets currently stored at the resource and the speed at which the resource can read and write data. While a SR does not perform computational work, it can be attached to the same network node as some CR. Interconnections between local resources are modelled as a collection of point-to-point VPN links, each offering a guaranteed total bandwidth available to Grid jobs. Of course, these VPN links can only be set up if, in the underlying network, a route (with sufficient bandwidth capacity) exists between the nodes to which these resources are attached. Different Grid Sites can be interconnected by a VPN link. These models are covered in more detail in [3].

B.3.3 Job model

The atomic (i.e. that which cannot be parallelized) unit of work used throughout this paper is coined with the term *job*. Each job is characterized by its length (measured in *instructions*), its required *input data sets*, its need for *storage*, and the *burstiness* with which these data streams are read or written. During a job's execution, a certain minimal computational progress is to be guaranteed at all times (i.e. a deadline relative to the starting time is to be met).

Knowing the job's total length (in million instructions, MI) and the frequency at which each input (output) stream is read (written), the total execution length of a job can be seen as a concatenation of instruction "blocks". The block of input data to be processed in such an instruction block is to be present before the start of the instruction block; that data is therefore transferred from the input source at the

start of the previous instruction block. In a similar way, the output data produced by each instruction block is sent out at the beginning of the next instruction block. We assume these input and output transfers occur in parallel with the execution of an instruction block. Only when input data is not available at the beginning of an instruction block or previous output data has not been completely transferred yet, a job is suspended until the blocking operation completes. The presented model allows us to mimic both *streaming* data (high read or write frequency) and *data staging* approaches (read frequency set to 1). A typical job execution cycle (one input stream and one output stream) is shown in figure B.1.

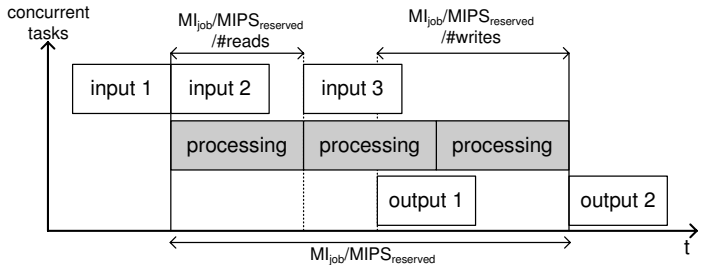


Figure B.1: Simulated job lifespan with indication of start-of-I/O events; non-blocking job

B.3.4 Scheduling hints model

From the job model described in the previous section, it is clear that the computational progress made by a job is determined by both the computational power and network bandwidth available to that job. As such, scheduling hints (distributed together with the job description) describe

- the resource types that are to be taken into account when scheduling this job; any subset of `{Computational / Network Resource}` can be specified.

- for each of the above resource types, the size of an acceptable (not preventing the job from being scheduled on that resource) deviation from the resource's performance delivered to that job (described in the job requirements).

It is not desirable to have critical resources deliver a less-than-minimal performance to the job, while this may not matter much for non-critical resources.

B.4 Algorithms

When jobs are submitted to a Grid Portal, a Scheduler needs to decide where to place the job for execution. As has been mentioned previously, we discriminate between algorithms using two criteria: the type of resources they take into account and whether or not they take into account scheduling hints. If the scheduler is unable to allocate the needed resources for a job, the job gets queued for rescheduling in the next scheduling round. The time between two scheduling rounds can be fixed, but it is also possible to set a threshold which triggers the next scheduling round. During each scheduling round, every algorithm processes submitted yet unscheduled jobs in a greedy fashion, attempting to minimize job completion time. Once scheduled, our scheduler does not pre-empt jobs.

B.4.1 Algorithm "NoNetwork"

As the name implies, this algorithm does not take into account the status of Network Resources when scheduling jobs. Rather, it assumes that only CRs are critical for each job. Furthermore, it will treat minimal job requirements as hard constraints; it disregards hints that might propose a softer approach. At first, "NoNetwork" will attempt to place a job on a site's local CRs, only using remote resources when strictly necessary (we believe this to be a plausible approach from an economic viewpoint). If this is impossible, and at least one remote CR is available, that job will be scheduled on the remote CR offering most processing power. It is therefore expected that this algorithm will perform badly when dealing with a significant amount of "I/O-bound" jobs: due to "blocking", these jobs will finish considerably later than predicted by the scheduling algorithm.

B.4.2 Algorithm "PreferLocal"

Similar to the "NoNetwork" algorithm, "PreferLocal" will a priori attempt to place a job on a site's local CRs. If this turns out to be impossible, remote CRs will be considered. While looking for the best resources for a particular job, however, "PreferLocal" not only considers the status of CRs, but also the residual bandwidth on network links connecting Computational and Storage Resources. The best resource combination is the one that maximizes the job's computational progress. For a job requiring one CR and one SR (in different Grid Sites, connected through a VPN link), the maximal computational progress (expressed in MIPS) that can be delivered to that job is given by

$$MIPS_{\mathit{eff}} = \min_{CR,VPN}(MIPS_{CR}, \frac{MI * BW_{VPN}}{8 * DATASIZE})$$

It is easily verified that it makes no sense to allocate a bigger portion of the best CR's power to this job, as due to network limitations, the job cannot be processed

at a higher rate. In a similar way, due to CR limitations, it makes no sense to allocate more bandwidth to the job than $\frac{8*DATASIZE*MIPS_{eff}}{MI}$. Like "NoNetwork", "PreferLocal" does not adapt its strategy using job-specific hints.

B.4.3 Algorithm "Service"

Algorithm "Service", like "PreferLocal", considers both Computational and Network Resources when scheduling jobs. However, instead of immediately rejecting those resource combinations that would not allow some of the job's minimal requirements to be met, it can still select those resources (if none better are found) if this is declared "acceptable" by the appropriate job hint. For instance, jobs can specify that their available network bandwidth requirements are less important than their computational requirements (and/or quantify the relative importance), or that there is no gain in finishing the job before its deadline (i.e. no gain in attempting to maximize that job's $MIPS_{eff}$). Using these hints, jobs can be divided into different classes, where all jobs in one class have similar associated hints. The algorithm can then be seen as delivering the same service to each of those jobs in a single class.

B.5 Simulation results

B.5.1 Simulated Grid

A fixed Grid topology was used for all simulations presented here (see figure B.2). This topology is depicted in table B.1. Grid control components are interconnected by means of dedicated network links providing for out-of-band Grid control traffic (as shown by the dotted network links).

B.5.2 Simulated jobs

In our simulations, two types of hints were used (i.e. two service types). The first type of hint is supplied with CPU-bound jobs, and specifies that the job should be scheduled on the fastest CR, even if this means scheduling the job on a remote resource when it could have been scheduled locally. The second type of hint is distributed with I/O-bound jobs, stating that these jobs are better off being scheduled using only local resources, as this offers better chances of allocating sufficient network bandwidth. In both cases, however, resource loads are not ignored; rather, the *preferred* execution rate for a job is no longer treated as a rigid minimum.

We have compared the "Service" algorithm (which understands and uses the hints as specified) for this job set with the "NoNetwork" and "PreferLocal" algorithms, both disregarding the hints.

Service type	I/O size	MI
I/O-bound	6100 MB	12500000
CPU-bound	0.4 MB	25000000

Table B.1: Simulated job classes

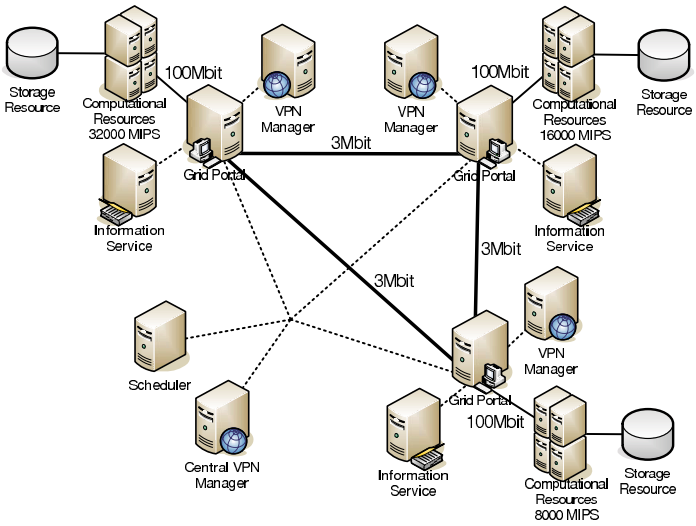


Figure B.2: Simulated topology

B.5.3 Per-class response times

The average job response time for each algorithm is shown in figure B.3. The figure shows both the overall response rate and the per-class response rates when 20% I/O jobs and 80% CPU intensive jobs are submitted to the Grid. As expected, "NoNetwork" fails to produce good schedules for I/O-bound jobs, as it ignores network loads (which are, of course, of particular importance to this type of job). In addition, notice that the use of hints (algorithm "Service") improves the average response time for CPU-bound jobs (which make up most of the jobs in this simulation). Indeed, some jobs are now processed at a rate (slightly) lower than the preferred one (the goal of the hints is exactly to specify that this is allowed), but finish sooner than if they were delayed in time.

B.5.4 Response times with varying class representation

In these simulations we stressed the three Grid sites by submitting a heavy job load (parameterized by the percentage of I/O-bound jobs in the total job load). The resulting average job response time for the three algorithms (as a function of

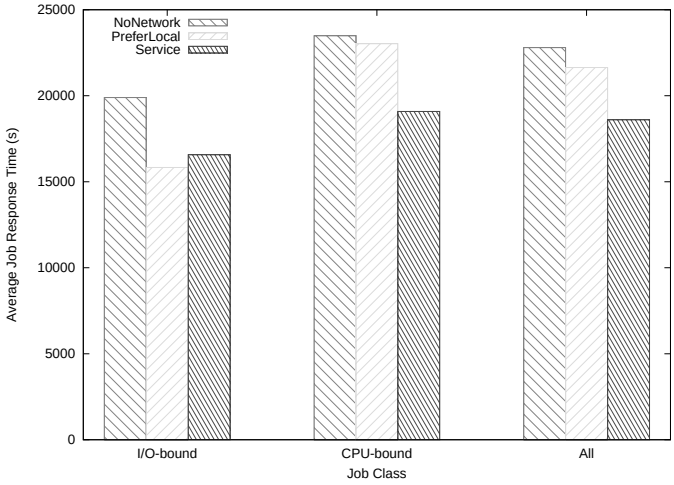


Figure B.3: Average job response time

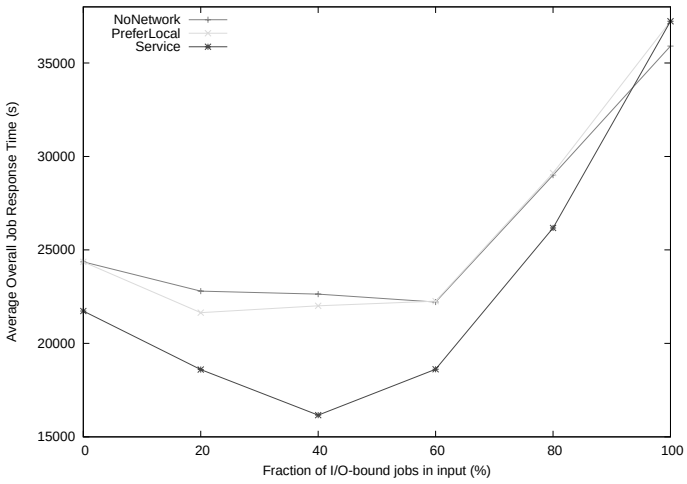


Figure B.4: NoNetwork, PreferLocal and Service schedule performance

the percentage of I/O-bound jobs) is shown in figure B.4.

When only CPU-bound jobs are submitted, “NoNetwork” performs like “PreferLocal”, as we remarked previously. Note that “Service” performs slightly better, as this algorithm does not prefer local resources over remote. When the amount of I/O-bound jobs is increased, “NoNetwork” performance degrades as network load status gains importance when scheduling jobs. Since “PreferLocal” always tries local resources first (instead of the best resources), it will schedule more I/O-bound

jobs remotely (i.e. using lower-bandwidth links) as CPU-bound jobs (the majority) use up these local resources; this accounts for the difference with "Service". Once the fraction of I/O-bound jobs passes 60%, the network links of our simulated Grid saturate, and the three algorithms' performance converges.

B.6 Conclusions

In this paper we have shown the benefits of using active scheduling mechanisms in a service oriented Grid environment. In particular, we used NSGrid to compare the efficiency of schedules produced by algorithms that do not take into account the service specific needs of a job, to schedules produced by algorithms that use service scheduling hints. It was shown that when using the latter algorithms, average job response times in our simulated scenario improved significantly (up to 30% in some cases).

References

[1] R. Buyya and M. Murshed. *GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing*. The Journal of Concurrency and Computation: Practice and Experience (CCPE), May 2002.

[2] Arnaud Legrand, Loris Marchal, and Henri Casanova. *Scheduling Distributed Applications: the SimGrid Simulation Framework.* In CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid, 2003.

[3] B. Volckaert, P. Thysebaert, F. De Turck, P. Demeester, and B. Dhoedt. *Evaluation of grid scheduling strategies through a network-aware grid simulator*. In published in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA'03, 2003.

[4] *The Network Simulator - NS2*. `http://www.isi.edu/nsnam/ns`.

[5] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. *Theory and Practice in Parallel Job Scheduling*. In Dror G. Feitelson and Larry Rudolph, editors, Job Scheduling Strategies for Parallel Processing, pages 1–34. Springer Verlag, 1997.

[6] L. Hall, A. Schulz, D. Shmoys, and J. Wein. *Scheduling To Minimize Average Completion Time: Off-line and On-line Algorithms*. In SODA: ACM-SIAM Symposium on Discrete Algorithms (Conference on Theoretical and Experimental Analysis of Discrete Algorithms), 1996.

[7] I. Foster K. Ranganathan. *Simulation Studies of Computation and Data Scheduling Algorithms for Data Grids*. Journal of Grid Computing, 1:53–62, 2003.

[8] J. Gehring and A. Reinfeld. *Mars - a framework for minimizing the job execution time in a metacomputing environment*. In Proceedings of Future General Computer Systems '96, 1996.

[9] F. Berman et al. *Adaptive Computing on the Grid Using AppLeS*. IEEE Transactions on Parallel and Distributed Systems, 14:369–382, 2003.

C

Network Aware Scheduling in Grids

B. Volckaert, P. Thysebaert, M. De Leenheer, F. De Turck, B. Dhoedt, P. Demeester

published in Proceedings of NOC2004, 9th European Conference on Networks & Optical Communications, TU Eindhoven, 2004, pp. 311-318.

Abstract *Computational Grids consist of an aggregation of data and computing resources, which can be co-allocated to execute large data-intensive tasks which cannot be handled by an individual resource. The vast amounts of (possibly remote) data these tasks process give rise to the need for a high-capacity network interconnecting the various resources. In this paper, it is argued that, in order to obtain quality job schedules resulting in low throughput times while making efficient use of available resources, network- and service-aware algorithms need to be used. We present several heuristic algorithms, which we simulate extensively in order to gain insight in the quality of the generated schedules. In particular, in a Grid with heterogeneous Computational Resources and link capacities, we show how the average job response time can be improved by distinguishing between data-intensive and compute-intensive jobs, and scheduling these jobs based upon both Computational Resource and network load. We show that the heuristics presented perform well when compared to an Integer Linear Programming approach modelling a periodic on-line scheduler.*

C.1 Introduction

In parallel and distributed computing communities, a lot of attention has recently been paid to Computational Grids. These Grids are conceived as a collection of resources of various types (computational, storage, etc.), distributed over multiple geographical locations. The underlying principle of an operational Grid is that several resources can be co-allocated to the same resource-intensive task, overcoming each resource's individual limitations. These resource-intensive tasks will typically operate on large amounts of data, which must be transferred from its storage site to the processing site. This implies that, when scheduling jobs on this Grid, gathering information on data location and network bandwidth availability is essential in order to generate quality schedules.

Scheduling algorithms for parallel computing systems have been studied extensively in literature [1]. Most approaches assume a discrete workload (e.g. jobs) and use a stochastic [2, 3] or linear programming model [4]. Divisible load theory [5], on the other hand, assumes a continuously divisible load, and seems well suited to model a Grid, as it leads to fairly easy systems of equations, even for large problem sizes. However, in order to remain analytically tractable, these techniques trade in some of the characteristic properties of a Grid by simplifying network and/or job models or focusing on a particular topology.

More accurate grid models require the use of simulation tools in order to be evaluated. In [6] and [7], network-awareness is introduced in the scheduling decisions by replicating data to suitable sites; jobs can then be scheduled to run at a site holding all the necessary data. Our in-house developed NS2-based Grid simulator NSGrid [8, 9] extends this model by allowing data to be transferred to the execution site while the job has already been started, and this extension is used in the simulations described here.

In this paper, we use this simulator to investigate the generation of schedules for a Grid in which the core network uses high capacity (possibly optical) core links. While this approach seemingly provides for abundant bandwidth in the network, it follows from the expected Grid job size in the near future (the expected data size generated yearly in the European DataGrid project is in the order of 12-14 PB/year [10]) that high-bandwidth (typically found in optical networks) interconnections are a necessity, not a luxury. This implies that naive scheduling heuristics, which assume "infinite" (or at least "sufficient") network bandwidth at all times or ignore the network altogether are unable to generate quality schedules, both in terms of average job response time and in terms of efficient resource utilization.

We verify this through simulation of different scheduling heuristics and calculation of an ILP solution to the problem, operating on a suitable workload (consisting of a mix of different types of Grid jobs requiring sufficiently high computational power and network bandwidth).

The remaining sections of this paper are organized as follows: Section C.2 explains how the various Grid components in our simulation environment are modelled. A concise description of the scheduling problem addressed here, the ILP model of a periodic, on-line scheduler and the heuristics used in our simulations are presented in section C.3. Our results are presented in section C.4, and the main conclusions drawn from them are summarized in section C.5.

C.2 Grid model

For the simulations described in this paper, a Grid has been modelled as a set of interconnected, geographically dispersed *Grid sites*. At each site, one or more resources are present. The status and properties of these resources are stored in an *Information Service* repository. The idea is that a single site groups "geographically close" resources, connecting them through a high-bandwidth, low-latency network. Each site is connected to the backbone network and all simulations use static shortest-path routing for inter-site traffic.

C.2.1 Grid resources

The resources modelled in the simulation framework include Computational Resources (CR) and Storage/Information Resources (SR/IR). Computational Resources represent either a cluster or multi-computer, and are described by the number of (time-shared) processors provided, the amount of memory and temporary disk space. In order to schedule a job, a reservation can be made with a CR, consisting of a time-share of a processor dedicated to that job. Storage/Information Resources hold the data written/read by the job. We use the notation $\mathcal{C}$ for the set of Computational Resources, with elements $c \in \mathcal{C}$.

Jobs transfer data from and to remote resources using *connections*, which act as point-to-point bandwidth pipes between a pair of resources and are allocated with the network management system by the grid scheduler. This allows the network to be viewed as a resource (with limited renewable capacity a.k.a. bandwidth, of which a portion can be reserved for a connection), giving the scheduler the possibility to generate accurate network-aware schedules, and balance computation vs. communication bottlenecks. We use the notation $\mathcal{E}$ for the collection of network links, where each $e \in \mathcal{E}$ has a bandwidth B_e.

C.2.2 Jobs

We reserve the term *job* for atomic units of work. Jobs $j \in \mathcal{J}$ are characterized by their computational and communication (I/O) complexity. The first property is captured using the *reference running time* t_j (time to execute on a reference processor when not experiencing any communication bottlenecks). The communication

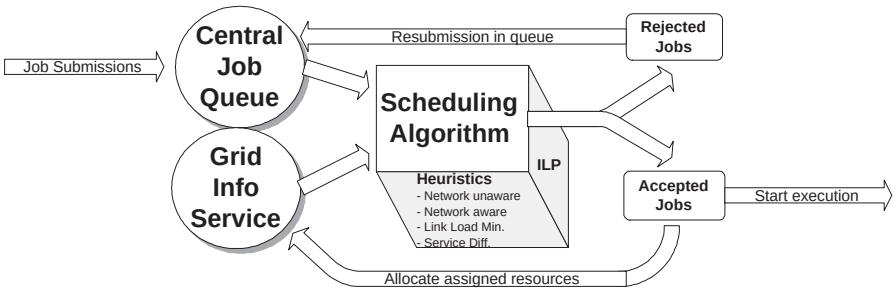


Figure C.1: General overview of scheduling algorithm

complexity is specified by a number of parameters: the data sets the job needs as input (total size d_j^r), the size of the outputs it produces (total size d_j^w), whether or not I/O are processed block-per-block (the opposite case is when all input data is pre-staged to the CR the job is running on *prior* to execution) and the amount of separate blocks I/O data is divided into (if applicable). This allows to simulate data *streams* with arbitrary precision. The fundamental relation between computation and communication in our model is that a data block can only be processed when it has arrived entirely; if this is not the case, the job's computational progress is *blocked*.

C.3 Scheduling algorithms

The scheduling algorithms we use are *queueing* algorithms (see figure C.1) that attempt to schedule jobs in a FCFS fashion [11], and, once a job is scheduled, the scheduler does not attempt to pre-empt jobs. Since this introduces the risk of allocating minimal leftover computational resource time shares to certain jobs, we have demanded that a processor can be time-shared by at most a specified number of concurrent jobs N_c. Our scheduling algorithms are executed periodically; after a specified amount of time T, all job requests are taken into consideration, and, based on the state of the grid's resources at that time, a schedule is calculated.

C.3.1 ILP

In this section we present an Integer Linear Programming (ILP) model, which captures all important parameters in the scheduling problem. It is well known that ILP is a computationally intensive technique. This, together with the fact that the proposed grid and job models are both very generic, obliges us to impose additional restrictions to the model. We model a grid as a collection $\mathcal{C}$ of resources, where each $c \in \mathcal{C}$ has a processing capacity P_c and unlimited storage capacity. We

assume a job can only be executed at its reference processing speed, and thus the execution length is given by the reference running time t_j. This also implies the job is not held up by communication bottlenecks. We further assume a job has only one input and one output block, which is to be transferred at a constant datarate during the job's execution. It follows that a job requires $b_j^r = \frac{d_j^r}{t_j}$ of input bandwidth, and $b_j^w = \frac{d_j^w}{t_j}$ of output bandwidth. Two additional parameters for the ILP model are the budget w_j which a job pays if it is accepted for execution, and the arrival site s_j. We introduce the following binary variables:

- $x_j = 1 \Leftarrow$: job j is accepted for execution.
- $y_{jc}^p = 1 \Leftarrow$: job j uses processing power of resource c.
  $y_{jc}^s = 1 \Leftarrow$: job j uses storage capacity of resource c.
- $z_{je}^r = 1 \Leftarrow$: job j uses network link e for input.
  $z_{je}^w = 1 \Leftarrow$: job j uses network link e for output.

These variables allow us to express the cost to execute a job, which is given by a weighted sum of the processing cost and the transfer cost:

$$C_j = \alpha \times k_j \times \sum_{c \in \mathcal{C}} (C_c^p \times p_j \times y_{jc}^p) + \beta \times k_j \times \sum_{e \in \mathcal{E}} (C_e^b \times (b_j^r \times z_{je}^r + b_j^w \times z_{je}^w)) \quad \text{(C.1)}$$

where $k_j = \lceil \frac{t_j}{T} \rceil$ is the number of periods a task will run, and C_c^p and C_e^b are the costs for using resource c and network link e during one time unit. Parameters α and β allow us to give priority to CPU-bound or network-bound jobs. We investigate their influence on the scheduler in section C.4.3.2.

Our objective function, stated below, expresses the *profit* our scheduler makes by choosing which tasks to execute where. Clearly our goal is to maximize this function, as it strikes a balance between a user's perceived satisfaction (high acceptance rate) and the provider's optimal resource utilization (keeping the costs low means jobs are executed close to where they are submitted).

$$f = \sum_{j \in \mathcal{J}} (w_j \times x_j - C_j) \quad \text{(C.2)}$$

We now introduce the constraints which need to be satisfied. Recall that all variables are binary-valued. We first state that an accepted job uses computational power of exactly one resource, uses its submission site as storage site, and possibly uses multiple network links:

$$\forall j \cdot \sum_{c \in \mathcal{C}} y_{jc}^p = x_j \quad ; \quad \forall j \cdot y_{jc}^s = \begin{cases} x_j & \text{if } c = s_j \\ 0 & \text{otherwise} \end{cases} \quad ; \quad \forall j \cdot \forall e \cdot z_{je}^{r,w} \leq x_j \quad \text{(C.3)}$$

The following constraints limit the usage of the individual resources:

$$\forall c \cdot \sum_{j \in \mathcal{J}} y^p_{jc} \leq N_c \quad ; \quad \forall c \cdot \sum_{j \in \mathcal{J}} (p_j \times y^p_{jc}) \leq P_c \quad ; \quad \forall e \cdot \sum_{j \in \mathcal{J}} (b^r_j \times z^r_{je} + b^w_j \times z^w_{je}) \leq B_e \tag{C.4}$$

The model is completed by expressing the balance of flows in the network (analogous constraints are necessary for variables z^w_{je}):

$$\forall j \cdot \forall u \cdot \sum_{\substack{v \in \mathcal{V} \\ (u,v) \in \mathcal{E}}} z^r_{j(u,v)} - \sum_{\substack{v \in \mathcal{V} \\ (u,v) \in \mathcal{E}}} z^r_{j(v,u)} = y^s_{ju} - y^p_{ju} \tag{C.5}$$

C.3.2 Heuristics

The ILP model described above can only be used to solve an offline scheduling problem because of its computational intensiveness (i.e. it is unsuitable as an actual scheduling algorithm implementation). Below we give an overview of the scheduling heuristics that were used in our study and compare their results with the ILP solution.

C.3.2.1 Network Unaware

We have used network unaware scheduling as a naive heuristic for comparison. This heuristic will compute job schedules based on CR/IR/SR status. It does not take into account information concerning the status of resource interconnections. Because of this, job processing can block on I/O operations: computational progress is no longer determined by the CR's processor fraction that has been allocated to a job (which, together with the job's length and the CR's relative speed determines its earliest end time *if all input and output transfers complete on time i.e. before the start of the appropriate instruction block*), but rather by the limited bandwidth available to its I/O streams.

C.3.2.2 Network Aware

Network aware algorithms will not only query the Information Services (for resources adhering to the job's requirements), but also the network management for information about the status of the interconnecting network links. Based on both the Information Services' and Connection Manager's answer, the heuristic is able to calculate job execution speed and end time more accurately, taking into account the speed at which data can be delivered to (or sent from) each available CR. For jobs with one input and one output stream, the best resource (CR/IR/SR) triplet is the one that minimizes the expected completion time of the job. This value is determined by the available processing power to that job on the Computational

Resource, the job's length, the job's total I/O size and the residual bandwidth on the links from IR to CR and from CR to SR.

C.3.2.3 Minimum Hop Count

This heuristic attempts to minimize network usage when scheduling jobs. In order to achieve this, the scheduler evaluates the network load each CR/IR/SR triplet selection would generate for a given job, and chooses to schedule the job on the triplet that minimizes this network usage. Typically, when jobs require that their I/O be streamed from/to the originating site, this heuristic will first attempt to schedule the job on local resources, and, when this is infeasible, will try to submit the job to a CR *as close as possible to the job's originating site*, minimizing the amount of network links that data has to be sent over.

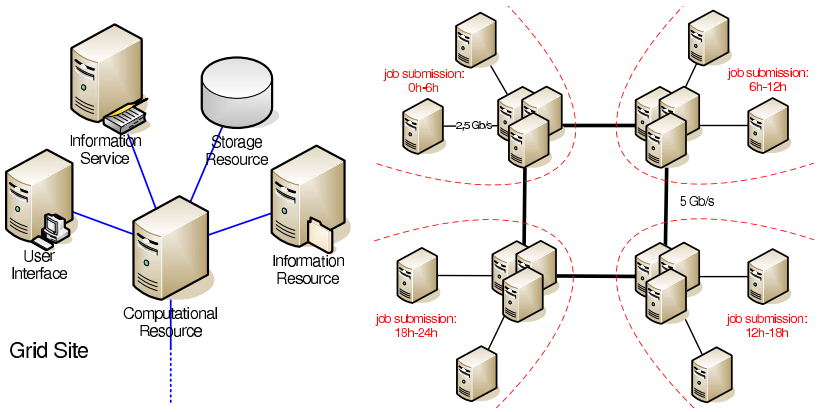
C.3.2.4 Service differentiation

The Service Differentiation heuristic will compute the data intensity of a job, based on the job's I/O and processing requirements, and, based on this metric, will classify the job as either a data intensive or a computational intensive job. Data intensive jobs will only be scheduled on resources local to the job's originating site (if these resources adhere to the job's requirements). If no local processing slots are available, the data intensive jobs are queued for local processing. Jobs in the other service class will be scheduled on remote processing resources (in a network aware manner), in order to maximize the chance of having local processing slots available for data intensive jobs.

C.4 Simulation results

C.4.1 Simulated topology

The Grid topology used in our simulations is shown in figure C.2. The topology is symmetric and consists of 12 Grid sites (8 edge and 4 core sites) interconnected by means of bidirectional optical links ($2,5Gb/s$ links between edge and core sites, and a $5Gb/s$ ring network between core sites). Furthermore, each core site connects two edge sites to the core ring network. Each site contains an IR/SR/CR and an Information Service management component responsible for tracking resource properties. Jobs are submitted to the site through that site's user interface. Core sites differ from edge sites in terms of the offered Computational Resource; both CR types have 4 processors, capable of running 8 jobs simultaneously, but a core CR offers three times the processing speed of its edge counterpart. A site's local interconnections are modelled as fiber channel links capable of transmitting data at $10Gb/s$.



(a) Grid site (b) Grid topology

Figure C.2: Simulated topology

C.4.2 Grid jobs

The user interface at each Grid site submits two different job types: data-intensive and cpu-intensive jobs. Both job types have the same reference run time, but they differ in the amount of I/O data they need/generate. Jobs are instantiated at specified time intervals, according to the interarrival time (IAT) distribution of the job's class. Furthermore, all jobs *stream* their I/O from/to the originating site's local IR/SR. Average job parameters have been summarized in table C.1. In each simulation, the job load consists of 405 jobs. We chose to use a fixed interval of $1000s$ between consecutive scheduling rounds.

	CPU-Job	**Data-Job**
Input(GB)	15.6	156
Output(GB)	15.6	156
IAT(s)	1350	5400
Ref. run time(s)	10000	10000

Table C.1: Job properties

C.4.3 Results

C.4.3.1 Bandwidth of core network

The job throughput for different core network bandwidth values is shown in figure C.3. We clearly see that, for the given job load, a "sufficient" bandwidth exists; going above this value has little to no effect on the job throughput and thus implies an overdimensioned core network.

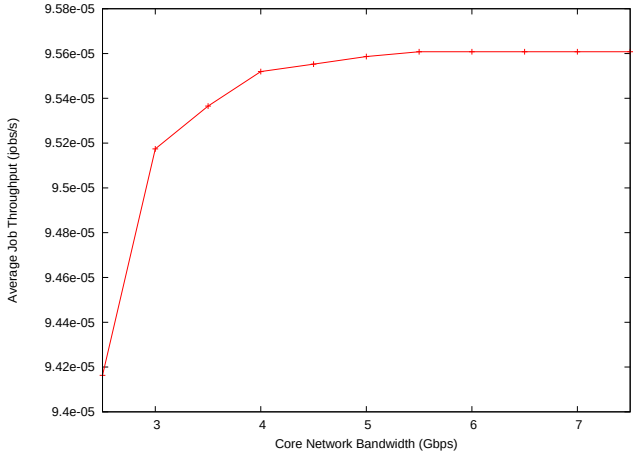


Figure C.3: ILP job throughput

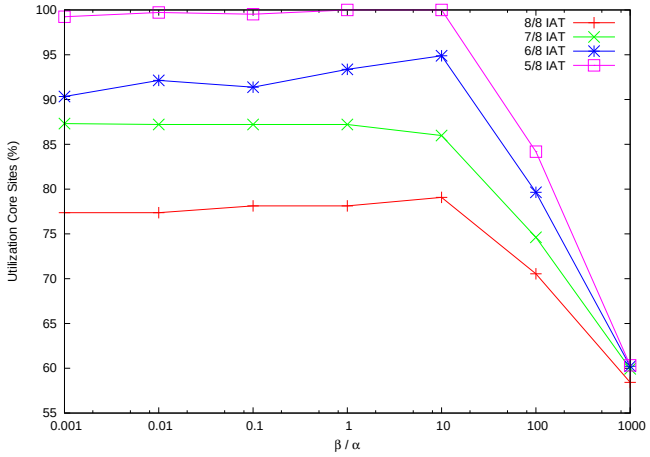


Figure C.4: ILP utilization of core sites

C.4.3.2 Parameters of ILP

Figure C.4 depicts the utilization of the core sites for various values of the quotient $\frac{\beta}{\alpha}$. This quotient expresses the relative cost between the usage of a network link and a computational resource. Clearly, higher values imply that using the network becomes more and more expensive, and thus local execution becomes preferable. We performed this experiment for different values of the job interarrival times (IAT); higher job arrival rates generate higher site utilization as long as the network is cheap enough. When the usage of the network becomes too expensive, the site utilization drops notably.

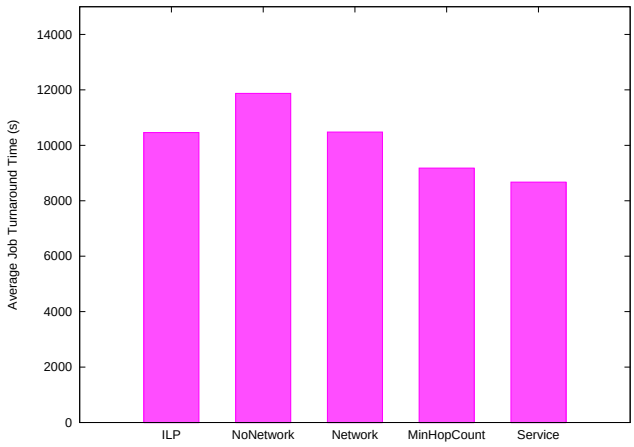


Figure C.5: Average job turnaround times

C.4.3.3 Turnaround time

Figure C.5 shows the average job turnaround time (time between arrival of a job and the time it finishes) for the different scheduling heuristics discussed above. The "Service" heuristic clearly provides the best performance, by pushing cpu-intensive jobs to remote processing sites, and, in doing so, reserving the local processing slots for data-intensive jobs. It is exactly because remote processing a lot of data-intensive jobs causes a network bottleneck (and thereby causes processing to be blocked), that the network aware "Minimum Hop Count" and "Service" heuristics perform well.

C.4.3.4 Network utilization

The difference in network utilization resulting from deploying different scheduling heuristics is shown in figure C.6 (the lower part of each bar represents network utilization of data-intensive jobs). The metric used here is called weighted hopcount, as we took into account the fact that data-intensive jobs use on average 10 times more data than other jobs. The results show that the least network traffic flows when we make use of the service differentiation heuristic, maximizing the chance that data-intensive jobs obtain a local processing slot.

C.5 Conclusions

Our main observations are that due to the expected size of the data transferred by a Grid job in the future, even with high-capacity network links, naive (non-network

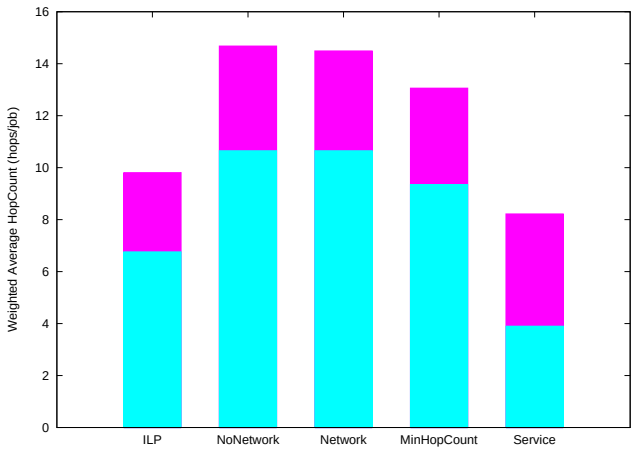


Figure C.6: Weighted average hopcount

aware) heuristics are outperformed by network-aware scheduling strategies both from the end user's viewpoint (job turnaround time) and from the provider's viewpoint (network usage efficiency). Furthermore, if multiple applications with different bandwidth requirements are run on the Grid, further improvements are possible by deploying different service-oriented scheduling strategies.

References

[1] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. *Theory and Practice in Parallel Job Scheduling*. In Dror G. Feitelson and Larry Rudolph, editors, Job Scheduling Strategies for Parallel Processing, pages 1–34. Springer Verlag, 1997.

[2] A.I.D. Bucur and D.H.J. Epema. *An Evaluation of Processor Co-Allocation for Different System Configurations and Job Struxctures*. In Proceedings of SBAC-PAD, 2002.

[3] A.I.D. Bucur and D.H.J. Epema. *The Influence of the Structure and Sizes of Jobs on the Performance of Co-Allocation*. In Proceedings of JSSPP6, 2000.

[4] L. Hall, A. Schulz, D. Shmoys, and J. Wein. *Scheduling To Minimize Average Completion Time: Off-line and On-line Algorithms*. In SODA: ACM-SIAM Symposium on Discrete Algorithms (Conference on Theoretical and Experimental Analysis of Discrete Algorithms), 1996.

[5] D. Yu and T.G. Robertazzi. *Divisible Load Scheduling for Grid Computing*. In Proceedings of the IASTED 2003 International Conference on Parallel and Distributed Computing and Systems (PDCS), 2003.

[6] I. Foster K. Ranganathan. *Simulation Studies of Computation and Data Scheduling Algorithms for Data Grids*. Journal of Grid Computing, 1:53–62, 2003.

[7] David G. Cameron, Ruben Carvajal-Schiaffino, A. Paul Millar, Caitriana Nicholson, Kurt Stockinger, and Floriano Zini. *Evaluating Scheduling and Replica Optimisation Strategies in OptorSim*. In 4th International Workshop on Grid Computing (Grid2003), 2003.

[8] *The Network Simulator - NS2*. `http://www.isi.edu/nsnam/ns`.

[9] B. Volckaert, P. Thysebaert, F. De Turck, P. Demeester, and B. Dhoedt. *Evaluation of grid scheduling strategies through a network-aware grid simulator*. In published in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA'03, 2003.

[10] *The DataGrid Project*. `http://eu-datagrid.web.cern.ch/eu-datagrid/`.

[11] M. Hovestadt, O. Kao, A. Keller, and A. Streit. *Scheduling in HPC Resource Management Systems: Queueing vs. Planning*. In Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing, 2003.