IMPORTANCE-AWARE MONITORING FOR LARGE-SCALE GRID INFORMATION SERVICES

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2007

By
Serafeim Zanikolas
School of Computer Science

Contents

List of Tables

List of Figures

Abstract

Grid computing aims to realise a software and hardware infrastructure for controlled sharing of networked resources within and across organisations. This thesis studies the scalability of large-scale grid monitoring and information services, which are mainly used for the discovery of resources of interest.

This work claims that large-scale grid monitoring systems have to balance between three competing performance metrics (query response time, imposed network overhead, and information freshness). Improving one of the three metrics will affect another; any solution will be based on a trade-off. The thesis is motivated by the observation that existing grid monitoring systems can only be manually configured for a trade-off among the three metrics that applies equally to all monitored resources; this implies that all resources in a grid are considered to be of equal importance (i.e., usefulness to users). Assuming that in a large heterogeneous grid this is unlikely to hold, the thesis proposes an importance-based monitoring architecture for large-scale grid information services, based on an adaptation of the web crawling paradigm. The thesis is that, since not all resources are of equal importance, one can vary the trade-off based on the importance of the monitored resources. This implies an increased frequency of updates for the sites that host the most important resources, while lower freshness is maintained for those hosting the least important resources. The proposed architecture is complemented by a query-independent definition of resource importance.

The proposed architecture is evaluated based on large-scale deployments of a prototype implementation in Planetlab, and compared against analytically derived performance results of typical grid monitoring architectures, to investigate which approach is more suitable for given settings. The proposed architecture is shown to scale better, at the expense of lower information freshness for less important resources. The proposed definition of resource importance is empirically evaluated in the context of ranking computer clusters.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I am grateful to my parents, Anastasia and Manos, my brother, Kostas, and my partner, Sophie, for their continued love and support.

I am also grateful to my supervisor, Dr Rizos Sakellariou, for his guidance and support over the past four years. His insights helped me to significantly improve this thesis, both in content and presentation.

I would also like to thank the people in the Information Management Group for maintaining a friendly and supportive environment; specifically the ones with which I had the fortune to share office space: Antoon, Desmond, Henan, Jun, Kwanchai, Nedim, Steven, Tasos, Viktor, Wei and Ye.

Finally, I am indebted to the Greek State Scholarships foundation (IKY), for supporting me financially.

Chapter 1

Introduction

1.1 The Advent of Grid Computing

Grid computing [FK99b, FK03, BFH03] aims to realise a software and hardware infrastructure for controlled sharing of networked resources within and across organisations. Grid participants can form *Virtual Organisations* (VOs) [FKT01], that is, *ad hoc* partnerships with no physical headquarters, for pursuing common goals by means of coordinated use of their resources. Supported resources were originally limited to conventional computer resources such as compute power, storage and network capacity, software and databases; this vision has been later extended to anything that it makes sense to share over a network, including domain-specific services and on-line instruments. The diversity of resources led to an informal categorisation of grids into computational, data, knowledge and so on.

In a broader context, the grid paradigm falls into the area of large scale distributed systems; this characterisation can be further specialised, depending on the nature of goals and resources of grids, into high performance, high throughput, data intensive and collaborative computing. Grid computing is the evolution of earlier efforts (e.g., [SC92]) to couple dispersed computers into what would be perceived by end-users as a single supercomputer.

Earlier efforts were mainly motivated by the need to address the so-called Grand Challenges, i.e., important science and engineering problems the solution of which requires unprecedented computational and networking capacity (e.g., [gra93]). In addition to resource-intensive problems, grid computing is further motivated by the demand to support a seamless collaboration of geographically

distributed scientists (e.g., carrying out projects, sharing results), as well as the integration of information technology resources within and across organisations.

The diversity of grid applications, despite significant progress, makes the realisation of general-purpose grids a demanding endeavour. As a result, the original vision of a single worldwide and generic Grid has shifted to the development of numerous special-purpose and isolated grids.[1] A key challenge for the integration of the existing grids into a unified Grid, is the provision of middleware that scales for large numbers of grid resources and users. This thesis focuses specifically on large-scale monitoring, to support users (or programs acting on behalf of users) to discover resources of interest in large grids.

Section 1.2 describes grid monitoring and information services. Section 1.3 motivates the problem of large-scale grid monitoring, and outlines the proposed approach. Section 1.4 states the present work's aims and contributions. Section 1.5 concludes the chapter with an overview of the structure of this dissertation.

1.2 Grid Monitoring and Information Services

Grid middleware aims to provide a standard way of doing common tasks in grid applications, by means of reusable software and standard application programming interfaces (APIs). Common grid tasks include user authentication and authorisation, resource enquiry and discovery, job scheduling, and file management. This dissertation focuses on large-scale monitoring for the purpose of resource discovery in large grids.

Users discover resources and query about their status and (hardware and software) configuration by submitting queries to *grid information services* [FFK+97, CFFK01], or simply information services. Information services are needed for discovering resources that are available in a dynamic and changing environment, and relevant to a user's needs.

Grids are envisioned to be dynamic in the sense that an organisation's memberships can vary over time. When an organisation joins a virtual organisation (VO), certain members of that VO may be authorised to use the new member's resources. The authorisation to use those resources seizes to exist as soon as that

[1] The capitalisation is used to distinguish the single Grid from its constituents. The Grid will be composed of numerous grids, in the same sense that the Internet is composed of numerous internets.

member leaves the VO. Changes to the VO memberships of an organisation may also happen temporarily as a consequence of network failures. Furthermore, the resources that are shared by a certain organisation may become unavailable, due to failures of the resources themselves or intermediate network links. In addition to the dynamism of the environment, there is the issue of how relevant a resource is to a user's requirements. To illustrate, consider two different classes of grid users: a computational linguist may have to look for any Unix-based workstation to run a pipeline of Perl scripts over a subset of Wikipedia, whereas a user of a computational fluid dynamics simulation may need a parallel machine of a given hardware architecture running a specific operating system.

On this basis, information services form a crucial part of grids; they are the grid equivalent of yellow (resource discovery) and white pages (resource enquiry.) An information service collects information about the configuration and status of designated resources, a process known as *monitoring*, and exposes this information to users through a query interface.

The need to monitor on-line resources is not new. Numerous applications are already in place making use of standard (most notably SNMP)[2] and proprietary protocols. However, generic monitoring systems are typically intended for intranet operation and thus do not scale well beyond organisational borders. Proprietary systems in particular are not compatible with the openness and standards-based approach that is needed for the wide adoption of grids. To this end, many existing systems were extended and customised for the grid (e.g., [RVSR98, MCC04, TG03, WSH99, TF03b, BKPV01, BBS+04]) and some grid-specific monitoring and information systems were developed from scratch [CFFK01, CGM+03b]. However, current systems are not intended, neither are capable to scale for Grids in the order of thousands of sites.[3]

1.3 The Problem

This section presents the vision that motivates the present work, discusses the challenges involved and outlines the thesis proposal.

[2]SNMP is the Simple Network Monitoring Protocol, a standard for monitoring and, to some extent, controlling networked devices.

[3]For instance, TeraGrid [ter], one of the largest existing grids as of mid 2007, consists of approximately 10 grid sites.

1.3.1 The Vision of the Grid

The original vision of a single, worldwide Grid, has been replaced with many disconnected grids. In a way, the current state of Grid computing resembles the time before the Internet, when internets[4] were using incompatible inter-networking protocols (as opposed to the TCP/IP protocol suite). This fragmentation of grids is mainly due to political and technical issues:

Scalability The incapacity of middleware to operate seamlessly for increasing numbers of grid administrative domains, resources and users.

Incentives The lack of incentives for users to strive for consensus on common issues, including the development of interoperable middleware. One could argue that an organisation's grid middleware need only be interoperable with the middleware of that organisation's potential partners. However, this line of thought limits the organisation's options for future grid collaboration, as well as the value of using grid middleware in the first place.

Interoperability The lack of integration between competing or overlapping software development efforts. Interoperability issues are partly political (lack of incentives to reach consensus) and partly technical (different user requirements lead to different ways of dealing with a problem).

Of the above issues, the present thesis focuses on the scalability problem, specifically for grid monitoring and information services. That is, the capability of such services to scale gracefully for increasing numbers of monitored resources and user queries. The motivating vision is large-scale information services, such as grid search engines and directories of resources. While web search engines are used to discover web pages of interest, grid search engines would be used for the discovery of grid resources (e.g., clusters of computers) with certain characteristics. In both contexts (web and grid), search engines resolve queries against large volumes of data, and are expected to deliver up-to-date query results in less than a second. The concept of a directory of grid resources is inspired from web directories, such as http://dmoz.org and http://directory.google.com. In the context of grids, directories would classify grid resources, in many different ways, including physical location, pricing, and membership in virtual organisations.

[4] An internet is a network of potentially heterogeneous networks, connected using an inter-networking protocol to mask the potential differences of the underlying networks.

However, in contrast to most web directories, grid directories would be updated automatically (as opposed to manually). The functionality of grid search engines and directories could be exposed to both human consumers (via web front-ends) and programs (via APIs or web services).

1.3.2 Challenges in Large-Scale Grid Monitoring and Information Services

The performance of grid monitoring and information services is mainly measured in terms of

- how quickly queries are answered (query response time);
- how much network traffic a service generates to answer user queries (network overhead); and
- the extent that query results are fresh compared to the actual properties of the resources they describe (information freshness).

A system cannot perform well in all of the above metrics: certain design choices determine the performance trade-offs. An information service can be considered perfectly scalable when it can maintain low query response time, low network overhead and high information freshness, in the face of a large number of monitored resources and a large number of queries.

Grid monitoring is a non-stop activity, and thus, over time, can impose significant network overhead. Network overhead certainly becomes an issue when monitoring a grid of thousands of sites, which are interconnected through expensive wide-area network links. At that scale, it is tempting to reduce network overhead at the expense of query response time, or information freshness. But an information service that requires tens of seconds to respond to queries or returns always stale results is not useful.

1.3.3 Outline of Proposal

All the existing grid monitoring and information services support only a manual configuration of a trade-off among the aforementioned performance metrics, and that trade-off applies to all monitored grid resources. (We call this approach a fixed trade-off, as the trade-off remains the same unless an administrator manually

reconfigures the monitoring service.) The present thesis is that the trade-off in grid monitoring can be automatically adjusted at a site-level basis, taking into account the importance of grid resources at every site. The thesis assumes that resources in a large, heterogeneous grid are not of equal importance. In addition, the thesis includes a generic definition of resource importance, which in the context of the present dissertation is intended to capture the usefulness of computer clusters, as perceived by users.

This proposed approach implies an increased frequency of updates for the sites that host the most important resources, while lower freshness is maintained for those hosting the least important resources. Moreover, the frequency of updates per site can be dynamically adjusted by periodically re-evaluating the importance of all known grid resources.

1.4 Aims and Contributions

The aim of this work is to contribute towards the realisation of large-scale grid information services by providing a scalable monitoring framework, which can be used as a basis for the discovery of resources of interest in a Grid of thousands of sites. In this context, the present dissertation contributes:

- A taxonomy of grid monitoring systems that relates architectural features to scalability, and allows to classify and compare diverse monitoring systems. The taxonomy is used to argue that a certain architectural approach, that will later be introduced as hierarchies of aggregation, has the best potential for scalability.

- A model for optimistic performance estimations of a certain class of hierarchies of aggregation, which will be introduced as just-in-time. The model is used as a basis for comparative evaluation of just-in-time hierarchies of aggregation against the proposed architecture.

- A new approach to large-scale grid monitoring, that introduces the concept of resource importance to vary dynamically the freshness of information about resources across sites.

- An architecture that materialises the importance-aware monitoring approach, which is an adaptation of the web crawling paradigm in the context of grid information services.

- Models for the estimation of the proposed architecture's performance in given problem settings, in terms of information freshness and imposed network overhead.
- A query-independent resource importance definition, that is, a way to quantify the importance of resources without reference to a specific query. The resource importance definition is evaluated in the context of computer clusters.
- An experimental evaluation of the proposed architecture, in several different problem settings, including various sizes of grids, using a large-scale deployment of a prototype implementation.
- A comparison of the proposed architecture against just-in-time hierarchies of aggregation, to assess which of the two approaches is most appropriate in different problem settings.
- A validation of performance models of the proposed architecture against experimentally derived results.

1.5 Thesis Structure

The remainder of the thesis is structured as follows.

Chapter 2 introduces the main terms and concepts that are related to grid monitoring and information services, describes a reference grid monitoring architecture from the Open Grid Forum (formerly known as Global Grid Forum), defines a taxonomy of grid monitoring systems, and characterises a number of existing grid monitoring systems using the presented taxonomy.

Chapter 3 discusses the performance trade-offs that are involved in large-scale monitoring, and their relation to architectural choices. Additionally, the chapter describes, models and assesses the scalability of just-in-time hierarchies of aggregation, which is one of the two main architectural approaches to grid monitoring. The evaluation in this chapter provides a performance baseline, against which the proposed architecture is later compared.

Chapter 4 presents the proposed architecture for importance-aware grid monitoring, along with details of a prototype implementation. The chapter

also defines the performance metrics to be used for the evaluation of the proposed architecture, and models for the estimation of the architecture's performance in given settings.

Chapter 5 complements the architecture in Chapter 4, with a generic definition of resource importance, which allows to quantify the importance of complex resources. Also, the chapter discusses and evaluates variations of the presented importance definition, in the context of computer clusters.

Chapter 6 evaluates experimentally the proposed architecture, based on large-scale deployments of the prototype. The results are contrasted to those of just-in-time hierarchies of aggregation, from Chapter 3. Also, the experimental results are used to test the accuracy of the performance models in Chapter 4.

Chapter 7 reviews the thesis contributions, and discusses potential future work.

In addition to Chapter 2, which classifies existing grid monitoring systems using the proposed taxonomy, related work is also discussed in Chapter 3 (performance modelling of grid monitoring systems), in Chapter 4 (large-scale grid monitoring), and in Chapter 5 (resource ranking and importance).

Chapter 2

Background

This chapter provides background material on grid monitoring, and surveys the current state of the art. The present chapter contributes:

- A taxonomy of grid monitoring systems, which aims to provide a basis for the classification of grid monitoring systems in terms of scope of deployment (and, implementation issues aside, scalability).
- The application of the taxonomy to a wide range of grid monitoring systems.

The chapter is organised as follows. Section 2.1 introduces main concepts and terms. Section 2.2 describes the set of actions that are referred to as monitoring. Section 2.3 summarises a reference grid monitoring architecture, the concepts of which are used in the definition of the taxonomy proposed in Section 2.4. The taxonomy proposed in Section 2.4 is used to survey a number of existing grid monitoring systems. Section 2.5 has a brief discussion on the scalability of the considered grid monitoring systems, and Section 2.6 concludes the chapter.

2.1 Concepts and Terminology

The following terms are defined, mainly drawn from [TAG+02]:

An entity, as defined in [PDH+02], is any networked resource, which can be considered useful, unique, having a considerable lifetime and general use. Typical entities are processors, memories, storage mediums, network links, and applications.

An event is a collection of timestamped, typed data, associated with an entity, and represented in a specific structure.

An event type is an identifier which uniquely maps to an event structure.

An event schema or simply schema, defines the typed structure and semantics of all events so that, given an event type, one can find the structure and interpret the semantics of the corresponding event.

A sensor is a process monitoring an entity and generating events. Sensors are distinguished in passive (i.e., use readily available measurements, typically from operating system facilities) and active (i.e., estimate measurements using custom benchmarks). The former typically provide operating-system-specific measurements while the latter are more intrusive.

2.2 The Monitoring Process

Monitoring distributed systems, and hence grids, typically includes four stages [MS92, MSS93]:

1. *generation* of events, that is, sensors enquiring entities and encoding the measurements according to a given schema;

2. *processing* of generated events is application-specific and may take place during any stage of the monitoring process, typical examples include filtering according to some predefined criteria, or summarising a group of events (e.g., computing average values);

3. *distribution* refers to the transmission of events from one computer to another;

4. finally, *presentation* typically involves some further processing so that the overwhelming number of received events will be provided in a series of abstractions, to enable an end-user to draw conclusions about the operation of the monitored system. A presentation, typically provided by a GUI application making use of visualisation techniques, may either use a real-time *stream* of events or a recorded *trace* usually retrieved from an archive. However, in the context of grids, we generalise the last stage as *consumption*

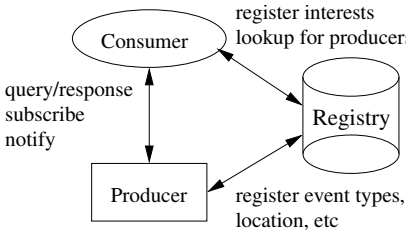

Figure 2.1: The GGF Grid Monitoring Architecture.

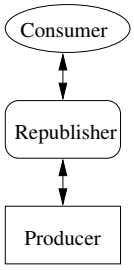

Figure 2.2: A republisher implements both producer and consumer interfaces.

since the users of the monitoring information are not necessarily humans and therefore visualisation may not be involved.

2.3 A Reference Grid Monitoring Architecture

This section provides a brief overview of the Grid Monitoring Architecture (GMA) [TAG+02] by the Open Grid Forum. GMA is of interest because the taxonomy in the next section builds on its concepts. The main components of the GMA are as follows (Figure 2.1):

A producer is a process implementing at least one producer Application Programming Interface (API) for providing events.

A consumer is any process that receives events by using an implementation of at least one consumer API.

A registry is a lookup service that allows producers to publish the event types they generate, and consumers to find out the events they are interested in.[1] Additionally, a registry holds the details required for contacting registered parties (e.g., address, supported protocol bindings, security requirements.) Even for systems with no notion of events, registries can be useful for producers and consumers discovering each other.

Interactions After discovering each other through the registry, producers and consumers communicate *directly* (i.e., not through the registry.) GMA defines

[1]The GMA document [TAG+02] refers to the registry as a directory service, which suggests the use of an engine based on the Lightweight Directory Access Protocol (LDAP). However, as LDAP is not a requirement, we use the implementation-agnostic term "registry".

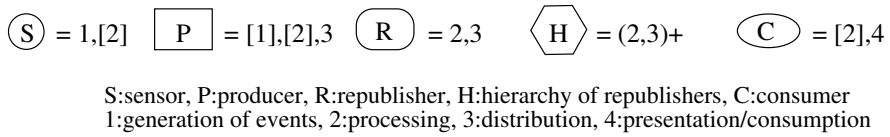

Figure 2.3: Mapping GMA components to phases of monitoring. Square brackets and parentheses indicate optional and grouped expressions respectively, whereas "+" stands for at least one repetition of the preceding item (see text for further explanation.)

three types of interactions between producers and consumers. *Publish/subscribe* refers to a three-phase interaction consisting of a subscription for a specific event type, a stream of events from a producer to a consumer, and a termination of the subscription. Both the establishment and the termination of a subscription can be initiated by any of the two parties. A *query/response* is an one-off interaction initiated by a consumer and followed by a single producer response containing one or more events. Finally, a *notification* can be sent by a producer to a consumer without any further interactions.

In addition to the three core components, the GMA defines a republisher (referred as compound component or intermediary) and a schema repository.

A republisher is any single component implementing both producer and consumer interfaces (Figure 2.2) for reasons such as filtering and aggregating.

A schema repository holds the event schema, that is, the collection of defined event types. If a system is to support an extensible event schema, such a repository must have an interface for dynamic and controlled addition, modification and removal of custom event types.

Republishers and the schema repository are considered as optional components, though one can see that they are essential parts of any sophisticated monitoring framework. The schema repository may be part of the registry, but in any case these two components must be replicated and distributed to allow for distribution of load and robustness.

The GMA, being an architecture, does not define implementation details such as employed data model, event schema, protocol bindings, registry engine and so on. Probably the most important feature of the GMA is the separation of the discovery and retrieval operations (i.e., discover from the registry and retrieve from producers or republishers.)

Revisiting GMA Because GMA's components are fairly general, we correlate its main components to the phases of the monitoring process (as described in Section 2.2.) As shown in Figure 2.3,

a sensor (shown by a circle) must generate events (i.e., the first phase of monitoring), may process them and may make them available to local consumers only (e.g., through a local file);

a producer (depicted as a box) may implement its own sensors, may process events (generated by built-in or external sensors) and must support their distribution to remote consumers, hence the producer interface;

a republisher (shown as a rounded box) must apply some type of processing to collected events and make them available to other consumers;

a hierarchy of republishers (shown as a polygon) consists of one or more (hence, the "+" sign) republishers;

a consumer (depicted as an ellipse) may apply some processing before presenting the results to the end user or application.

2.4 A Taxonomy of Grid Monitoring Approaches and Frameworks

The previous section has refined the GMA components by mapping them to phases of the monitoring process. This section proposes a taxonomy of monitoring systems, which is primarily concerned with a system's provision of GMA components (as they were defined in Figure 2.3.) The categories of the proposed taxonomy are named from zero to three depending on the provision and characteristics of a system's producers and republishers (Figure 2.4.)

Level 0 Events flow from sensors to consumers in either an on-line or an off-line fashion (i.e., at the time of measurements being taken or afterwards, using a trace file.) In the on-line case, the sensors store locally any measurements, which are accessed in an application-specific way. This typically is via a web interface that provides interactive access to HTML-formated information that includes measurements, hence not what one would consider a generic

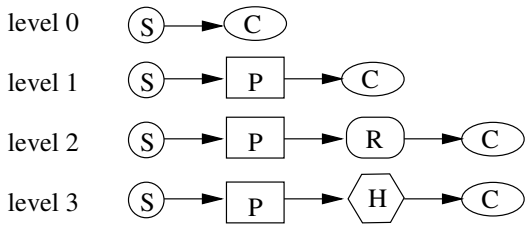

Figure 2.4: The categories of the proposed taxonomy of monitoring systems. Note that, although not shown to avoid clutter, the sensor symbol can be omitted in systems where producers have built-in sensors.

API. The distinguishing feature of level zero or *self-contained* systems is the lack of producer APIs that would enable the distribution of events to remotely located components, in a programmable fashion (as opposed to non-programmable such as web pages intended for interactive use.)

Level 1 In *first level* systems, sensors are either separately implemented and hosted at the same machines with producers, or their functionality is provided by producers. In either case, events are remotely accessible via a generic API provided by producers.

Level 2 In addition to producers, *second level* monitoring systems feature at least one type of republisher, which however has a fixed functionality. Republishers of different functionality may be stacked upon each other but only in predefined ways. A second level system is differentiated by a functionally equivalent first level system by the distribution of the functionality (that would otherwise be provided by a single producer) among different hosts.

Level 3 Highly flexible monitoring systems, apart from producers, provide republishers that are configurable, allowing their organisation in an arbitrarily-structured hierarchy. As explained earlier, in a hierarchy of republishers every node collects and processes events from lower level producers or republishers for reasons such as the provision of customised views or preparation of higher-level events. *Third level* systems have a potential for scalability and may form a standalone Grid Information Service (assuming support for a variety of grid resources.) Whether they are actually scalable depends on implementation choices, and the multiplicity and distribution of components according to the arbitrary hierarchy that is adopted in specific

deployments. Since the actual scalability depends on configuration and implementation choices, the taxonomy simply highlights which systems have the potential for scalability.

The taxonomy includes a *multiplicity* qualifier to capture whether republishers in second level systems are centralised (i.e., one republisher), merely distributed, or distributed with support for replication. The multiplicity (and distribution) of producers over sensors (in systems of all levels) is not significant since, according to the given definition, at least one producer is required per monitored host. The multiplicity of republishers over producers in third level systems can vary based on the adopted hierarchy.

Another qualifier refers to the *type of entities* that are primarily monitored by a considered system. This qualifier can be any of hosts, networks, applications, availability and generic. The last one denotes general-purpose systems that support events for at least hosts and networks.

Last, the *stackable* qualifier denotes whether a monitoring system is intended to employ another system's producers or republishers, in other words operate on top of it. A stackable, say, first level system can be more promising than a non-stackable system of the same level, because the former encapsulates (part of) the functionality of another monitoring system.

Based on the above categories and qualifiers, considered systems are characterised using the form L{0-3}.{H,N,A,V,G}.[S], where the number denotes the level, the following letter the nature of monitored entities (Hosts, Networks, Applications, aVailability, Generic), and an optional S implies whether the system is stackable. Specifically for second level systems, an a, b or c letter follows the level number to denote (a) a single centralised republisher; (b) more than one distributed republisher; and (c) distributed republishers supporting data replication. For instance, L2c.H denotes a second level system that is concerned mainly with hosts and has two or more republishers, with support for data replication.

Goals and Audience The goal of the taxonomy is to provide a simple means to describe a monitoring system's features with respect to:

- provision of core GMA components;
- main target of monitored entities (e.g., network, hosts, etc.); and
- whether a system can or has to operate on top of another system.

Developers would be primarily interested in systems that provide their functionality via APIs; thus they would be looking for at least L1 systems. Administrators that are interested in performance or reliability will have to look for at least L2b (distributed republishers) and L2c (distributed and replicated republishers) systems, respectively; they can also identify systems that depend or may operate on top of other systems with the stackable qualifier. Users who need general-purpose systems which, when appropriately deployed, can scale beyond organisation-wide installations need to look for L3 systems. Users and others can also find useful the monitored entities qualifier to identify generic systems or those primarily concerned with hosts, networks, applications, or availability.

2.4.1 Level 0: Self-Contained Systems

A monitoring system is characterised as "self-contained" when it does not expose its functionality through a producer interface. To this end, such a system can be used only in predefined and rigid ways (e.g., through a GUI.) The only self-contained system considered here is MapCenter, an administration system for monitoring the availability of grid resources, via a web front-end.

MapCenter (L0.V.S)

MapCenter [BHP02], developed as part of the EU DataGrid project, is a monitoring application which provides web users a visualisation of the availability and distribution of services throughout a Grid. It is intended as a grid administration tool for tracking availability problems. As of mid 2007, MapCenter was deployed in more than ten major grid installations, including the EU DataGrid.

Overview MapCenter builds and periodically updates a model of the network services available in a grid, and provides this information in several logical views (sites, Virtual Organisations (VOs), applications, geographical) through a web interface. The information provided by MapCenter is about the *availability* of grid nodes and their services (e.g., the daemons of Globus' Monitoring and Discovery Service (MDS), etc); hence MapCenter does not keep details concerning configuration and utilisation of resources. However, it does allow users to dynamically query an MDS server (using a PHP based LDAP client), ping and otherwise interact with hosts (using CGI scripts.)

MapCenter has a number of coordinated threads, polling specific ports of a set of hosts according to a configuration file. In addition, there is support for automatic discovery of UDP, TCP, and HTTP based (e.g., web) services, by means of probing to well-known ports [BHP04].

MapCenter's configuration along with the retrieved status information is kept in a flat text file. The latter is used to update periodically a set of HTML files which present the previously mentioned logical views. Of these, the graphical view illustrates the nodes on a geographical map using localisation information retrieved from the WHOIS Internet service.

In the taxonomy's context, MapCenter's polling threads are considered *sensors* because they merely store locally any acquired events, as opposed to providing them via a producer API. To this end, MapCenter is classified as L0.V.S because it lacks producers (level zero), is concerned with hosts and services availability, and operates on top of existing information services (stackable.) MapCenter can be converted to a first level system by providing the monitoring events through a producer API.

2.4.2 First Level: Producer-Only Systems

As an example of a first level system, this section presents Autopilot, a framework for implementing self-adapting applications.

Autopilot (L1.A)

Autopilot [RVSR98] is a framework for enabling applications to dynamically adapt to changing environments. This run-time tuning ability is important for applications that have to choose among a variety of policies (e.g., schedulers) as well as those that need to adapt to a dynamic environment, such as the grid. Autopilot's main ideas are leveraged in the Grid Application Development Software (GrADS) project [BCC+01], which aims to support end-users in the development, execution and tuning of grid enabled applications. In GrADS, grid applications are characterised with performance requirements (specified in so-called "contracts"); among others, a real-time monitor compares an application's progress against the requirements of its contract and triggers corrective actions in case of violations [VAMR01, RM05].

Overview Application adaptivity requires real-time performance measurements (i.e., generation of application-specific events), reasoning whether there is a need for a corrective action and, if so, instructing the application to perform the latter. Autopilot's functionality is implemented in separate components, namely sensors, actuators, clients and distributed name servers.

Applications instrumented for Autopilot include sensors and actuators for remotely reading and writing respectively, application-level variables. Sensors and actuators are described by property lists (e.g., name, location, type of variable measured/controlled, etc), have attached functions and register themselves to a name service (i.e., a *registry*). Property lists are used by clients (i.e., *consumers*) to lookup in the registry for sensors. Attached functions implement data reduction techniques (e.g., summarisation) in case of sensors or perform requested actions in case of actuators.

An Autopilot client finds out "interesting" sensors through the registry and subscribes for receiving their events. Subsequently, a client uses an application-specific logic – that is defined as a fuzzy logic rule base – to make decisions and, if applicable, instruct an actuator to perform adaptive actions. In addition, clients manage sensors, in terms of activation and sampling frequency, through appropriate actuators.

Autopilot's events are represented in either binary or ASCII encodings according to the Pablo *Self-Defining Data Format* [Ayd92]. As the name implies, SDDF encoded events include descriptions of their structure, though the actual semantics are not specified. The binary format can be employed between machines of same or different byte order conventions, whereas the ASCII encoding must be used in cases of different word length or floating point representations.

In the context of the taxonomy, the Autopilot sensors operate as *producers* since they not only take measurements but also provide the means for accessing them remotely. Combined with the focus on application monitoring, Autopilot is classified as L1.A.

2.4.3 Second Level: Producer and Republisher Systems

This section is concerned with monitoring systems that include producers and one or more special purpose republishers. The considered systems, described in alphabetical order, are listed here under the three subcategories of second level systems.

Centralised Republisher a monitoring framework for systems administration based on the CODE system; GridICE, an administration system for monitoring the availability and utilisation of grid resources; Hawkeye, an administration system for monitoring and management of computer clusters;

Distributed Republishers GridRM, a proposal for integrating the diverse monitoring sources typically available in a grid site; HBM, an unreliable fault detector of fail-stop failures; NetLogger, an application performance analysis toolkit that was extended with components for providing a producer interface and controlling application-level sensors; OCM-G, an interactive-applications monitoring system for execution steering and performance analysis; Remos, a prototype similar to NWS that additionally provides logical views of network topologies; SCALEA-G, an extensible, service-oriented monitoring and performance analysis system for both applications and resources.

Distributed Republishers with Replication NWS, a network monitoring and forecasting service that provides end-to-end measurements and predictions.

CODE based Monitoring System (L2a.G)

The considered system [Smi02] is concerned with monitoring and managing organisation-wide Globus based grid installations, and is used in the NASA Information Power Grid (IPG.) It is based on CODE [Smi01], a framework for Control and Observation in Distributed Environments, primarily intended to support computer systems administration.

Overview The CODE framework includes observers, actors, managers and a directory service. Each observer process manages a set of sensors and provides their events through an event producer interface, hence acting as a *producer*. Every actor process can be asked, through an actor interface, to perform specific actions, such as restarting a daemon or sending an email. A manager process consumes the events from one or more producers and, based on a management logic (a series of if-then statements, or an expert system), instructs actors to perform specific actions.

Producers and actors register their location, events and actions in a an LDAP based *registry* where managers lookup for the appropriate producers and actors.

A management agent, consisting of an observer, an actor and a manager, is placed in each Grid Resource and Allocation Manager (GRAM) and Grid Information Service (GIS) server of a Globus installation. Events generated by management agents are forwarded to an event archive which is discovered through the registry. A GUI management front-end (i.e., a *consumer*) retrieves events from the archive (a *republisher*) to illustrate the current status of hosts and networks. Also, the management application allows a user to perform simple actions, such as Globus daemons and user accounts management, based on what is advertised in the registry. The GGF's XML producer-consumer protocol is employed for exchanging events, while the event archive is an XML database queried using XPath.

The CODE monitoring system is classified as L2a.G, as it is intended for hosts, networks and services, and it only has a single republisher.

GridICE (L2a.G.S)

GridICE [ABF+05] was developed as part of the DataTag project to support grid administrators. It provides status and utilisation information at Virtual Organisation, site and resource level, as well as basic statistics derived from historical traces and real-time alerts, through a web front-end or an HTTP based API.

Overview GridICE has a centralised architecture where a main server periodically queries a set of nodes to extract information about the status of grid and network services, and the utilisation of resources. The main server is based on Nagios, an open source, host and network service monitor that can be easily extended by the use of custom monitoring and notification plugins. GridICE has an MDS plugin (discussed in Section 2.4.4) for periodically querying Globus index information servers and information providers, whereas other plugins can be built, say, for R-GMA. The collected information is stored in an LDAP server and used to build aggregate statistics (e.g., total memory per site), trigger alerts and dynamically configure Nagios to monitor any newly discovered resources. End-users access the service through a web front-end, which includes logical views at VO, site and entity level as well as a geographical map.

GridICE employs a custom extension of the GLUE schema [ABF+] to support network-, and process-related events. There are concerns in terms of scalability

given the centralised architecture and the frequent polling that has to be performed. A way of resolving this could be to distribute the overall load among several Nagios servers organised in a hierarchy.

Earlier versions of GridICE [ABF+03] were classified either as a zero or a first level system depending on whether the information of interest is the abstracted events or the raw measurements, respectively. However, more recent versions of GridICE have a republisher that provides resource information as XML documents via HTTP, thus making it a second-level system with a centralised republisher.

GridRM (L2b.G.S)

Grid Resource Monitoring (GridRM) [BS03] is a research project that aims to provide a unified way of accessing a diverse set of monitoring data sources that are typically found in grid environments (e.g., Simple Network Management Protocol (SNMP) [CFSD90], Ganglia, NetLogger, Network Weather Service (NWS), etc.)[2]

Overview In GridRM, every organisation has a Java based gateway that collects and normalises events from local monitoring systems. In this respect, every gateway operates as a *republisher* of external (to GridRM) *producers*. A global *registry* is used to support consumers in discovering gateways providing information of interest.

Each gateway has a global and a local layer. The former includes an abstract layer which interfaces with platform-specific consumer APIs (Java, Web/Grid Services, etc) and a security layer that applies an organisation's access control policy. The local layer, has several components including an abstract data layer and a request handler. The latter receives consumer queries from the global layer and collects real-time or archived data from appropriate sources depending on the query's type (last state or historical). The abstract data layer includes several JDBC based drivers, each one for retrieving data from a specific producer. The Java Database Connectivity (JDBC) interface is Java's standard way of interoperating with databases; GridRM hides the diversity of monitoring sources behind JDBC's widely used interface.

Gateways represent events according to the GLUE schema. Consumers form and submit SQL queries using GLUE as the vocabulary, and gateways forward

[2]Ganglia, NetLogger and NWS are considered in later sections.

the queries to the appropriate drivers.

In the context of the taxonomy, GridRM is classified as L2b.G.S, namely a stackable, general-purpose, second level system with distributed republishers.

Hawkeye (L2a.G)

Hawkeye [haw], is a monitoring and management tool for clusters of computers. Hawkeye uses some technology from Condor [LLM88], but it is available as a standalone distribution for Linux and Solaris.

Overview Every monitored node hosts a monitoring agent (*producer*) that periodically calculates a set of metrics, which reflect the host's state, and communicates them to a central manager. The metrics are represented in XML-encoded Condor's classified advertisements (classads) [RLS98], that is, simple attribute-value pairs with optional use of expressions.

The central manager (*republisher*) indexes the current state of nodes for fast query execution, and periodically stores it into a round robin database to maintain an archive. The monitoring information in the central manager, in addition to an API, can be accessed via command line utilities, and web and GUI front-ends.

Administrators can submit jobs to monitored nodes, either for unconditional execution or to be triggered as a response to specific events (e.g., when disk space is running out.)

Hawkeye is a second level, general-purpose monitoring system with a centralised republisher, i.e., L2a.G.

HBM (L2b.V)

The Globus Heartbeat Monitor [SFK+98] (HBM) is an implementation of an unreliable fault-detection service of fail-stop failures of processes and hosts. A fault detector is considered unreliable if there is a possibility of erroneously reporting failures. A fail-stop failure of a component refers to the class of failures that are permanent and can be detected by external components. HBM was employed in early versions of Globus to verify the availability of grid services, but has been dropped due to the later adoption of soft-state protocols (i.e., services subscribe to a registry and periodically renew their subscription, which otherwise expires.)

Overview HBM consists of local monitors (*producers*), data collectors (*republishers*) and *consumers*. A consumer can be a program that is responsible for the availability of specific services, or a process of a distributed parallel program. In HBM, a local monitor has to be running in monitored hosts, and hosts of monitored processes. Every monitored process registers to the local monitor residing in the same host. A local monitor periodically detects the state of all the monitored processes and communicates their status to interested data collectors. Upon the receipt of messages from local monitors, data collectors have to determine the availability status of specific processes, and notify accordingly any previously registered consumers.

With respect to the taxonomy, HBM is characterised as L2b.V, namely a second level system with distributed republishers that is concerned with the availability of hosts and processes.

NetLogger (L2a.A)

The Network Application Logger Toolkit (NetLogger) [TG03] is used for performance analysis of complex systems such as client-server and/or multi-threaded applications. NetLogger combines network, host and application events and thus provides an overall view that helps to identify performance bottlenecks.

Overview NetLogger has four components: an API and its library (available for C, C++, Java, Perl and Python), tools for collecting and manipulating logs (i.e., events), host and network sensors (typically wrappers of Unix monitoring programs), and a front-end for visualisation of real-time or historical events.

An application is manually instrumented by invoking NetLogger's API calls typically before and after (disk or network) I/O requests and time-consuming computations. Events are tagged by the developer with a textual description, and by the library with a timestamp and some host and network related events. The generated events are stored to either a local file, a syslog daemon or a remote host. Prior to transmission, events are locally buffered to minimise the overhead imposed in high rates of generation.

In terms of data encoding, NetLogger supports the text based Universal Logger Message format (ULM), along with binary and XML based encodings, allowing developers to choose the imposed overhead. In addition, the API was extended to allow for dynamic (de)activation of logging by periodically checking

a local or remote configuration file. Concerning robustness, another extension to the API supports dynamic fail-over to a secondary logging destination in case the original remote host becomes unavailable.

GMA-like NetLogger Application Monitoring In an attempt to line up NetLogger with the GMA concepts, [GTJ+02] extended NetLogger's framework by adding a monitoring activation service which is further elaborated in [TGL+03, GTTV03]. An activation service consists of three components: an activation manager and an activation producer per installation (*republisher*), and an activation node per host (*producer*).

An activation manager holds the logging detail required for applications instrumented with NetLogger (including an option for deactivation of logging) and a simple client is provided for setting these values remotely. Each activation node periodically queries the activation manager for the required logging levels and communicates this information to the local applications through configuration files that are periodically checked by the latter. Applications are instructed to log events in a local file from where the activation node forwards them asynchronously to the activation producer. The latter passively accepts events from activation nodes and matches them to consumer subscriptions, expressed as simple filters.

All interactions among the activation service components employ pyGMA, which is a Python SOAP based implementation of producer, consumer and registry interfaces similar to those defined in the GGF XML producer-consumer protocol [SGQ02]. In contrast, the transfer of events from activation nodes to producers can be done using any of the NetLogger transport options, namely ULM, binary or XML formats.

Remarks Considerable intrusiveness is introduced because of activation nodes having to periodically poll the activation manager, and applications to periodically check their configuration file. Also, instead of manual configuration, a registry could be used to support the dynamic discovery of activation service components. Given a registry where all components would be subscribing their details, it would be far less intrusive to have the activation manager to inform activation nodes of logging level updates instead of the current design where activation nodes have to poll activation managers every five seconds.

NetLogger is classified as L2a.A, namely a second level system with a centralised republisher that is intended for application monitoring.

NWS (L2c.N)

The Network Weather Service is a portable (ANSI C based) and non-intrusive performance monitoring and forecasting distributed system, primarily intended to support scheduling and dynamic resource allocation [WSH99, WMOS03, Wol03].

Overview In NWS, a host employs sensors for estimating CPU load, memory utilisation and end-to-end network bandwidth and latency for all possible sensor pairs. Sensors combine passive and active monitoring methods, to accomplish accurate measurements, and are stateless to improve robustness and minimise intrusiveness. Network sensors in particular employ a set of techniques for avoiding conflicts among competing sensors. Sensors are managed through a sensor control process and their events are sent to a memory service, both of which can be replicated for distribution of load and fault-tolerance. All components subscribe to an LDAP based *registry* (referred as name service), using a soft-state protocol.

A forecasting process consumes events from the memory service to generate load predictions using a variety of forecasting libraries. A CGI based front-end exposes current performance measurements and predictions to end-users.

NWS has a small number of sophisticated and portable sensors, while there are prototypes for disk I/O, Network File System (NFS) and system availability. Current interfaces include C, LDAP and Globus Monitoring and Discovery Service (MDS) wrapper shell scripts, and web services.

In terms of the taxonomy, NWS sensors operate as *producers* (i.e., measure and disseminate events), and memory services and forecasters serve as *republishers*; forecasters always operate on top of memory services. Since memory services can be configured for replication, NWS is classified as L2c.N.

Topomon [dBKB02] extends NWS to provide network topology information, which can be used to compute minimum spanning trees between two given hosts, in terms of latency or bandwidth. As part of that extension, Topomon has a republisher operating on top of NWS memory and forecasting processes, and employs GGF's XML producer-consumer protocol.

OCM-G (L2b.A)

OMIS Compliant Monitor (OCM-G) [BBS+04] is a monitoring system for interactive grid applications, developed as part of the EU CrossGrid project [cro]. OCM-G is a grid-enhanced implementation of the On line Monitoring Interface Specification (OMIS) [IW97]. OMIS defines a standard interface between instrumented applications and consumers.

Overview OCM-G has per-host local monitors and per-site service managers. Local monitors have a *producer* interface for disseminating events generated by statically or dynamically instrumented applications (*sensors*.) End-user performance tools (*consumers*) address commands to service managers (*republishers*), which in turn contact the local monitors of the involved applications.

OCM-G supports three kinds of services: on-demand retrieval of events; manipulation of running applications for performance-enhancement and steering; and execution of actions upon the detection of specific events.

OCM-G is intended to be customisable with respect to performance/overhead trade-offs (such as the buffer size for storing the events in local monitors). This is due to the emphasis on interactive applications that require low-latency monitoring to support real-time user feedback. Also, OCM-G defines numerous low-level events (so-called metrics) to allow performance tools to define composite events with custom semantics.

In addition, OCM-G allows enquiries for host- and network-related events, to facilitate performance analysis (i.e., as in NetLogger), and supports certificate-based authentication and access control of users as well as local monitors and service managers.

In the context of the taxonomy, OCM-G is characterised as L2b.A, i.e., a second level system with distributed republishers that is focused on application monitoring.

Remos (L2b.N.S)

The REsource MOnitoring System (Remos) provides to network-aware applications an application programming interface (API) for run-time enquiry of performance measurements of local and wide area networks [DGL+98, DGK+01].

Overview Remos has a query based interface featuring two abstractions, namely *flows* and *network topologies*. A flow represents a communication link between two applications. In contrast, a network topology graph provides a logical view of the physical interconnection between compute and network nodes, annotated with information on link capacity, current bandwidth utilisation and latency.

Remos has several types of collectors, a modeller and a prediction service. A variety of collectors is employed to accommodate the heterogeneity of networks: SNMP and bridge collectors for SNMP-aware routers and Ethernet switches respectively; benchmark collectors for network links where SNMP is not available, typically wide area network (WAN) links. SNMP and bridge collectors correspond to *republishers* making use of external *producers* (SNMP); benchmark collectors are *producers* that implement active sensors.

In addition, every site has a master collector, which accepts queries from modellers. A master collector coordinates the appropriate collectors for the execution of a given query, and collects, merges and sends the results to the query's originator, effectively acting as a higher level *republisher*. These results are raw measurements and it is the modeller's responsibility to build the (flow or topology) abstractions before forwarding them to the application. Every application has its own modeller, which is a process running on the same host, which makes modellers part of *consumers*.

In addition to current load measurements, Remos' API supports predictions for host load and network events (e.g., bandwidth utilisation and latency), of which only the former was implemented as of mid 2007, using the RPS toolkit [DO99].

Because of the complexity involved in networks (e.g., routing protocols, heterogeneity of devices, etc), Remos provides "best effort" measurements annotated with statistical parameters (e.g., standard deviation, confidence.) Remos' design focuses on the provision of a consistent interface, independently of the underlying network technicalities, and on portability, hence the use of SNMP and simple system-independent benchmarks.

Remos employs a variety of producers (external SNMP producers, benchmark collectors) and republishers (SNMP collectors, master collectors, predictor), which however have to be connected in a predefined way. On this basis, Remos is classified as L2b.N.S, that is, a stackable (since it operates on top of SNMP) second level system with optional support for multiple first-level republishers (SNMP

collectors) per installation.

SCALEA-G (L2b.G)

SCALEA-G is an extensible, service-oriented[3] instrumentation, monitoring and performance analysis framework for hosts, networks and applications.

Overview SCALEA-G [TF03b] implements a variety of grid services (see Section 2.4.4), including sensor, sensor manager, instrumentation, archival, registry and client services. A sensor manager service interacts with sensor service instances, which may be on the same or different hosts. This implies that SCALEA-G sensors are equivalent to *producers* and sensor managers to *republishers*. An archival service provides persistent storage of monitoring information. Producers and republishers are registered in and discovered from a sensor repository and a directory service, respectively. (The directory service and sensor repository services combined, provide the functionality of a *registry*).

An application instrumentation service is partially built on existing systems (SCALEA [TF03a] and Dyninst [BH00]) to support source-level and dynamic instrumentation. Consumers interact with the instrumentation service using an XML based instrumentation request language that defines request/response messages.

All messages are encoded in XML according to predefined schemas, and consumers can pose XPath/XQuery queries or establish subscriptions with sensor manager services. Globus' Grid Security Infrastructure (GSI) is employed to provide security services such as consumer authentication and authorisation. SCALEA-G is complemented by GUI programs for configuring the system and conducting performance analysis.

SCALEA-G is a second level general-purpose monitoring system with distributed republishers, namely L2b.G.

2.4.4 Third Level: Hierarchy of Republishers

This section focuses on third level monitoring systems, that is, frameworks featuring producers and general purpose republishers which can form an arbitrarily structured hierarchy. The considered systems are: Ganglia, a fairly scalable and

[3]SCALEA-G is implemented using the Open Grid Services Architecture (OGSA), which has been superseded by the newly adopted Web Services Resource Framework (WSRF.)

widely used cluster monitoring system; Globus MDS, the Monitoring and Discovery Service of the most widely deployed grid middleware; Mercury, GridLab's organisation-level monitoring system; MonALISA, a Jini based monitoring prototype for large distributed systems; and R-GMA, a relational approach to GMA that is intended to operate as a standalone Grid Information Service (GIS). MDS, MonALISA and R-GMA can be configured for stackable operation, i.e., run on top of other monitoring systems.

Ganglia (L3.G)

Ganglia [MCC04] is an open source hierarchical monitoring system, primarily designed for computer clusters but also used in grid installations. Ganglia is a mature monitoring system with numerous deployments.

Overview At the cluster level, membership is determined with a broadcast, soft-state protocol (soft-state means that membership must be periodically renewed by explicit messages or otherwise expires). All nodes have a multi-threaded daemon (Ganglia monitoring daemon) performing the following tasks:

- Collecting and broadcasting External Data Representation (XDR) encoded events from the local host.
- Listening the broadcasts sent by other nodes and locally maintaining the cluster's state.
- Replying to consumer queries about any node in the local cluster, using XML encoded messages.

Given the above actions, a cluster's status is replicated among all nodes, which act as *producers*, resulting in distribution of load and fault-tolerance, but also in high network and host intrusiveness.

An arbitrarily structured hierarchy of *republishers* (referred as Ganglia meta-daemons) periodically collect and aggregate events from lower level data sources, store them in round-robin databases, and provide them on demand to higher level republishers. Data sources may be either producers (on behalf of a cluster) or other republishers (on behalf of several clusters); in both cases an XML-encoding is employed.

Ganglia does not have a registry and therefore the location of producers and republishers must be known through out-of-band means. The databases serve as archives and are also used by a web based visualisation application providing cluster- and node-level statistics. Simple command line utilities are provided for adding new event types and querying producers and republishers.

Remarks Ganglia introduces a considerable, albeit linear, overhead both at hosts and networks at cluster and hierarchy levels, because of the cluster-wide updates in the former, and XML event encoding in the latter. The network intrusiveness imposed by republishers connected through WAN links is of considerable importance given the associated costs. Other concerns include the availability of IP multicast, and the lack of a registry since Ganglia was primarily intended for clusters, which are fairly static compared to grids.

Globus MDS (L3.G.S)

The Monitoring and Discovery Service [FFK+97, CFFK01, SRP+06], formerly known as the Metacomputing Directory Service, constitutes the information infrastructure of the Globus toolkit [FK99a]. MDS, and Globus as a whole, have been through several radical transformations due to paradigm shifts in grid middleware. MDS versions 1 and 2 followed the Unix/C style of development; MDS 3 and 4 were implemented using grid and web services, respectively. The present discussion considers version 2 onwards, as MDS2 is still in use.

Overview MDS 2.x is based on two core protocols: the Grid Information Protocol (GRIP) and the Grid Registration Protocol (GRRP.) The former allows query/response interactions and search operations. GRIP is complemented by GRRP, which is for maintaining soft-state registrations between MDS components.

The Lightweight Directory Access Protocol (LDAP) [WHK97] is adopted as a data model and representation (i.e., hierarchical and LDIF respectively – LDAP Directory Interchange Format), a query language and a transport protocol for GRIP, and as a transport protocol for GRRP. Given the LDAP based hierarchical data model, entities are represented as one or more LDAP objects defined as typed attribute-value pairs and organised in a hierarchical structure, called the Directory Information Tree (DIT.)

The MDS framework has information providers (sensors), Grid Resource Information Services (GRIS – *producers*) and Grid Index Information Services (GIIS – *republishers*.) Both producers and republishers are implemented as backends for the open source OpenLDAP server implementation.

Producers collect events from information providers, either from a set of shell scripts or from loadable modules via an API. In addition, producers provide their events to republishers or to consumers using GRIP, and register themselves to one or more republishers using GRRP.

Republishers form a hierarchy in which each node typically aggregates the information provided by lower level republishers (and producers in case of first level republishers.) Republishers use GRIP and GRRP as part of the consumer and producer interfaces, though custom implementations could offer alternative producer interfaces (i.e., relational.) Several roles may be served by republishers, including the provision of special purpose views (e.g. application specific), organisation-level views and so on.

Consumers may submit queries to either producers or republishers, or discover producers through republishers, in any case using GRIP.

Remarks The hierarchical data model along with LDAP's referral capability (i.e., forward a query to an authoritative server) accommodates well the need for autonomy of resource providers and decentralised maintenance. Also, MDS supports security services, such as access control, through the use of the Grid Security Infrastructure (GSI) [FKTT98]. However, LDAP features a non-declarative query interface that requires knowledge of the employed schema. In addition, the performance of OpenLDAP's update operation – which is by far the most frequently used – has been very much criticised.

MDS3 Globus was re-designed and implemented as part of the *Open Grid Services Architecture* (OGSA) [FKNT02] and Open Grid Services Infrastructure (OGSI) [TCF+03] specifications. OGSA is a web services based architecture that aims to enhance interoperability among heterogeneous systems through service orientation (i.e., hiding the underlying details by means of consistent interfaces.) In OGSI, the implementation-part of the OGSA architecture, everything is represented as a grid service, that is, a web service that complies to some conventions, including the implementation of a set of grid service interfaces (portTypes in WSDL terminology.) Every grid service exposes its state and attributes through

the implementation of the GridService portType and, optionally, the NotificationSource portType, which correspond to pull and push data delivery models respectively.

In this respect, the functionality of the MDS2 Grid Resource Information Service (GRIS) is encapsulated within grid services. In OGSA, the equivalent of the MDS2 Grid Index Information Service (GIIS) is the Index Service which, among others, provides a framework for aggregation and indexing of subscribed grid services and lower level Index Services. Index Services are organised in a hierarchical fashion just like the GIISs in MDS2.

Information is represented in XML according to the GLUE schema. Simple queries can be formed by specifying a grid service and one or more service data elements, whereas more complex expressions are supported using XPath.

MDS4 MDS4 [SRP+06] is a re-engineering of MDS3, in which OGSI's grid-specific conventions have been replaced by web services specifications, most notably, the Web Services Resource Framework (WSRF). Thus, the two versions do not differ significantly in terms of functionality. MDS4 can use various external information services as producers, including Ganglia and Hawkeye.

Mercury (L3.G)

Mercury [BKP01, BG03, BG01, PBG04, GMB05] is a generic and extensible monitoring system, built as part of the GridLab project. The latter aims in the development of a *Grid Application Toolkit* to help developers to build grid-aware applications [ADD+03]. Mercury is a grid-enhanced version of the GRM distributed monitor [PK00, BKPV01] of the P-GRADE graphical parallel program development environment. GRM is an instrumentation library for message-passing applications in traditional parallel environments (such as clusters and supercomputers.)

Overview Mercury has one local monitor per host (*producer*), a main monitor, and a monitoring service (*republishers*). Local monitors employ a set of sensors, implemented as loadable modules, to collect information about the local node, including host status, applications, etc., and send it to the main monitor. The latter coordinates local monitors according to requests received from the monitoring service and also serves requests from local (i.e., site-level) consumers.

The monitoring service is where external consumers submit their queries. Upon the receipt of such a query, the monitoring service validates it against the site policy and, if valid, instructs the main monitor to perform the query, which in turn coordinates the involved local monitors. Eventually, the monitoring service receives the results from the main monitor, transforms the platform-specific measurements to comparable values and finally forwards them to the consumer.

Mercury defines a custom producer-consumer protocol that supports multi-channel communication and uses eXternal Data Representation (XDR) for the encoding of events. In addition, a library and a special sensor are provided for the instrumentation of applications, so that a job can generate custom events which are sent to the sensor and read by the local monitor. Mercury also provides decision-making modules that inform actuators on adapting the monitoring process and steering applications.

The main monitor in Mercury can be run in many instances to form an arbitrary hierarchy. Combined with the support for events related to hosts, networks and applications, Mercury is classified as L3.G.

MonALISA (L3.G.S)

MonALISA (Monitoring Agents using a Large Integrated Services Architecture) [NLG+03] is a Jini based [Wal99], extensible monitoring framework for hosts and networks in large scale distributed systems. It can interface with locally available monitoring and batch queueing systems through the use of appropriate modules. The collected information is locally stored and made available to higher level services, including a GUI front-end for visualising the collected monitoring events.

Overview MonALISA is based on the Dynamic Distributed Services Architecture (DDSA) [NLB01] which includes one station server per site or facility within a grid, and a number of Jini lookup discovery services (i.e., equivalent to *registries*.) The latter can join and leave dynamically, while information can be replicated among discovery services of common groups.

A station server hosts, schedules, and restarts if necessary, a set of agent based services. Each service registers to a set of discovery services from where can be found from other services. The registration in Jini is lease based, meaning that it has to be periodically renewed, and includes contact information, event types

of interest and the code required to interact with a given service.

Each station server hosts a multi-threaded monitoring service, which collects data from locally available monitoring sources (e.g., SNMP, Ganglia, LSF, PBS, Hawkeye) using readily available modules. The collected data are locally stored and indexed in either an embedded or an external database, and provided on demand to clients (i.e., *consumer* services.)

A client, after discovering a service through the lookup service, downloading its code and instantiating a proxy, can submit real time and historical queries or subscribe for events of a given type. Custom information (i.e., not part of the schema) can be acquired by deploying a digitally signed agent filter to the remote host. In addition, non-Java clients can use a WSDL/SOAP binding.

Services and modules can be managed through an administration GUI, allowing an authenticated user to remotely configure what needs to be monitored. Also, MonALISA has a facility for periodically checking the availability of new versions and automatically updating any obsolete services.

Current applications of MonALISA are a GUI front-end featuring several forms of status visualisation (maps, histograms, tables, etc) and dynamic optimisation of network routing for the needs of a video conference application. Future plans include building higher level services for supporting job scheduling and data replication.

In conclusion, MonALISA provides a general-purpose and flexible framework, even though its deployment requires support for multicast (due to Jini). In terms of the taxonomy, external (to MonALISA) monitoring sources are *producers*, whereas monitoring services and other higher-level services serve as *republishers* by collecting data from producers and providing higher level abstractions, respectively. To this end, MonALISA is classified as L3.G.S meaning that custom-built republishers can be structured in a custom hierarchy; a variety of entities can be monitored; and the system is stackable.

R-GMA (L3.G.S)

Conceptually, R-GMA provides access to the information about resources across different Virtual Organisations, as if that was stored in a single RDBMS. R-GMA [CGM+03b] has been developed as part of the EU DataGrid project, and is being re-engineered in the EGEE project. R-GMA is a framework which combines grid monitoring and information services based on the relational model. That is,

R-GMA defines the GMA components and their interfaces, in relational terms.

Overview In R-GMA, producers were originally distinguished in five different classes, including *database producers* for static data stored in databases, and *stream producers* for dynamic data stored in memory resident circular buffers. The latest version of R-GMA [BCC+05] has primary producers, secondary producers (which correspond to GMA's producers and republishers), and on-demand producers. (On-demand producers are a lazy version of primary producers, which produce data only in response to a query.)

New producers announce their event types (relations, in R-GMA terminology) using an SQL "create table" query, offer them via an SQL "insert" statement, and "drop" their tables when they cease to exist. A *consumer* is defined as an SQL "select" query. For a component to act as either a consumer or a producer, it has to instantiate a remote object (agent) and invoke methods from the appropriate (consumer or producer) API.

The *global schema* includes a core set of relations, while new relations can be dynamically created and dropped by producers as previously described. *Republishers* are defined as one or more SQL queries that provide a relational view on data received by producers or other republishers.

The *registry* holds the relations and views provided by database producers, stream producers and republishers. A mediator uses the information available in the registry and cooperates with consumers to dynamically construct query plans for queries that cannot be satisfied by a single relation (i.e., involving "joins" from several producers.) The registry used to be centralised, but has recently been reimplemented as a distributed service that supports replication [BCC+05].

R-GMA is implemented in Java Servlets and its API is available for C++ and Java, while wrapper API implementations exist for C, Perl and Python. The R-GMA implementation is considered stable and is in use in production settings, including the LCG grid [LCG].

2.4.5 Other Related Work

Other related work is not considered in this chapter, because there is not insufficient information for it to be classified according to the taxonomy. For instance, the proposals [KLH+05, DL05] focus on large-scale grid information services but do not elaborate on important issues of monitoring (e.g., information is assumed

to be centrally located without explanation of how it is collected and kept up to date). These and other proposals (including peer-to-peer based ones) are discussed in the context of the proposed architecture, in Section 4.2.

For a more feature-oriented (as opposed to architectural) characterisation of grid-oriented performance analysis systems one can consult a white paper [GWB+04] by the "Automatic Performance Analysis: Real Tools" (APART) working group.

2.5 Scope of Deployment and Potential Scalability

Monitoring systems that are classified as level-three in the taxonomy have the largest scope of deployment, compared to level-zero and up to level-two systems. Large scope of deployment also implies more potential for scalability. In particular, the capability to extend a monitoring hierarchy in terms of number of nodes per level, and number of levels in the hierarchy, implies a higher potential of distribution of (monitoring and query) load in many nodes.

Level-three systems, or hierarchies of republishers, were described as systems that have sufficiently generic republishers that can be used in an arbitrarily structured monitoring hierarchy. We distinguish a certain case of level-three systems in which every republisher is responsible for all the information in its child nodes (as opposed to, for instance, republishers that are configured to collect only a subset of their child node's information, to maintain special, e.g. domain-specific, views).[4] This case is of primary interest because it satisfies the requirement for grid-wide resource discovery. Level-three systems in which republishers are responsible for all the resource information of their child nodes, will be referred to as hierarchies of aggregation.

2.6 Closing Remarks

This chapter provided background material on grid monitoring, including concepts and terminology, a detailed description of the steps involved in monitoring, and an overview of a reference grid monitoring architecture. A taxonomy of

[4] "Child node" is used in the sense of immediate descendant.

grid monitoring systems was also presented and used to classify a wide range of relevant work.

The chapter also discussed existing monitoring systems and proposals with respect to their capability to scale gracefully for large grids. Level-three systems were identified as most likely to scale well. The following chapter discusses and evaluates hierarchies of aggregation, a certain case of level-three systems, in terms of scalability.

An extended version of this chapter has been published in [ZS05].

Chapter 3

Performance of Monitoring Hierarchies of Aggregation

The previous chapter asserted, among others, that hierarchies of aggregation (i.e., a certain case of level-three systems as described in Section 2.5) are in principle more scalable than lower level systems, due to the distribution of load in many hosts. The aim of this chapter is to motivate the thesis by discussing the performance trade-offs in monitoring hierarchies of aggregation, and to provide a performance baseline against which the thesis proposal will be evaluated. Specifically, this chapter makes the following contributions:

- A discussion of the trade-offs involved in large-scale monitoring, between information freshness, network overhead (that is imposed by monitoring), and the response time to queries about grid resources. Two opposite choices for these trade-offs, *just-in-time evaluation* and *prefetching* (where just-in-time fetches resource information only on cache misses, while prefetching fetches resource information periodically and independently of query arrivals), as well as in-between, hybrid approaches are discussed. Prefetching is the basis for the present thesis, described in detail in Chapter 4.

- A model for an optimistic (i.e., best case) estimation of the performance of arbitrary configurations of hierarchies of aggregation that use just-in-time evaluation, in given Grid settings.

- An evaluation of alternative hierarchies of aggregation that use just-in-time evaluation for the monitoring of three Grids of different scale, using

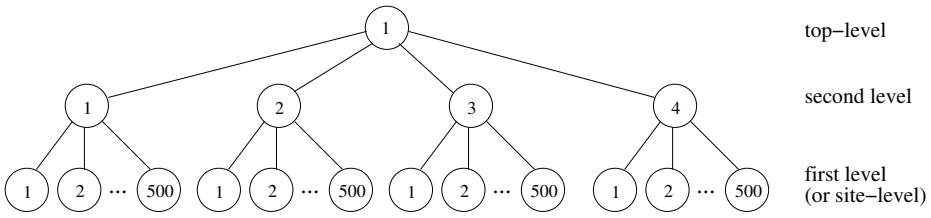

Figure 3.1: Illustration of the $1 \times 4 \times 500$ hierarchy.

the aforementioned performance model. The purpose of the evaluation is manifold:

- to motivate the thesis by demonstrating the scalability issues in just-in-time evaluation;
- to compare alternative hierarchies of aggregation in the same Grid setting;
- to compare the performance of hierarchies of the same depth (i.e., number of levels of a monitoring hierarchy) in Grids of different scale;
- to provide optimistic performance estimates for hierarchies of aggregation that use just-in-time evaluation; these estimates are used as a basis for a comparison against experimental results of the thesis proposal, in Chapter 6.

The chapter is structured as follows. Section 3.1 introduces concepts and terms with respect to hierarchies of aggregation. Section 3.2 discusses related work. Section 3.3 discusses the trade-offs between network overhead, information freshness and query response time, in hierarchies of aggregation. Section 3.4 presents a performance model for hierarchies of aggregation that use just-in-time evaluation, which is used in Section 3.5 to compare various configurations of such hierarchies for monitoring three Grids of different scale. Section 3.6 concludes the chapter.

3.1 Background

Existing grid monitoring and information services typically have a hierarchical (i.e., tree) structure. For instance, Figure 3.1 shows a three-level hierarchy of aggregation for monitoring 2000 grid sites.

This chapter refers to nodes depending on the level of the hierarchy at which they are located. Numbering levels is trivial for balanced hierarchies but ambiguous for unbalanced ones. In the present work, a node's level is determined as follows: (i) leaf nodes are at the first-level; (ii) a non-leaf node's level equals to the hierarchy's total number of levels (i.e., the tree's height; e.g., 3 and 4, respectively, for Figures 3.1 and 3.3 (page 64)) minus the number of edges in the path between the considered non-leaf node and the top-level node. For instance, nodes 2.2 and 3.2 in Figure 3.3 are at the second and third level, respectively.

In Figure 3.1, every node represents a host that runs a grid information service. Every first-level node is responsible for the resources that are hosted in the respective grid site. The second-level nodes collect and republish resource information from the respective first-level nodes, and so on, all the way up the hierarchy. Every node maintains a local cache, where it caches resource information that it receives from its child nodes (i.e., immediate descendants). A node may be configured to periodically collect all the information that is available from its child nodes, or may collect resource information only upon the arrival of a query. (These two approaches are discussed in detail in Section 3.3.)

Users (with the right permissions, if applicable) can query any of the nodes in the hierarchy. It is the responsibility of users to choose the node to which they submit a query, taking into account the scope of the query. For instance, if a user is looking for resources at a specific site, then the query should be submitted to the respective site-level node. For a query about resources in a specific region, e.g., North-West England, one would have to query the North-West node (or the UK node, which presumably would have a superset of the relevant information). This, however, requires that:

- the hierarchy of monitoring nodes is organised appropriately for the query at hand (in the above example, nodes have to be organised based on locality); and

- users have sufficient knowledge of the way a monitoring hierarchy is organised, to choose the appropriate node.

The present work assumes that, in most cases, at least one of the two aforementioned conditions is not true, and thus most user queries are submitted to the top-level node of a monitoring hierarchy.

Consider what happens when a user submits a grid-wide resource discovery

query at the top-level node of Figure 3.1. An example of a grid-wide query would be "list clusters with at least 16 nodes and MPI installed, ordered by site-level network capacity". The way a query is handled depends on the used configuration. The top-level node may have cached the results to that specific query, or may have a local copy of all the relevant resource information. In either case the query is resolved locally. Alternatively, when a cache miss occurs, the top-level node forwards the query to its child nodes, and waits to collect the results. Everyone of the child nodes follows the same process. The query may have to be forwarded all the way down to one or more site-level nodes. Eventually, every node responds to its parent, and the top-level node receives the results and responds to the user.

A key indicator for efficient information services is **query response time (QRT)**, that is, the time to return the results that match a given query. Also of interest are: the **network overhead** imposed by monitoring, and the level of **information freshness** that is delivered to users (i.e., to what extent the information is up-to-date). There is a trade-off between network overhead and information freshness. The more frequently resource information is updated, the higher the freshness of information and the imposed network overhead are.

In the present thesis, QRT is considered as the server-side costs of query resolution, that is, not including the client-side costs for submitting the query and receiving the query results. Thus, QRT is composed of network retrieval time and query processing time.

Network retrieval time (NRT) is the time to retrieve resource information over the network (assuming it is not available from the cache).

Query processing time (QPT) is the time to process the query once all the relevant information is locally available.

Unless a query is resolved locally, QRT is typically dominated by NRT (because CPU processing is cheaper than network access [Gra03]). Thus, many systems strive to minimise the number of cache misses. This can be done in various ways. For example, in many systems, the administrator can manually configure the duration for which data can reside in the cache (i.e., Time To Live, TTL).

3.2 Related Work

The performance of grid monitoring and information services has been the subject of several experimental studies. Some studies evaluate a single implementation [SWMY00, KD03b, KD03a], while others compare several systems [PJL+03, PJJ+04, ZFS03, ZFS05].

However, little work exists on modelling QRT. QPT is highly implementation-specific [PJL+03, PJJ+04] and NRT depends on network performance, and thus can vary significantly. (Especially taking into account that monitoring nodes are typically connected via wide area network links, which can be highly unstable and unpredictable.) As as result of these difficulties, the predictability of QRT decreases as the complexity of a monitoring hierarchy increases [KD03a]. Keung *et al.* [KD03b] consider various prediction models for the query throughput of a Globus republisher (i.e., GIIS server) that caches information from a single producer (i.e., GRIS server). In contrast, the model presented in Section 3.4 is intended for hierarchies of aggregation of arbitrary complexity (e.g., not only two levels), is implementation-agnostic, and yields an optimistic estimate of QRT. Specifically, the model does not attempt to capture variations in QPT performance. Nevertheless, QPT in hierarchies of aggregation that use just-in-time evaluation tends to be insignificant compared to NRT.[1]

3.3 Monitoring Trade-offs and Scalability

As explained in Section 3.1, every node in a monitoring hierarchy maintains a local cache. This section distinguishes approaches to configurations of hierarchies of aggregation based on the mode used for the evaluation of queries. Specifically, the distinction is based on the times at which every node's cache is updated:

- in prefetching, the cache is updated every TTL time units (where TTL is a constant, typically manually set by the service administrator);
- in just-in-time evaluation, the cache is updated only upon the arrival of queries that cause cache misses;
- in hybrid approaches, both prefetching and just-in-time evaluation are used for different kinds of information.

[1] This may not hold for sufficiently complex queries, but we assume that monitoring services will impose certain upper bounds in query complexity.

The discussion is focused on scalability in terms of (i) number of monitored resources and (ii) number of queries.

3.3.1 Prefetching

In prefetching, resource information is updated every TTL time units, regardless of when queries arrive. Queries are always resolved against local, previously collected resource information, regardless of whether that is up to date. Whether query results are actually fresh, depends on how often the information is updated (i.e., the TTL setting).

The trade-offs involved in prefetching are as follows. TTL is inversely proportional to network overhead and information freshness. On the one extreme, high TTL means infrequent updates, thus low network overhead but also low information freshness. On the other extreme, low TTL means frequent updates, thus high information freshness and network overhead.

The advantages of prefetching are: (i) low, predictable QRT, because all queries are run against locally cached information; and (ii) good scalability with respect to the number of queries. On the downside, using low-medium TTL imposes medium-high network overhead, which is unnecessary when the number of queries is small. Also, prefetching does not scale well for a large number of monitored resources in terms of network overhead. This is often addressed using high TTL, at the cost of low information freshness.

3.3.2 Just-In-Time Evaluation

Another configuration is to fetch resource information only as needed: on a cache miss. In just-in-time evaluation, the local cache keeps the result of every query for TTL time units. Thus, despite the name "just-in-time", on cache hits the results are as fresh as the data in the local cache.

As a result, information freshness is inversely proportional to TTL, as is the case in prefetching. In just-in-time evaluation, network overhead is inversely proportional to TTL and proportional to cache miss ratio. Also TTL is inversely proportional to cache miss ratio (e.g., when TTL=0, cache miss ratio=100%). In general, the probability of cache hits largely depends on TTL and the extent that closely-submitted queries are identical. On the one hand, if only a few cache misses occur (e.g., because most queries over an interval that is less than TTL

are identical) there is little network overhead. On the other hand, if most queries cause a cache miss (e.g., because they are unique and/or TTL is small), many network transfers have to be carried out.

The advantage of just-in-time evaluation is that the imposed network overhead is the least necessary: no information is fetched unless it is actually going to be used. On the downside, just-in-time evaluation has unpredictable and potentially high QRT, especially for large hierarchies (in terms of number of nodes). The problem is that any network transfer that crosses administrative domains can, in principle, take an unbounded amount of time to complete. This is because cross-domain network links are shared: there is no dedicated bandwidth capacity to guarantee that a given transfer will be completed within a certain time interval. Moreover, just-in-time evaluation does not scale well for a large number of queries (more precisely, cache misses) in terms of network overhead.

3.3.3 Hybrid Approaches

Hybrid approaches of hierarchies of aggregation combine both prefetching and just-in-time evaluation. An example would be a system in which nodes prefetch part of the resource information for which they are responsible, in the hope to reduce cache misses. As with just-in-time evaluation, hybrid approaches also suffer by the fact that there can be no guarantees for the completion time of cross-administrative network transfers. Thus, QRT can be arbitrarily long when a cache miss occurs.

3.3.4 Thesis Context

Prefetching-based monitoring is the basis of the architectural proposal in Chapter 4. The remainder of the present chapter focuses on the evaluation of hierarchies of aggregation that use just-in-time evaluation, to motivate the thesis (by demonstration of their scalability issues) and to provide a performance baseline (for the thesis evaluation in Chapter 6).

In the remainder of the thesis, hierarchies of aggregation that use just-in-time evaluation will be abbreviated as JITHAs.

3.4 A Performance Model of Just-In-Time Hierarchies of Aggregation

This section describes a model for the assessment of the performance behaviour of just-in-time hierarchies of aggregation (JITHAs). Specifically, the model gives optimistic estimates of query response time and network overhead of a given hierarchy of aggregation in certain settings.

3.4.1 Assumptions

The model assumes grid-wide or large-scope queries with sorted results, meaning that queries have to be resolved against information about resources throughout the grid and results ordered by some property. As such, the queries are submitted to the top-level node of a monitoring hierarchy, as opposed to the site-level information service of the requester. In addition, the following assumptions are made to produce optimistic estimates of QRT, and simplify the model.

1. The resources that match a given query are evenly distributed across sites. For instance, when a query matches 1000 resources in a 500 site grid, every site-level monitoring node would be responsible for 2 matches. This assumption results in more optimistic estimates of QRT because the query processing load at the first-level is evenly distributed across all site-level nodes.

2. When a cache hit/miss occurs at a node of the Nth level, the same holds for all nodes at that level. Normally, at a level of n nodes, there are 2^n combinations of cache hits/misses that can occur at that level. In the best case, all nodes have a cache hit; in the worst case, all nodes have a cache miss; in all other cases only some of the nodes have a cache hit (hereafter referred as the typical cases). The typical cases' QRT tends to be closer to that of the worst case, because QRT overall is determined by the slowest node that is involved in a query. The model yields optimistic QRT estimates, by considering only the best and worst cases (and not the typical cases which are typically closer to the worst).

3. When a cache miss occurs, a node has to forward the query to its child nodes and eventually collect the responses. There is a potential for network

contention due to overlapping sets of parallel transfers from closely submitted queries that cause cache misses (e.g., when the collection of results of a newly submitted query overlaps with that of a pending query). The model does not account for such network contention.

Also, query processing time (recall from Section 3.1 that QRT is composed of QPT and NRT) is set as a fixed input parameter. This choice is made because QPT is highly implementation-specific and tends to be small compared to NRT (that is, when network transfers actually occur). Using a fixed and low QPT yields optimistic QRT estimates, as it does not take into account that (i) QPT increases with the amount of queried data (e.g., higher-level nodes would typically have higher QPT) [ZFS05]; and (ii) a node's load is increasing with the number of child nodes it interacts with. Consider an example for the second point. A node that has 2000 child nodes is expected to have higher load compared to a node that has 2 child nodes, which in turn have 1000 nodes each, because the former has to interact with 2000 (instead of 2) child nodes. Using a fixed, low QPT, however, is sufficient for the model's purpose, i.e., an optimistic estimation of QRT.

3.4.2 Description of the Model

The model has the following input parameters:

- the number of grid sites and that of clusters per site (*nsites, nclusters*);
- the number of queries submitted over a time interval (*nqueries*);
- the percentage of queries submitted to a node that can be served directly from its cache (*cache hit ratio*);
- the percentage of resources (in this case clusters) throughout the Grid that match a given query (*query selectivity*);
- a fixed value for QPT;
- a set of network performance measurements (described below);
- the average size of the description of a single query match (*match size*). In relational database terms, match size is the average record size of a

query's result set. This depends on the compactness (or lack of) of the used representation, the amount of information that a query requests, and the expressiveness of the query language that is used.

- For instance, a query that only requests the identifiers of matched clusters is likely to have a smaller match size than one that requests a list of available software in matched clusters.
- Also, match size can be significantly smaller when an information service uses, say, a binary representation as opposed to XML.
- Match size can also be affected by query expressiveness in that in restricted query languages one may have to submit several queries (and retrieve and process intermediate result sets) to retrieve the final result set, as opposed to only one query (and one result set). For instance, to express a query that includes a join in a language without support for joins, one has to submit a part of the query, followed by a second query based on the result set of the first query, and so on.

For the estimation of network overhead and QRT for a given query, there are two possibilities: a cache hit or a cache miss. A cache hit is straightforward, as QRT equals QPT and there is no network overhead. On a cache miss, a node has to forward the query to its child nodes, and eventually collect the results. The child nodes may in turn have a cache miss in which case they have to carry out the same process, until there is a cache hit. The problem is simplified considerably due to assumption 2 in Section 3.4.1 (namely, when a cache hit/miss occurs at a node, the same holds for all nodes at that level). Given this assumption, the possible cases (of where a cache hit can occur) are reduced to at most the number of levels of a monitoring hierarchy.

Consider an example assuming the hierarchy in Figure 3.2. Recall that based on the assumptions at the previous section, a query's matches are equally distributed across all sites. For the purpose of the example, let the description of the matched resources at every site be N bytes. When a cache hit occurs at the first level (i.e., when cache misses occur at the top, third and second levels):

1. every second level node has to perform 500 transfers, N bytes each;
2. node 3.1 has to perform two transfers from nodes 2.1 and 2.2, 500×N bytes each; the same holds for node 3.2, collecting from nodes 2.3 and 2.4;

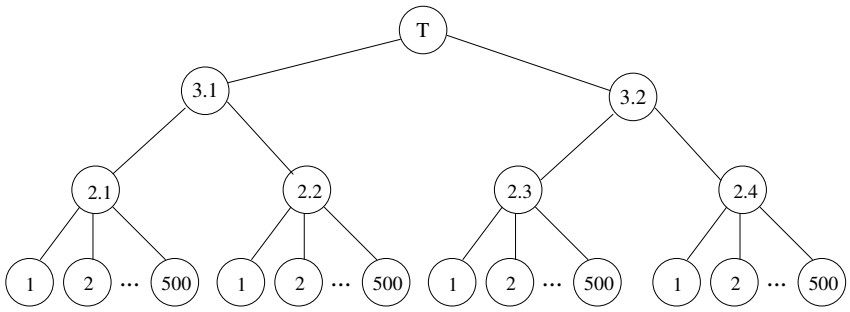

Figure 3.2: A symmetric, 4-level hierarchy for monitoring 2000 sites.

3. the top-level node has to perform two transfers from nodes 3.1 and 3.2, 1000×N bytes each.

When the cache hit occurs at the second level, only steps 2 and 3 apply. Similarly, when the cache hit occurs at the third level, only step 3 applies.

The performance model has two phases.

- The first phase, for a given hierarchy and set of settings, generates the combinations of <number of transfers, transfer size> for each level where a cache hit may occur. Note that a cache hit at level K implies that (i) there are cache misses at the top-level node down to and including the K+1 level nodes; (ii) the query is never seen by nodes at levels 1 up to K − 1.

  In the above example, the list of combinations would be <500, N> (i.e., 500 parallel transfers of N bytes each), <2, 500×N> (i.e., 2 parallel transfers of 500×N each), <2, 1000×N> (i.e., 2 parallel transfers of 1000×N each), where <M, L> denotes that M network transfers occur in parallel, and each one of those transfers has L bytes length. For instance, <500, N> indicates the time required by node 2.1 (and all the second-level nodes) in Figure 3.2 to collect query results from its child nodes.

- The second phase of the model, which produces the actual performance estimates for QRT and network overhead, has to be given values that indicate the time that is required to perform M transfers of L bytes length for all the combinations that were given in the first phase. These input values are hereafter referred as the network performance measurements.

The extent that the model's estimates are reasonable depends on these measurements. The measurements should be indicative of the network performance of

the environment in which a monitoring hierarchy of aggregation is intended to be deployed. This is further discussed in Section 3.5.3.

At the second phase, given the network performance measurements, the model calculates QRT and network overhead for every different level at which a cache hit may occur. Based on these it then calculates the average QRT and total network overhead for a given number of queries.

On a cache hit, QRT equals to QPT and there is no network overhead. On a cache miss, a node has to forward the query to its child nodes, and eventually collect the results. Thus, on a cache miss, QRT is QPT plus NRT plus the child nodes' largest QRT. Given the above, the query response time $\mathrm{QRT}_{i,j}$ of node j at level i is recursively calculated as follows:

$$QRT_{i,j} = \begin{cases} \text{QPT, on a cache hit} \\ \text{QPT} + \max_{\mathrm{c \in C_{ij}}}(\mathrm{NRT_{c,ij}} + \mathrm{QRT_c}), \text{ otherwise} \end{cases} \tag{3.1}$$

where

- C_{ij} is the set of child nodes of node j at level i, and c denotes the subscripts of such a child node;
- $\mathrm{NRT}_{c,ij}$ is the time needed to transfer a query's result set from child node c to its parent j at level i (based on the network performance measurements).

The network overhead $v_{i,j}$ of node j at level i is:

$$v_{i,j} = \begin{cases} 0, \text{ on a cache hit} \\ \sum_{c \in C_{ij}} r_c \times \text{query selectivity} \times \text{match size, otherwise} \end{cases} \tag{3.2}$$

where r_c is the number of resources the node c is responsible for monitoring. The r_c of a given node is the sum of the r_c of all its child nodes (i.e., immediate descendants); the bottom level nodes have $r_c = nclusters$. (In other words, the r_c of a node c is the sum of the number of resources that are monitored by the first-level nodes of the tree that is rooted at c.)

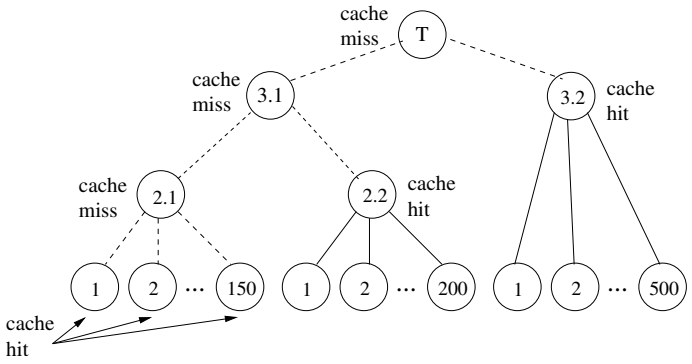

Figure 3.3: An example of query resolution with just-in-time evaluation; dashed (resp. solid) lines denote network connections that are used (resp. not used) for resolving the query.

3.5 Evaluation of Various Just-In-Time Hierarchies of Aggregation

The purpose of the evaluation is to (i) compare alternative just-in-time hierarchies of aggregation (JITHAs) in the same Grid setting; (ii) compare the scalability of JITHAs of the same depth in Grids of different scale; and (iii) provide a performance baseline of JITHAs.

3.5.1 Preliminaries

This section presents further assumptions that are made to ensure that the model yields optimistic QRT. The assumptions aim to eliminate load imbalance among hosts at the same level of a hierarchy.

To explain load imbalance, consider the example of query resolution in Figure 3.3. A query is submitted to the top-level node, where a cache miss occurs. The node sends the query to its child nodes, 3.1 and 3.2, and so on, until a hit occurs at every branch. As a result, the following transfers have to be carried out:

- node 2.1 has to perform 150 transfers concerning the resources that match the given query at 150 grid sites;
- node 3.1 has to perform 2 network transfers concerning the matches from 2.1 and 2.2, which together represent 350 sites;

- the top-level node has to perform 2 network transfers concerning the matches from all 850 sites;

Load imbalance refers to a situation in which at least one of the following is true:

- nodes at the same level of the hierarchy (i.e., siblings) are responsible for monitoring a significantly different number of resources (e.g., in the example, nodes 3.1 and 3.2 are responsible, respectively, for 350 and 500 sites);
- the number of resource matches in a set of sibling nodes is not evenly distributed across those nodes.

Load imbalance is important because it limits the performance benefits of data distribution and can thus significantly increase QRT. From the example, one can notice the following causes of load imbalance:

- the hierarchy of monitoring nodes in Figure 3.3 is not symmetrical (as opposed to the one in Figure 3.1); thus, node 3.2 would typically take longer to reply to a query compared to node 3.1, as the former is responsible for a larger number of grid sites and has to contact directly 500 nodes (as opposed to only 2 for node 3.1);
- the cache of nodes at the same level of the hierarchy may not be equally up to date, e.g., node 2.1 has a cache miss whereas node 2.2 has a cache hit.

The presented model is intended to estimate best-case QRT, thus the causes of load imbalance must be eliminated. On this basis, the following assumptions are made.

1. The hierarchy of monitoring nodes is symmetric, i.e., all nodes at a given level have the same number of child nodes (e.g., Figure 3.1).
2. The site-level (i.e., lowest-level) monitoring nodes are responsible for the same number of resources; i.e., every grid site hosts the same number of resources.

3.5.2 Settings

Due to the focus on scalability, the evaluation concerns grids much larger than today's deployments of up to a few tens of grid sites. Specifically, the evaluation

two levels	three levels	four levels	five levels
500 sites (496 in 5 cases)			
1×500	$1 \times 2 \times 250$	$1 \times 2 \times 4 \times 62$	
	$1 \times 4 \times 125$	$1 \times 2 \times 8 \times 31$	
	$1 \times 8 \times 62$	$1 \times 4 \times 4 \times 31$	
	$1 \times 16 \times 31$		
2000 sites (1984 in 5 cases)			
1×2000	$1 \times 2 \times 1000$	$1 \times 2 \times 4 \times 250$	$1 \times 2 \times 4 \times 8 \times 31$
	$1 \times 4 \times 500$	$1 \times 2 \times 8 \times 125$	
	$1 \times 8 \times 250$	$1 \times 4 \times 4 \times 125$	
	$1 \times 16 \times 125$	$1 \times 2 \times 16 \times 62$	
	$1 \times 32 \times 62$	$1 \times 4 \times 8 \times 62$	
		$1 \times 4 \times 16 \times 31$	
5000 sites (4992 in 10 cases)			
1×5000	$1 \times 2 \times 2500$	$1 \times 2 \times 2 \times 1250$	$1 \times 2 \times 4 \times 8 \times 78$
	$1 \times 4 \times 1250$	$1 \times 2 \times 4 \times 625$	$1 \times 2 \times 4 \times 16 \times 39$
	$1 \times 8 \times 625$	$1 \times 2 \times 8 \times 312$	
	$1 \times 16 \times 312$	$1 \times 4 \times 8 \times 156$	
	$1 \times 32 \times 156$	$1 \times 4 \times 16 \times 78$	
	$1 \times 64 \times 78$	$1 \times 4 \times 32 \times 39$	
	$1 \times 128 \times 39$		

Table 3.1: The evaluated hierarchies of aggregation.

concerns grid settings of 500, 2000, and 5000 grid sites, where every site is assumed to host 5 clusters. Table 3.1 lists the evaluated hierarchies of aggregation, for every considered grid size. The $A \times B$ notation denotes that each of the A nodes has B child nodes.

We measure QRT and network overhead for 2000 queries. (The exact timing of the queries is not significant, as the model does not account for network or CPU contention.) Experiments were performed for two values of query selectivity, i.e., the number of clusters that match a given query: 10% and 40%. (That is, an experiment has either 10% or 40% query selectivity throughout its duration.)

For cache hit ratio, experiments were performed for 10% and 20% at every level of a monitoring hierarchy (that is, either 10% or 20% in any given experiment), and 100% at the site-level nodes. For instance, for cache hit = 10% in the hierarchy $1 \times 4 \times 500$, there is a 10% probability of a cache hit at the top-level and another 10% in the 4 second-level nodes; if there is a cache miss at both levels, then a cache hit is assumed at the site-level nodes. A cache hit ratio of 0% effectively means no caching (i.e., TTL=0) and results in ideal information

freshness. Thus, the above settings of 10 and 20% cache hit imply a rather low TTL setting, which would typically result in fairly high information freshness (assuming that the average time between resource information changes is larger than TTL).

Various settings are considered for implementation-specific parameters. QPT is set to 25 and 100 ms for idle and lightly loaded conditions, respectively. These settings are intentionally low, compared to experimental results of grid information systems. For instance, Figure 29(a) in [ZFS05] shows that the QPT of a republisher that monitors 50 hosts ranges from about 40 ms up to 1 second (depending on the system). The same figures for a republisher that monitors 400 hosts range from 200 ms up to 8 seconds.

Match size is set to 0.1, 1 and 2 KB, for compact and verbose representations (e.g., binary vs XML). Note that even the 2 KB setting is optimistic for systems with query languages of limited expressiveness. A complex query in LDAP or XPath, may require more than one interaction, thus dramatically increasing network overhead (and QRT) [PJJ+04].

3.5.3 Network Measurements

A custom client/server was implemented to record network performance measurements (i.e., the time that is needed to perform M parallel transfers of a given size). Specifically, the client represents a parent node, and threads in server instances represent the child nodes of that parent. The server was run in 17 hosts. Every server was configured to fork 300 threads that listen to consecutive TCP ports. (That is, 5100 threads, which is more than what is required for the largest considered setting of 5000 sites.) Every thread accepts messages that indicate the length of the requested response, and sends back a response of that length. (The content of the response is irrelevant, as long as the response is of the requested length.)

For every combination of M transfers of L bytes each, the client (run in a dedicated host) contacts M ports requesting L bytes, with a maximum of 5000 concurrent connections.[2] For every combination, 15 measurements are taken, to compensate for the fact that the available network bandwidth at any given time may vary significantly (i.e., instantaneous bandwidth may differ significantly

[2]This requires to increase the default maximum number of open file descriptors, using the `ulimit` command in Linux.

from average bandwidth). The client interleaves the requests to the servers to reduce the effect of hosting 300 server threads per host. Specifically, every request at a host is followed by a request at all the other hosts. The client discards downloaded data as soon as they are received (as host load is not the subject of the measurements).

The measurements were taken in the 100 Mbps local area network of the School of Computer Science, at the University of Manchester. As this is a shared LAN, any network measurements can vary considerably, due to external factors that affect current network load. To reduce the effect of such variation, the above measurement process was performed for 16 distinct evenings, in the period from February 28th until March 19th, 2007, for each one of the 249 <M, L> combinations. Furthermore, every set of transfers was subject to a 10 second time-out. This resulted in a total of 59760 measurements (249 combinations times 16 evenings, times 15 measurements per combination per evening, or $15 \times 16 = 240$ measurements for every combination), and 59471 measurements after excluding time-outs.

In the second phase of the model, the time that is used for every <M, L> combination is the average of the 240 measurements for that combination.[3] (Appendix A lists the average duration of measurements for every <M, L> combination.)

3.5.4 Validation

Figure 3.4 shows a transcript of operation of the performance model implementation, when applied to the hierarchy $1 \times 2 \times 4 \times 8 \times 31$ with settings of 0.1 KB match size, 10% match ratio, 25 ms QPT and 5 clusters per grid site. The figure shows the estimated network overhead and QRT, for every level of the hierarchy at which a cache hit may occur. With respect to network overhead, consider case (v), when a cache hit occurs at the first level:

1. each one of the 8 second level nodes collect from every one of their respective 31 child nodes a result set of 0.05 KB (determined as 5 clusters per site-level node times 10% match ratio times 0.1 KB match size), in 74 ms;

2. each one of the 4 third level nodes collect from every one of their respective 8

[3] To be precise, up to 240 measurements, depending on whether any time-outs occurred during the measurements for that combination.

(i) cache hit at the top level
(no transfers occur)
network overhead: 0.00 KB; qrt: 25 ms

(ii) cache hit at the fourth level
4 level –> 5 level: 2 transfer(s) of 49.60 Kbyte messages in 33 ms (33)
network overhead: 99.20 KB; qrt: 83 ms

(iii) cache hit at the third level
3 level –> 4 level: 4 transfer(s) of 12.40 Kbyte messages in 36 ms (36)
4 level –> 5 level: 2 transfer(s) of 49.60 Kbyte messages in 33 ms (69)
network overhead: 99.20 KB; qrt: 144 ms

(iv) cache hit at the second level
2 level –> 3 level: 8 transfer(s) of 1.55 Kbyte messages in 220 ms (220)
3 level –> 4 level: 4 transfer(s) of 12.40 Kbyte messages in 36 ms (256)
4 level –> 5 level: 2 transfer(s) of 49.60 Kbyte messages in 33 ms (289)
network overhead: 99.20 KB; qrt: 389 ms

(v) cache hit at the first level
1 level –> 2 level: 31 transfer(s) of 0.05 Kbyte messages in 74 ms (74)
2 level –> 3 level: 8 transfer(s) of 1.55 Kbyte messages in 220 ms (294)
3 level –> 4 level: 4 transfer(s) of 12.40 Kbyte messages in 36 ms (330)
4 level –> 5 level: 2 transfer(s) of 49.60 Kbyte messages in 33 ms (363)
network overhead: 99.20 KB; qrt: 488 ms

Figure 3.4: Estimation of QRT and network overhead of the hierarchy $1 \times 2 \times 4 \times 8 \times 31$, based on the level at which the cache hit occurs.

child nodes a result set of 1.55 KB (0.05 KB times 31 result sets), in 220 ms;

3. each one of the 2 fourth level nodes collect from every one of their respective 4 child nodes a result set of 12.4 KB (1.55 KB times 8 result sets), in an average of 36 ms;

4. the fifth (top) level node collects from every one of its 2 child nodes a result set of 49.6 KB (12.4 KB times 4 result sets), in 33 ms; thus the network overhead is 99.2 (49.6 times 2 result sets).

With respect to QRT in the considered example, Figure 3.4 lists:

- the time required to perform a set of transfers of a given size (e.g., in case (ii) two transfers of 49.6 KB require 33 ms);

number of levels in JITHA	best QRT (in ms) of JITHAs with N levels in		rel. diff. between best and worst QRT in	
	LDIS	MDIS	LDIS	MDIS
two	581	1081	n/a	n/a
three	189	898	2.41	0.41
four	184	890	2.61	0.51
five	276	1071	n/a	n/a

Table 3.2: The second and third columns list the best QRT of JITHAs of a certain number of levels, for the least and most data intensive settings (denoted as LDIS and MDIS); the fourth and fifth columns show the relative difference of QRT of the best and worst JITHAs of a certain number of levels, in the least and most data intensive settings. All results are for the baseline setting (2000 sites and 20% cache hit ratio).

- the cumulative time of all network transfers up to a certain level, in parentheses;
- the total QRT, which is the cumulative time of all network transfers (NRT) at the top-level node, plus QPT times the number of levels involved in resolving a query.

For instance, in case (v), the total QRT, 488 ms, is calculated as the cumulative network transfers time at the top-level node (363 ms) plus QPT (25 ms) times 5 levels.

3.5.5 Results and Discussion

Query response time

This section makes some observations that hold for most of the considered settings but, for brevity, only a subset of the results are presented here.[4] Table 3.2 presents the QRT results for monitoring 2000 sites, and with 20% cache hit ratio. The results are for the least (resp. most) data intensive settings: 0.1 KB (resp. 2 KB) resource match size; 10% (resp. 40%) query selectivity), and assuming QPT=25 ms.

QRT is increasing with the amount of data that has to be collected on cache misses. This is evident by comparing the QRT of JITHAs with the same

[4]The full results are in Appendix B.

number of levels, when evaluated in the least and most data intensive settings (second and third columns, respectively, in Table 3.2). For instance, the best QRT among three-level JITHAs is 189 ms in the least data intensive setting, and is increased to 898 ms in the most data intensive setting.

The QRT performance of JITHAs with the same number of levels can vary significantly. In most of the considered combinations of settings and number of sites, there is at least a 0.5 relative difference between the best and worst QRT of JITHAs with the same number of levels, when evaluated at exactly the same setting. In Table 3.2, the third (resp. fourth) column shows the relative difference between the best and worst QRT in the least (resp. most) data intensive setting.[5] This observation underlines the importance of choosing the right hierarchy when just-in-time evaluation is used.

Deep hierarchies tend to have better QRT performance, up to a certain number of levels. Consider the best QRTs in either the least or the most data intensive settings in Table 3.2. The best three-level JITHA is faster than the best two-level one, and the best four-level JITHA is faster than the best three- and two-level ones. However, the best five-level JITHA appears to have worst QRT than the best two- and three-level JITHAs.

The same holds when comparing the range of worst QRTs of JITHAs of every level:

- 970-6000 ms in two-level hierarchies;
- 1230-3000 ms in three-level hierarchies;
- 570-2600 ms in four-level hierarchies;
- 580-2800 ms in five-level hierarchies;

This suggests that after a certain number of levels (in this case 5), the accumulated latency (due to many intermediate nodes) exceeds the benefits of sharing the query load across many intermediate nodes.

[5]The relative difference is calculated as $\frac{(max-min)}{min}$. The table does not show relative difference for two- and five-level JITHAs, because only one JITHA is considered for both, in the baseline setting.

Network overhead

By network overhead we refer to the amount of data, if any, that the top-level node has to retrieve to process a query (i.e., using Equation 3.2, page 63, in which $node_{i,j}$ is the top-level node). As such, the network overhead is not affected by QPT and the number of levels in the monitoring hierarchy, so we discuss results only for different settings of number of grid sites, match size, and query selectivity.

Figure 3.5 shows the network overhead (y axis in log scale) against the number of submitted queries (x axis), for the evaluated JITHAs, when cache hit is set to 20%. The two plots in the figure are for query selectivity of 10% (top) and 40% (bottom). Specifically, the imposed overhead varies from 39 up to 7790 MB for 10% query selectivity (top plot in Figure 3.5), and from 155 to 31161 MB for 40% selectivity (bottom plot in the same figure). Network overhead is obviously higher for 10% cache hit ratio (not shown here). In this case, the overhead ranges from 43 to 8673 Mb for 10% query selectivity, and from 172 to 34690 MB for 40% selectivity.

Overall, the results demonstrate that the network overhead that is imposed as a result of monitoring can vary greatly across different Grid settings, based on the number of monitored resources, and grows in proportion to the number of cache misses, query selectivity, and match size. The considered network overhead is imposed as a result of 2000 queries, but no assumption was made about the time interval within which these queries are spread. The imposed network overhead may be acceptable if 2000 queries is a daily workload, but not if it is a hourly workload. Either way, network overhead is important because monitoring is a non-stop activity and the cost of traffic in wide-area links is non-trivial.

Validation

The results on network overhead seem plausible, as the reported values clearly grow in proportion to match size, number of resources, query selectivity, and cache hit ratio.

It is not obvious, however, how the QRT results can be assessed in terms of correctness. Part of the problem is the lack of relevant experimental data to compare against. All the experimental evaluation studies we are aware of are concerned with two-level configurations, as opposed to multi-level hierarchies of monitoring nodes. On the other hand, it is quite evident that the QRT results that were reported in this section, are very optimistic. Consider, for instance,

the results of an experimental study, in which three different implementations of republishers are configured to collect information from 1 up to 1000 producers, using prefetching (Figure 18 in [ZFS03]), and accept queries about a subset of that information. In the cited experiment, the QRT is in the order of 1 second when monitoring 100 child nodes; between 6 to 20 seconds when monitoring 400 child nodes, and nearly 60 seconds for monitoring 1000 child nodes. In a more recent study (Figure 25 in [ZFS05]), the same systems' republishers are shown to have QRT in the range 1 to nearly 2 seconds when monitoring 100 child nodes; 4-6.5 seconds when monitoring 400 child nodes, and 6 to nearly 9 seconds when monitoring 600 child nodes. Note that the cited results are for two-level hierarchies that use prefetching. Compared to the results in [ZFS03, ZFS05], it becomes evident that the model's results are optimistic, as for instance, its estimates for QRT are much smaller (up to 6 seconds) for significantly larger numbers of monitored nodes.

Discussion

The present evaluation motivates the thesis by demonstration of the unstable and potentially large query response times of JITHAs, in which QRT depends on network performance, which in turn is sensitive to the number of monitored resources, query selectivity, and match size. Even though the model's QRT estimates are not significantly high (the worst being 6 seconds), it should be stressed that the estimates are highly optimistic due to the model's assumptions and the use of LAN measurements. Compared to the presented results, the experience from [ZFS03, ZFS05] suggests that QRT can be significantly larger, even when monitoring a smaller number of nodes (compared to the number of nodes considered in the present study). We conjecture that a JITHA for monitoring a grid of thousands of sites in realistic conditions (i.e., non-dedicated nodes that are interconnected via shared WAN links) would easily have average-case query response times in the order of several seconds and occasionally tens of seconds.

The model's QRT estimates, being highly optimistic, are most useful as a baseline. (In fact, the best QRT results in this chapter are used as a baseline for the evaluation of the thesis proposal in Chapter 6.) The model is also useful for the study of performance trends, and the comparison of different hierarchies of aggregation. An interesting observation of the present study, that deep hierarchies tend to have better QRT performance, confirms a similar finding in [ZFS05].

3.6 Closing Remarks

This chapter discussed the trade-offs involved in monitoring hierarchies of aggregation. A performance model was presented and used for the evaluation of various two- and up to five-level just-in-time hierarchies of aggregation (JITHAs), for monitoring Grids of 500, 2000 and 5000 sites. The model is intended to yield optimistic and implementation-agnostic estimates of query response time (QRT) and network overhead. From the evaluation, the following general conclusions can be drawn with respect to QRT:

- deep JITHAs appear to scale rather well in the examined (ideal) conditions, although in certain cases QRT can be more than 5 seconds; in any case, QRT in JITHAs can be arbitrarily large due to lack of guarantees for bounded network transfer times in cross-administrative wide area networks.

- in JITHAs, the choice of hierarchy for a given Grid setting can have a significant effect in QRT performance, even when the choice is limited among hierarchies of the same depth;

- narrow and deep JITHAs tend to be more scalable up to a certain level, because intermediate nodes have lower load due to interacting directly with few child nodes.

Regarding the network overhead that is imposed by monitoring, the results show that it can vary greatly across different Grid settings, depending on the number of monitored resources, and that it grows in proportion to the number of cache misses, query selectivity, and query match size. The network overhead of monitoring hierarchies of aggregation can be significant (as high as several tens of gigabytes, for 2000 queries, in the considered settings).

Overall, hierarchies of aggregation with just-in-time evaluation or hybrid approaches cannot be guaranteed to achieve consistently low QRT (unless one reserves sufficient bandwidth across administrative domains for exclusive use by grid information services). Prefetching on the other hand, does well with QRT but does not scale well in terms of imposed network overhead, if information freshness is to be kept high. The next chapter proposes a more flexible approach, based on prefetching, that allows the monitoring trade-offs to be determined on a site-level basis, instead of globally.

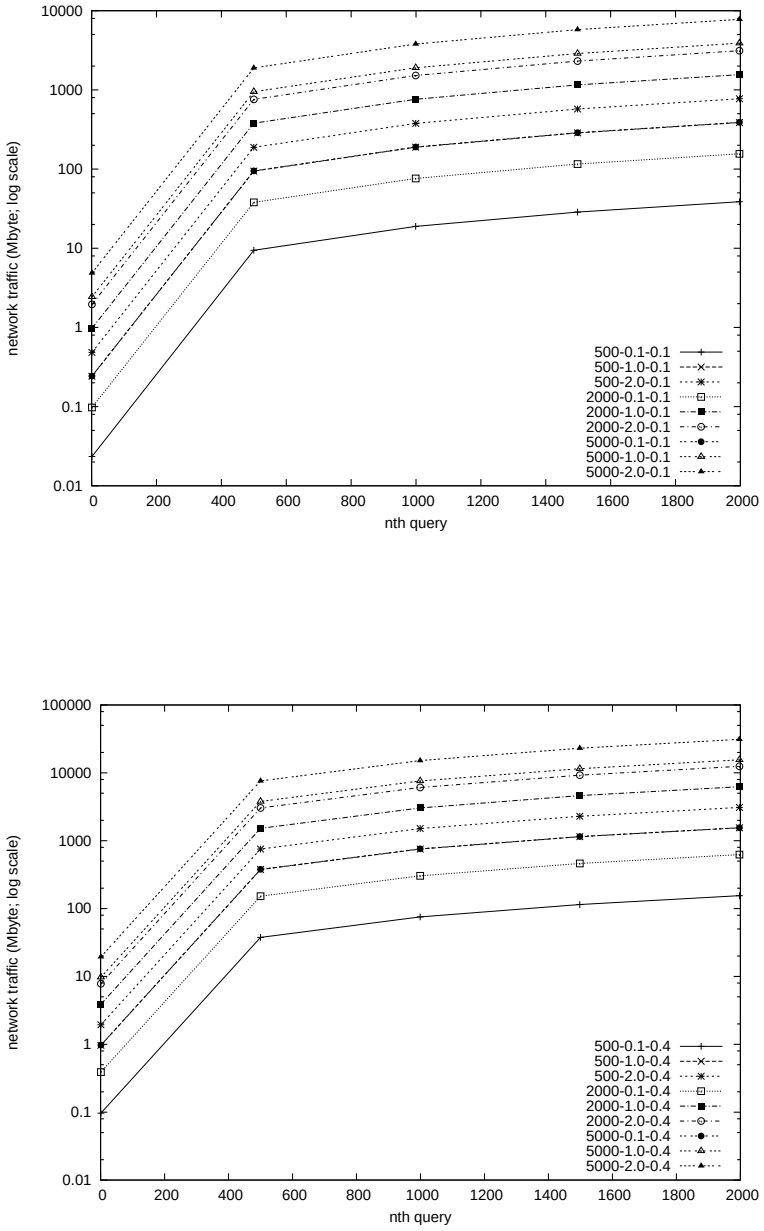

Figure 3.5: Network overhead vs number of queries run for 20% cache hit ratio (top: 10% query selectivity; bottom: 40% query selectivity); in the legend, 500-0.1-0.1 refers to 500-site Grid, 0.1 KB match size and 10% query selectivity; y-axis is in log scale.

Chapter 4

An Architecture for Grid Monitoring Using Importance-Aware Prefetching

The previous chapter (i) discussed the trade-offs involved in large-scale grid monitoring, with hierarchies of aggregation that are configured to perform prefetching or just-in-time evaluation, and (ii) evaluated many different configurations of just-in-time hierarchies of aggregation (JITHAs) in various settings. The present thesis is motivated by two observations:

- hierarchies of aggregation are configured to maintain a fixed, typically manually pre-defined, trade-off of information freshness on the one hand and network overhead on the other, for all monitored resources (this holds for both prefetching and just-in-time evaluation: in prefetching (resp. just-in-time evaluation) the trade-off is determined by the time interval between consecutive updates (resp. the cache TTL, i.e., the time interval query results are held in a cache);

- in a large-scale grid one may reasonably expect that not all resources are of equal importance.

The first observation is supported by the fact that most of the performance studies of grid monitoring and information services do not explicitly measure the achieved information freshness, as they assume a fixed frequency of updates for all sites.

Based on resource importance, one can redefine the large-scale grid monitoring problem by allowing the information freshness versus network overhead trade-off

to be dealt with at a site-level basis, taking into account the importance of grid resources at every site. This would imply an increased frequency of updates for the sites that host the most important resources, while lower freshness is maintained for those hosting the least important resources. Moreover, the frequency of updates per site can be dynamically adjusted by periodically re-evaluating the importance of all known grid resources.

This chapter makes the following contributions:

- A new approach to large-scale grid monitoring, which introduces the concept of resource importance as a basis to determine and periodically re-evaluate the frequency of updates per site. As a result, it is now useful to explicitly measure the level of information freshness, as it can be varied across sites.

- An architecture that is sufficiently flexible to realise the outlined approach. In particular, an architecture that allows the calculation of resource importance of all sites, which assumes that resource information is centrally located (as opposed to distributed across many administrative domains).

- The use of differential updates (i.e., avoiding to re-collect previously retrieved resource information) to reduce network overhead.

- Models for the prediction of the performance of the proposed architecture, in terms of average information freshness and imposed network overhead, for given settings.

The proposed architecture has been implemented as a prototype. The term "prototype" will be hereafter used to refer to the prototype realisation of the proposed architecture. The exact definition of resource importance is orthogonal to the issues described in this chapter. As such, resource importance is defined in Chapter 5.

This chapter is structured as follows. Section 4.1 introduces the main concepts of RDF, which are required for understanding the proposed architecture's resource information model in this chapter, and the definition of resource importance in the next chapter. Section 4.2 discusses related work. Section 4.3 describes the proposed architecture, i.e., the components and interactions between them. Section 4.4 elaborates on the prototype's resource information model. Section 4.5 describes a hypothetical monitoring scenario to exemplify the operation of the

proposed architecture. Section 4.6 defines the performance metrics that are used to evaluate the proposed architecture in Chapter 6. Section 4.7 presents models for the estimation of the architecture's performance in terms of average information freshness and network overhead in given settings. Section 4.8 discusses various issues about the proposed architecture. Specifically, the section explains the trade-offs that need to be considered as a result of the architecture's features (importance-aware monitoring and differential updates), discusses the matter of information heterogeneity, and relates the architecture to hierarchies of aggregation that follow different modes of evaluation. Section 4.9 concludes the chapter.

4.1 Background

The prototype's resource model in this chapter and the definition of resource importance in Chapter 5 require an understanding of the main concepts of the Resource Description Framework (RDF). Thus, this section provides a brief introduction to RDF.

RDF [rdf04c, Pow03] is a recommendation of the World Wide Web Consortium (W3C) for describing resources in a well-structured and unambiguous way. RDF is an information model and is associated with several formally specified ways of information representation (i.e., notation). In RDF, everything is expressed in triples of subject, predicate (or property) and object. An RDF triple, similarly to a statement in English, states that a subject is associated with an object in terms of a property.

Subject A subject can be either a resource identified by an URI,[1] or an anonymous resource (blank node or bnode in RDF terminology) which for some reason is not considered worthy of a URI. The present work does not use bnodes so that every subject is identifiable. Subjects may have an arbitrary number of properties.

Property A property is always a URI.

Object An object is either a resource (i.e., URI or bnode), or a literal value.

Although subjects, properties and non-literal objects are all URI resources, we will use the term "resource" to refer to subjects. Properties will always be

[1] An URI may be a unique name for an abstract resource, or a location identifier of a network resource (e.g., an address).

referred as such. Objects, when required, will be qualified as literal or non-literal objects. This is just a convention: a non-literal object of one triple may also be the subject of another triple; referring to such a resource as an object is intended to clarify the context in which it is discussed.

Because subjects and non-literal objects are URIs, a collection of RDF statements forms a labelled graph, in which nodes are subjects and objects, and edges are properties. Information that is modelled after RDF can be notated in various ways, including a visual representation of the respective graph and RDF/XML. RDF/XML [rdf04b] specifies a standard XML syntax for representing RDF triples.

Often, one needs to explicitly specify the meaning and relations of the URIs used in a given RDF information model. This can be accomplished using RDF Schema (RDFS). RDFS [rdf04a] is a W3C recommendation that allows one to specify the intended use of resources (e.g., that http://example.com/#author is a type of a resource, whereas http://example.com/#hasAuthored is a type of property), as well as the relation of resource types (e.g., that a host is a subclass of a computing element). Every resource is an instance of a resource type or *concept*; a resource is an instance of a concept in the same way that an object is an instance of a class in object oriented programming. The type of a resource is denoted with the standard property rdf:type. Furthermore, one can specify the *domain* of a property, which is the types of resources or literals the property can be associated with.

4.2 Related Work

The grid community has developed several monitoring and information services (e.g., [CFFK01, MCC04, WSH99, TF04, BKPV01, ABF+05, CGN+04]), most of which can be manually configured (e.g., via a static configuration file) for either prefetching or just-in-time evaluation (described in Section 3.3). The proposals in [KLH+05, DL05] are similar to the present thesis in that they also collect grid-wide resource information at a central relational database. Both proposals collect detailed host and network related information. [DL05] deals with the large amount of data with purpose-built query processing techniques. [KLH+05] deals with the problem by abstracting host-level information to user-oriented higher-level concepts (e.g., strongly or loosely connected set of hosts). The architecture

that is proposed in the present thesis is similar to that of [KLH+05] in that it also uses higher-level abstractions. The difference is that our prototype collects abstracted data (about groups of hosts) instead of collecting host-level data and building the abstractions afterwards. In particular, the prototype implementation of the proposed architecture collects data about subclusters, where a subcluster is a homogeneous set of hosts. In general, the present thesis is complementary to the aforementioned proposals. For instance, the present thesis can benefit from the query techniques in [DL05]. Also, the work in [KLH+05] focuses on information querying but does not prescribe how monitoring is performed. This need can be fulfilled by the monitoring framework that is the subject of the present chapter.

The wide success of peer-to-peer systems motivated work on its application for grid resource discovery (e.g., [IF01, IF04]). Flooding-based peer-to-peer systems have been shown to have scalability problems [Rit01] and cannot guarantee a bounded query response time. Peer-to-peer systems that are based on a distributed hash table (DHT) (e.g., [RFH+01, ZKJ01, SMLN+03]) typically support key-to-address translation, thus by default do not support multi-criteria queries. Andrzejak and Xu [AX02] enhance Chord for supporting range queries (e.g., RAM $\geq$ 2GB) but only for a single attribute per query. The MAAN proposal (Multi-Attribute Addressable Network) [CFCS03, CFCS04] also extends the Chord DHT framework to support multi-attribute and range queries but suffers in terms of query response time for queries with large selectivity,[2] and assumes a predefined and fixed information schema. An extensive survey of peer-to-peer based grid resource discovery can be found in [RHB06].

Sundaresan *et al.* [SLK+03, SKL+03] propose a host monitor in which the frequency of updates is dynamically adjusted, and information freshness is explicitly measured. The update frequency is adjusted after every update, based on whether the previously monitored value is up to date compared to the value retrieved from the latest update. In addition, [SLK+03, SKL+03] evaluate various methods to predict the next significant value change, based on historical data. [SLK+03, SKL+03] focus on memory availability, which varies based on the way a host is used (e.g., desktop, server). The present thesis is similar to the work in [SLK+03, SKL+03] in that it also dynamically adjusts the frequency of updates and explicitly measures information freshness. However, the present thesis adjusts the frequency of updates based on the importance of resources in a site, as

[2]Query selectivity is the percentage of queried resources that match a given query.

opposed to the frequency of resource changes.

Secure Service Discovery (SDS) [CZH+99] is a proposal for a hierarchy of aggregation that combines both prefetching and just-in-time evaluation (Section 3.3). Complete resource descriptions are kept at the leaf nodes, which pass summaries of resource descriptions (using hashing) at their parents. Higher-level nodes produce lossy aggregations of the summaries using bloom filters [Blo70]. SDS resembles prefetching because summaries are sent upward the hierarchy regardless of the arrival of queries. It also resembles just-in-time evaluation because the summaries are only meant to filter out the branches of the hierarchy at which query matches do not exist; the query has to be forwarded to child hosts to actually retrieve the information for the matched resources. SDS reduces network overhead at the cost of more computation, and potentially reduces query response time by not forwarding a query to child nodes that are known not to have a query match. Range queries and wildcards are not well supported, as resource summaries (by means of hashing and bloom filters) capture only exact information.

An important feature of the present thesis is the trade-off between information freshness on the one hand and network overhead on the other. A relevant effort in the context of the web [DKP+01, BDK+02] evaluates adaptive combinations of push and pull data acquisition methods to explore the trade-offs among freshness tolerance, computational and network overheads and resiliency. Freshness tolerance refers to the acceptable deviation of the monitored value from the actual value, or the maximum age of non-fresh values (e.g., a user may assert that "non-fresh values are acceptable but only for up to 5 minutes"). The work in [DKP+01, BDK+02] is concerned with the frequency of change of a single monitored value and the acceptable tolerance of lack of freshness. This approach is not easily applicable in the grid context, as one would have to consider the frequency of change and freshness tolerance for all monitored resource properties. Instead, the present thesis assumes that user expectations about information freshness vary based on the importance of a resource, i.e., that users are more likely to tolerate stale information when it refers to less important resources.

None of the cited systems has a notion of resource importance, and hence none allows to treat resources differently based on their importance.

Relevant to the present chapter's contribution of an information freshness model is a similar study for the purpose of synchronising web pages using web

crawling [CGM03a]. [CGM03a] assumes that resource changes follow a Poisson process. That is, resource changes are independent, occur at a fixed rate over time, and the time between consecutive changes is exponentially distributed. In the case of clusters, there is certainly dependence between many types of changes. For instance, the completion of a job affects the number of available CPUs, the number of running jobs, or even the percentage of failed jobs, if a job is not successful. Instead, the present work assumes the more general case where resource changes are uniformly distributed over time.

Also, [CGM03a] proposes methods to determine the optimal frequency of updates for synchronising resources that change at different frequencies, with the objective of achieving the highest freshness. This is relevant in the web, where one web page may change once a day while another once a year. However, it is not particularly relevant for monitoring cluster changes, such as pending jobs and available CPUs, because clusters typically have much less diverse patterns of change (e.g., they typically change at least several times during a day).

4.3 Overall Architecture

The proposed architecture, shown in Figure 4.1, is an adaptation of the web crawling paradigm [ACGM+01] for the purpose of large-scale grid monitoring [DIS04, DSI06, ZS04]. In the figure, boxes with solid lines denote administrative boundaries; boxes with dashed lines denote hosts; boxes with rounded corners denote software programs. The architecture distinguishes between two kinds of administrative domains: grid sites, which host grid resources, and one (or more) monitoring site, which monitors the resources across grid sites. These two kinds of sites, and the functionality therein are the subject of this section.

4.3.1 Grid Sites

Administrative domains that host grid resources are referred as grid sites. A *grid site* is one or more networks and resources therein with a common sharing policy, and at least one grid information service. (Hereafter, the term "site" when not qualified refers to a grid site.) Every grid site is expected to host a proxy. A *proxy* periodically collects information about the resources in its site, either directly from resources themselves or via the site's information service(s) (the latter case is shown in Figure 4.1). The purpose of proxies is to provide

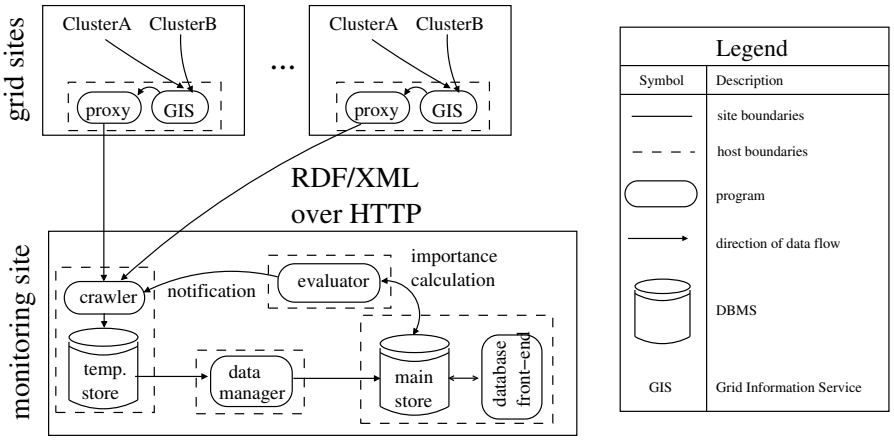

Figure 4.1: Overall system architecture.

a contact point with a consistent network interface and resource representation for every grid site. The need for proxies arises as a result of the diversity of existing grid information services (further discussed in Section 4.8.2). Modelling and representation matters aside, proxies provide information for local resources upon request, via standard HTTP (HyperText Transfer Protocol).

A query to a proxy results in one of the following response types:

full update A full update is the response to an unconditional query; an unconditional query requests data about all the resources that are hosted at a proxy's site (first line in Table 4.1).

resource-specific update A resource-specific update is the response to an unconditional query that requests data about a specified resource (second line in Table 4.1).

differential update A differential update is the response to a conditional query about all the resources that are hosted at a proxy's site (third line in Table 4.1). A conditional query specifies a timestamp using the standard if-modified-since HTTP header. The result of the query is that of a full update, excluding the data that has not changed since the specified timestamp.

differential resource-specific update A differential resource-specific update is similar to a differential update, except that the scope of the queried data is limited to the specified resource (fourth line in Table 4.1).

request	information that is requested	response type
GET / HTTP/1.0	all available resource information	full update
GET /Cluster123 HTTP/1.0	information about Cluster123	resource-specific update
GET / HTTP/1.0 if-modified-since: timestamp	all available resource information that has changed after the specified timestamp	differential update
GET /Cluster123 HTTP/1.0 if-modified-since: timestamp	information about Cluster123 that has changed after the specified timestamp	differential resource-specific update

Table 4.1: Examples of proxy request types.

The purpose of differential updates is to minimise the network overhead of monitoring by not collecting information more than once, i.e. information that is already available at the monitoring site and is still up to date. A differential update may include three types of events: (i) a change of information that has been previously collected; (ii) an addition of new resources, and thus newly available information that has to be collected; (iii) a removal of previously monitored resources, in which case the relevant information about the removed resources at the monitoring site has to be discarded.

Differential updates are further explained in Section 4.4, where the resource model is described in detail. The evaluation in Chapter 6 uses only full updates and differential updates; the resource-specific cases are listed here for completeness.

The differential updates that are supported by proxies in our architecture are context-free patches, according to the distinction in [BLC01], which distinguishes patches in context-free and context-sensitive. A context-free patch describes the differences between the original and the modified document by referring to named resources. In contrast, context-sensitive patches (such as the ones produced by the UNIX utility "diff") describe changes in terms of line numbers, thus can only be used when one has the original document from which the patch was computed. It is important that proxies' differential updates are context-free patches so that monitoring sites do not have to keep the textual representation of previously

```
 1 retrieve proxy addresses, importance of sites,
 2            timestamp of previous update at every site
 3 while true {
 4     for every site {
 5         if site has not been visited before {
 6             request full update from proxy
 7             store received update in temporary store
 8         } else if now - timestamp of previous site update
 9                           + window > site update interval {
10             request differential update from proxy
11             store received update in temporary store
12         }
13     }
14     if a notification was received from the importance evaluator {
15         retrieve latest site importance values from main store
16         re-evaluate the update interval of every site
17     }
18 }
```

Figure 4.2: High-level pseudo-code of the crawler.

retrieved updates.

4.3.2 Monitoring Sites

The purpose of monitoring sites is to provide large-scale information services for grid resources. To do so, a monitoring site has to systematically collect resource information from proxies throughout the Grid, keep it up to date, and perform any necessary pre-processing to enable high-performance querying of information about all known grid resources. The present architecture does not specify how proxies are discovered by monitoring sites. In the simplest case, grid site administrators would manually submit their sites' proxy addresses to a monitoring site (in the same fashion that some web search engines allow webmasters to register their web site). Alternatively, a monitoring site could use an automated method that scans IP addresses for certain services (e.g., [BHP04]).

The remainder of this section describes the software infrastructure of a monitoring site: a crawler, a data manager, an importance evaluator, a database front-end, and the relevant storage requirements (Figure 4.1).

Crawler The crawler is responsible for collecting resource information from proxies. At the core, the present implementation of the crawler is a HTTP client that uses non-blocking I/O for high performance, and implements policies to determine the frequency of requests to every proxy. Figure 4.2 shows a highly abstracted description of the crawler's logic (which does not account for the complex networking functionality).

At launch time, the crawler (lines 1-2 in Figure 4.2) retrieves from the main store:[3]

- the address of every site's proxy;
- the importance of every site;
- and the timestamp of the latest update that has been received from every site (a special value is used to denote that a site has never been visited).

The timestamp of the latest update of every site is renewed as soon as the crawler performs a new update at that site. Thus, before shutdown, the crawler has to store the latest timestamps at the main store. At the present implementation, however, these timestamps are not stored in the database. Instead, the crawler is explicitly instructed at launch time whether to treat all sites as previously visited or not.

The crawling process is described in lines 3-13 of Figure 4.2. The crawler iterates over the list of proxy addresses. If a site has never been visited before, a full update is requested. Otherwise, the crawler requests from the proxy a differential update, on the condition that the time since the previous update from that proxy plus a predefined `window` exceeds the update interval for the specific site.[4] The if-modified-since HTTP header for the differential update is set to the timestamp of the previous update that has been received from the specific site.

The importance evaluator notifies the crawler whenever the former re-evaluates the importance of all sites. On this occasion, the crawler retrieves the latest site importance values,[5] which are used to re-calculate the update interval of every

[3]The main and temporary stores are databases (discussed in the paragraph "Storage" of this section).

[4]`window` is meant to mitigate for potential delays between requesting an update from a proxy until receiving a response.

[5]As described in Chapter 5, the importance of a site is calculated based on the importance of resources that are hosted at that site, and the conditions under which the resources are being provided. Site importance is between 0 and 1.

site (lines 14-17 in Figure 4.2). In other words, the frequency of visits per site is determined based on site importance, the general principle being that the freshness of the information about a resource should be proportional to the importance of the resource the information refers to. (For practical reasons, the frequency of updates, and thus the attained level of information freshness, is determined on a site-level basis, as opposed to the level of individual resources).

The crawler implements two ways of mapping site importance to frequency of updates: *topn* and *proportional*. In *topn mode*, the crawler operator specifies the time between consecutive updates for (i) the top N% most important sites; and (ii) the remaining 100−N% less important sites (e.g., update the top 10% of sites every 5 minutes, and the rest 90% every 20 minutes). In *proportional mode*, the crawler operator specifies the minimum and maximum number of minutes between consecutive updates at every site, and the crawler dynamically maps importance to a value in that range inversely proportionally (e.g., given 1 and 15 minutes as the minimum and maximum time between consecutive updates, sites of maximum importance are updated every 1 minute; sites of minimum importance every 15 minutes; and all other sites are updated at a frequency that is inversely proportional to importance and lies in the range $[1, 15]$, computed at a granularity of seconds). In general, the time between consecutive updates $Mm \in [min_t, max_t]$ of a site with importance imp, where $imp \in [0, 1]$, is:

$$Mm = \begin{cases} (1 - imp) \times max_t, \text{ when } imp < 1 - \frac{min_t}{max_t}; \\ min_t, \text{ otherwise.} \end{cases} \tag{4.1}$$

Data Manager The resource information, as provided by proxies, is encoded in RDF/XML. (As explained in Section 4.1, RDF/XML is a standard way to notate RDF triples in XML.) However the current state of the art for querying RDF cannot deliver the low query response time that is required in grid-wide information services. On this basis, resource information is transformed for storage in a relational database. Specifically, a data manager uses the incoming RDF/XML resource updates to generate the appropriate SQL insert, update and delete statements. The data manager constructs such statements on the fly based on a predefined mapping between the source RDF schema and the target relational schema.

Importance Evaluator The importance evaluator (or simply evaluator) is the program that implements the definition of resource importance to calculate the importance of all sites. The importance values are periodically re-evaluated to reflect potential changes that can affect a site's importance (e.g. the addition of a new cluster service). The assumption is that the importance of sites may vary over time, and that has to be taken into account for the adjustment of the update frequency of every site. Upon completion of re-evaluating resource importance, the evaluator notifies the crawler to retrieve the latest importance of all sites and re-evaluate accordingly the frequency of updates per grid site.

Storage The crawler stores proxy responses (i.e., full and differential updates) in a temporary store, along with the time at which they are downloaded. The data manager processes the updates as a FCFS queue (i.e., ordered by time of arrival and starting from the oldest). Both the temporary store and the main store are RDBMSs, although a filesystem could also be used for the temporary store. The main store, however, is required to have the query expressiveness of SQL. The RDF/XML updates in the temporary store are processed by the data manager to generate the appropriate SQL statements for inserting, updating and removing resource information to and from the main store. Conventional database replication techniques can be used to distribute the query load imposed on the main store across several databases.

Database Front-end The database front-end is a program that accepts query identifiers via a TCP port, and forks a message handler thread for every incoming message. A message handler thread submits the query that corresponds to the identifier that is specified in the received message, to the main store, and records QRT upon the completion of the query. The database front-end is located at the same host as the main store. In a setting where the main store would be replicated, the database front-end should also perform load-balancing.

Query Generator For the purpose of the evaluation in Chapter 6, a query generator was developed as a way to emulate predefined query workloads. A query workload determines the total number of queries, query types, and the arrival of query submissions over a given time interval. At the beginning of every experiment, a query generator schedules a given number of queries over the experiment duration, according to a user specified query workload. Whenever

a query is due for submission according to the generated schedule, the query generator forks a thread that submits a query identifier to the database front-end.

In a production-quality implementation of the proposed architecture, the query generator would be substituted with a program that constructs SQL queries based on user-supplied queries.

4.4 Resource Model

The prototype uses a simplified version of the GLUE information model [ABF+], which specifies the properties and relations of grid resources. The proposed architecture is intended to operate on top of the existing information services, thus it seems reasonable to adopt the GLUE model as it is widely used.

A service in the GLUE model is typically realised by a web service, a computer cluster or a storage element. The GLUE model defines load properties at various levels of abstraction. We aggregate load-relevant properties at the level of clusters, as it is not practical to monitor highly dynamic host-level properties, such as available RAM, at a large scale. The current implementation handles only the cluster realisation of services, so hereafter *cluster service* is used instead to refer to a service that provides access to a cluster via some middleware, subject to sharing conditions. (The term is intended to avoid confusion with web services.) According to the GLUE information model, a cluster is composed of one or more subclusters, where a subcluster is a homogeneous set of hosts. Also, we define a sharing policy as a set of sharing options, where every option specifies authorisation (i.e., list of users, or users that have a given role/property) and pricing details.

As part of the prototype development, the adapted GLUE model has been mapped to the Resource Description Framework (RDF) information model, and the current implementation uses the RDF/XML notation. Figure 4.3 lists the concepts of the adapted GLUE information model, and how they are related. Concepts that are one level of indentation apart have a parent-child relation; concepts at the same level of indentation are siblings. For instance, the top-level concept, Site, relates directly to the concepts SiteAvailability, InetAccess, Location and Services. Figure 4.4 shows the cluster part of the information model.

An important part of monitoring is the use of timestamps to denote the latest

```
Site
    SiteAvailability
    InetAccess
    Location
    Services
        Service
            ServiceAvailability
            Policy
                SharingOptions
                    SharingOption
            ComputingElement
                LRMS (local resource management system)
                ComputingElementState
                    FreeJobSlots
                    WaitingJobs
                    FractionOfJobsFailed
                    EstimatedResponseTime
                    WorstResponseTime
                Cluster
                    SubCluster
                        Host
                            Software
                                ApplicationSoftware
                                OperatingSystem
                            Hardware
                                Processor
                                LocalStorage
```

Figure 4.3: Relation of concepts of the resource information model.

change of every piece of information. For this purpose, every resource in the RDF model normally has a "hasTimestamp" property. To avoid unnecessary duplication, a resource may not have such a property, in which case it inherits the timestamp of the closest ancestor. For example, a cluster service's resources will all have the timestamp at which the cluster service was added. If at a later time, the cluster service's operating system is upgraded, then the properties of the relevant OperatingSystem resource will be updated accordingly and a new timestamp property will be added to the OperatingSystem resource (which will override the ancestor cluster service's timestamp).

Having introduced the hierarchy of concepts in the information model and the use of timestamps, one can now explain the exact meaning of differential updates

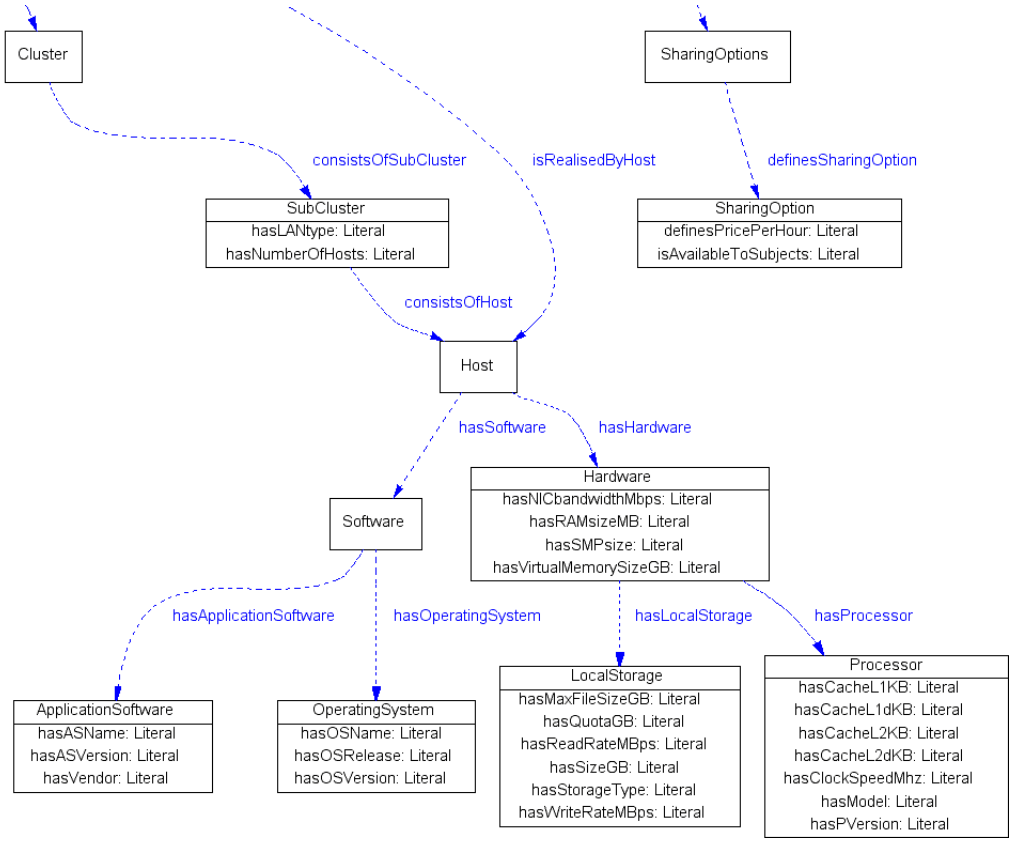

Figure 4.4: Part of the RDF schema of the adapted GLUE model. Concepts and associated literal properties are within boxes (upper and lower parts respectively); arrows indicate non-literal properties.

from proxies. An if-modified-since request to a proxy for a specific resource returns the changes related to that resource and all its descendants that have occurred since the specified timestamp. For instance, consider that a crawler performs a full update for a site, an operating system upgrade is performed later on at one of the site's clusters, and eventually the crawler queries again the site's proxy, this time using the if-modified-since header. The information about the upgraded cluster in the latter query response will be restricted to the properties related to the modified OperatingSystem resource.

4.5 Example Operation

To exemplify, this section describes the operation of the described architecture in a hypothetical scenario. The following assumptions are made:

- the monitoring site has a list of proxy addresses;
- the crawler operates in topn mode, in which the top 10% sites are updated every 5 minutes and the remaining 90% of sites are updated every 20 minutes;
- the importance evaluator re-evaluates the importance of sites every 24 hours.

At launch time, the crawler retrieves the addresses of all known proxies. For simplicity, the discussion assumes that all sites have not been monitored before. The crawler requests from every proxy a full update (containing data about all resources at every site). Every update that is received is stored by the crawler at the temporary store. Some proxies may occasionally timeout, in which case the crawler resets the corresponding connections and re-sends the request. This process is performed until all proxies have responded with a full update.

At the same time, the data manager constructs SQL INSERT statements to populate the main store based on the full updates that are placed at the temporary store by the crawler. As soon as all updates are received by the crawler and processed by the data manager, the latter notifies the importance evaluator. The importance evaluator calculates every site's importance, stores the outcome at the main store, and notifies the crawler.

The crawler retrieves from the main store the latest importance values of all sites. The values are used to rank the sites. Based on the ranking the crawler determines whether a site should be updated every 5 or 20 minutes. Eventually, at 295 seconds (5 minutes minus a window of 5 seconds) after the receipt of the full update of every one of top 10% most important sites, a new request is sent to every one of the topn sites. The request is conditional, using the timestamp of every site's previous update as the value for the if-modified-since header. The crawler stores the differential updates at the temporary store as soon as they arrive. The data manager uses the differential updates to construct the equivalent SQL UPDATE statements.

The same process is performed two more times (at 10 and 15 minutes since the initial crawl) until minute 20, at which point all proxies, including the 90% least important, are contacted for differential updates.

In the mean time, some sites may host new cluster services while others may seize to host previously offered cluster services. These additions and removals are reported by proxies in differential updates. In these cases, the data manager constructs SQL INSERT and SQL DELETE statements to keep up to date the main store. After the user-specified 24 hour interval, the importance evaluator re-evaluates the importance of all sites, taking into account potential changes in the cluster service offerings at every site. Once again, the crawler is notified by the importance evaluator, and re-calculates the update interval per site accordingly.

4.6 Definition of Performance Metrics

This section defines the performance metrics for the evaluation of the proposed architecture: network overhead, average information freshness, and average query response time.

4.6.1 Network Overhead

The network overhead, v, is the number of bytes that the crawler downloads during a given time interval. In the following definition U is the set of updates that are downloaded by the crawler during the given interval, and b_i is the number of bytes of the i^{th} update.

$$v = \sum_{i \in U} b_i. \tag{4.2}$$

In the context of the proposed architecture, one can distinguish the network overhead to that of (i) the full updates that are performed at the first visit at every site; (ii) the network overhead of differential updates that are performed at previously visited proxies; and (iii) the total network overhead, which is the sum of both full (i) and differential updates (ii). Unless otherwise specified, network overhead will refer to that of differential updates.

Note that the above distinctions are not applicable to the definition of network overhead for JITHAs (just-in-time hierarchies of aggregation). In that context, network overhead refers to the number of bytes that are collected by the top-level node of a monitoring hierarchy of aggregation during a given interval, as part of resolving resource discovery queries.

4.6.2 Average Information Freshness

A value for a particular resource property at a monitoring site is considered fresh when it is synchronised with its real-world equivalent value. The following is the definition of freshness of an information collection C at time t that has k resource property values p_j:

$$F(C,t) = \frac{1}{k}\sum_{j=0}^{k-1} F(p_j,t) \tag{4.3}$$

where

$$F(p_j,t) = \begin{cases} 1, \text{ if the property value } p_j \text{ is up to date at time } t \\ 0, \text{ otherwise.} \end{cases}$$

The above states that the freshness of an information collection at a given time is calculated as the percentage of resource properties that are up to date at that point in time.

Furthermore, one can define time-average information freshness or simply average freshness as the average of information freshness values $F(C,t)$ at j consecutive equally-distanced time points during a time interval T:

$$AF(C,T) = \frac{1}{j}\sum_{t=0}^{j-1} F(C,t). \tag{4.4}$$

4.6.3 Average Query Response Time

Query response time (QRT) is measured as the server-side cost, i.e., the interval from the receipt of a query from the database front-end until the results are ready to be sent back to the query originator. In other words, QRT does not include the time to transmit the results to the query originator. Average QRT refers to the average of a set of response times to queries that are submitted over a given time interval.

4.7 Performance Modelling

The proposed architecture allows the trade-offs of information freshness and network overhead to be determined at the level of individual sites. Given this increased flexibility, the operator of a monitoring site should know (i) the required frequency of updates to attain a certain level of information freshness, and (ii) the network overhead that is imposed as a result of monitoring at a given update frequency. This section addresses these two questions, in the context of monitoring clusters of computers. It is assumed that changes of cluster-related information are uniformly distributed over time.

4.7.1 Average Information Freshness Modelling

This section presents (i) a simulator that yields the expected level of freshness for given settings of frequency of updates (Rm) and frequency of resource information changes (Rc); (ii) two analytical functions that give the expected level of freshness for given Rm and Rc, and the Rm that is required to attain a desired level of freshness for given Rc.

Simulation

The frequency of updates, Rm, is the number of updates that a crawler requests from a certain proxy per time unit. Thus, if updates are performed every Mm time units, $Rm = \frac{1}{Mm}$. Similarly, Rc is the number of resource changes per time unit, and Mc is the average number of time units between consecutive changes.

A simulator has been implemented that accepts parameters for Rm, Rc, duration of experiment (D), and a time interval (I, explained below). Based on the input parameters, the simulator calculates the total number of changes

($Nc = Rc \times D$) and updates ($Nm = Rm \times D$) and uniformly distributes those changes and updates across D. The time interval I determines the granularity of the calculated average information freshness. For instance, for $I = 5$, information freshness is calculated at time $5, 10, 15, \ldots$, etc. In general, information freshness is calculated at the end of consecutive time intervals $i = n \times I, \forall n \in [1, \frac{D}{I}]$.[6] The average freshness is calculated as the average of all those freshness values. Because the distribution of changes and updates is random, every setting is simulated 1000 times, based on which we report the average and standard deviation of information freshness.

We have simulated 41 settings with the following parameters: $D = 8640$ time units, $I = 100$, $Rm = \frac{1}{40}$ (i.e., one update every 40 time units), and Rc ranging from 1 to $\frac{1}{8000}$ (i.e., one change every one time unit, down to one change every 8000 time units). (The exact settings for Rc are not significant, as long as a large range of values of the $\frac{Rc}{Rm}$ ratio is covered.) I and D determine the granularity of measured average freshness; the measurement accuracy is higher with low values of I and high values of D ("low" and "high" in relation to the frequency of updates and changes.) Figure 4.5 shows the average and standard deviation of information freshness (y axis) for every examined ratio of Rc over Rm (x axis). It can be seen that for $\frac{Rc}{Rm} = 1$, freshness is 0.5, which is reasonable because both resource changes and information updates are uniformly distributed across D.

Also of interest is that the standard deviation of information freshness is small for extreme (small or large) values of x and peaks at around $x = 1$ (0.048 for $x = 1$, 0.016 for $x = 40$, 0.003 for $x = 0.005$). This is because for a given setting of Rc and Rm, freshness varies depending on the exact timing of updates and resource changes. On one extreme, changes can occur right after updates (resp. right before updates), in which case freshness tends to zero (resp. tends to one). The significance of the randomness of timing appears to be limited with extreme values of x.

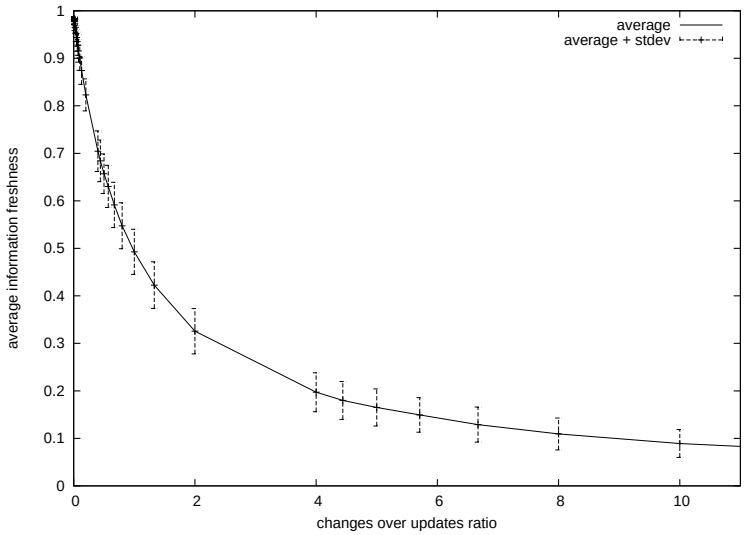

Figure 4.5: Average information freshness (y axis) as a function of the ratio $\frac{Rc}{Rm}$ (x axis), based on the simulation.

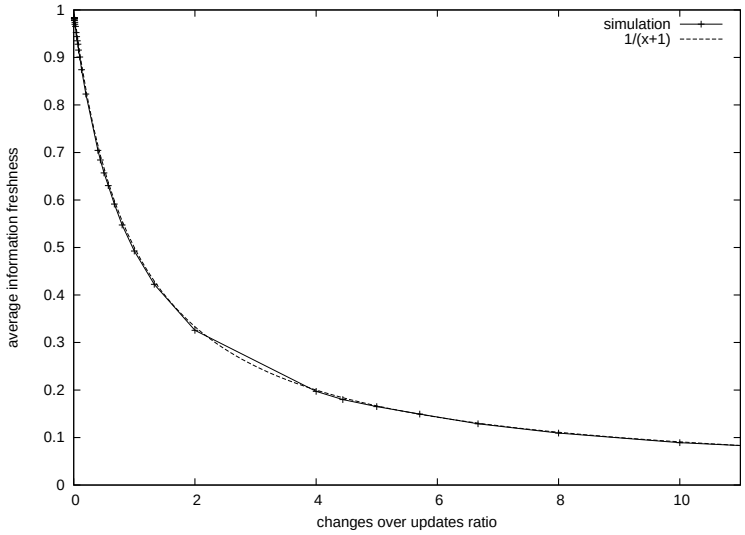

Figure 4.6: Average information freshness (y axis) versus the ratio $\frac{Rc}{Rm}$ (x axis), based on simulation results and Equation 4.5.

Analytical Model

The simulation results fit well to an exponential decay model (Figure 4.6):

$$f(x) = \frac{1}{x+1} \tag{4.5}$$

where $f(x)$ is the expected average information freshness for $x = \frac{Rc}{Rm}$. The function maps Rm and Rc to freshness, and *vice versa*, for arbitrary values. Equation 4.5 can be understood as follows. Information freshness varies in the range $[0, 1]$ based on the relation of Rc and Rm, as captured by their ratio x. When x is large, i.e., when the frequency of resource changes, Rc, is much higher than the frequency of updates, Rm, then freshness tends to 0. On the other extreme, when x tends to zero, then $f(x)$ tends to 1. When, $Rc = Rm$, $f(x) = 0.5$ because both information changes and updates are uniformly distributed.

By substituting $x = \frac{Rc}{Rm}$ in Equation 4.5 and with simple algebraic manipulations, one can derive the function that gives the required frequency of updates Rm, to maintain a desired level of freshness F for a given frequency of changes Rc:

$$Rm(Rc, F) = \frac{Rc}{\frac{1}{F} - 1}. \tag{4.6}$$

For instance, if one requires an average information freshness $F = 0.8$ and the information changes on average every 15 minutes ($Rc = \frac{1}{15}$) then $Rm = 0.266$, i.e., updates should be performed every $Mm = \frac{1}{Rm} = 3.75$ minutes.

4.7.2 Network Overhead Modelling

To estimate the network overhead of differential updates, when monitoring over a period D, the following parameters have to be taken into account.

$\mathbf{F_{i,s}}$ average information freshness about property type i of services in site s (calculated according to Equation 4.5, in which Rm is specific to site s, and Rc is specific to property type i, i.e., $x = \frac{Rc_i}{Rm_s}$);

nupdates the number of differential updates that the crawler requests from the proxy of site s:

$$\text{nupdates}_\text{s} = \text{D} \times \text{Rm}_\text{s}. \tag{4.7}$$

[6]The first resource change is always scheduled at time zero, to ensure that freshness is computable at all times (i.e., calculation of freshness is not meaningful when there is no resource state to be monitored).

When Rm is the same for all sites, the total number of updates can be estimated as:

$$\text{nupdates} = \text{D} \times \text{Rm} \times \text{nsites}. \tag{4.8}$$

nmchanges the number of nchanges that are monitored during an experiment (monitored changes). For instance, consider the following sequence of resource changes and updates: (i) change; (ii) update; (iii) change; (iv) change; (v) update; (vi) update; (vii) change. Change (i) is monitored by update (ii); change (iv) is monitored by update (v); changes (iii) and (vii) are never monitored. In the example 4 resource changes occur (nchanges); the crawler performs 3 updates (nupdates), which monitor only 2 resource changes (nmchanges).

Consider the relation of nupdates and nmchanges. A single differential update describes zero or more resource changes.[7] The information about a certain resource property is included in a differential update only if the previously collected information about that property is stale. Thus, the number of monitored changes of properties of type i, $\text{nmchanges}_\text{i}$ at a site s, is the number of performed updates times the probability that a property of type i is not fresh (i.e., $1 - \text{F}_{\text{Rc}_\text{i},\text{Rm}_\text{s}}$)

$$\text{nmchanges}_{\text{s,i}} = \text{nupdates}_\text{s} \times (1 - \text{F}_{\text{Rc}_\text{i},\text{Rm}_\text{s}}) \tag{4.9}$$

where

- Rc_i is the frequency of changes of properties of type i; and
- Rm_s is the frequency of updates of site s.

Equation 4.9 is specific to a single property type, and assumes that every site has only one service. For a site s with nservs_s services, and assuming that every service has a single instance of every property type i, the overall number of monitored changes at site s is:

$$\text{nmchanges}_\text{s} = \sum_{\text{i} \in \text{P}} \text{nupdates}_\text{s} \times (1 - \text{F}_{\text{Rc}_\text{i},\text{Rm}_\text{s}}) \times \text{nservs}_\text{s} \tag{4.10}$$

[7] Differential updates may also describe the addition of new cluster services or the removal of previously existing ones. The presented model does not account for such events, because they are far less frequent than resource changes (compare, for instance, the frequency of installing a new cluster against the frequency of such a cluster changing status).

where P denotes the set of resource properties that change during an experiment, and nservs_s is the number of services at site s. When Rm and nservs_s are constant across all sites, the total number of monitored changes is estimated as:

$$\text{nmchanges} = \sum_{\text{i}\in\text{P}} \text{nupdates} \times (1 - \text{F}_{\text{Rc}_\text{i},\text{Rm}}) \times \text{nservs}_\text{s}. \tag{4.11}$$

Counter-intuitively, the number of monitored changes of a property of a given type is inversely related to the average information freshness for that type. Consider the two extremes. If the collected information about a property is always fresh then the number of monitored changes will be zero, as there will be no changes to be monitored in the first place. If the information about a property is always stale, then the number of monitored changes will be as many as the performed updates. Thus, the lower the freshness, the higher the number of nmchanges and thus the higher the network overhead is.

Given nmchanges, the network overhead due to differential updates during an experiment is the sum of the network overheads for every property type.

$$\text{overhead}_\text{s} = \sum_{\text{i}\in\text{P}} \text{nmchanges}_{\text{s,i}} \times \text{size}_\text{i} \tag{4.12}$$

where

- $\text{nmchanges}_{\text{s,i}}$ is the number of monitored changes of properties of type i at site s;
- size_i is the average number of bytes required for the description of a property of type i. These sizes depend on the used information schema, and are calculated based on experiment logs.

The overhead of differential updates for all sites, is the sum of all the site-level overheads (i.e., $\sum_{s\in Sites} \text{overhead}_\text{s}$). When one can assume an average size of descriptions of changes regardless of property type, the network overhead can be estimated as:

$$\text{overhead} = \text{nmchanges} \times \text{size}_\text{avg}. \tag{4.13}$$

4.7.3 Critique of the Performance Models

This section discusses the limitations of the presented information freshness and network overhead models.

Average Information Freshness

The information freshness model yields the expected average information freshness, for given settings of frequency of resource changes and frequency of updates, over a certain time interval. However, in addition to the aforementioned frequencies of changes and updates, information freshness also depends on practical constraints, as determined by network behaviour and a monitoring system's scalability. For instance, even though a site's proxy may be queried for an update every N time units, the effective update interval may be much larger, because of any of the following reasons.

- The proxy's response to a query for an update may be received after an arbitrarily long time, because the network or the proxy itself, or both are overloaded.
- The data manager may be overwhelmed with incoming updates, or the main store may be overloaded as a result of a large number of user queries. The extent that such issues arise depends on several factors, including the size of the monitored grid, the frequencies of resource changes and monitoring updates, the query workload, and the amount of engineering that is employed to make the monitoring site's software scalable (e.g., use of batch SQL UPDATES by the data manager, replication of the main store for load balancing of user queries).

The freshness model does not attempt to capture the implications of the aforementioned issues. Addressing the first item requires modelling of wide-area network behaviour, which is beyond the scope of the present work. The latter, involves parameters that are specific to a certain implementation and deployment of a monitoring site's software infrastructure.

Network Overhead

The network overhead model estimates the network overhead of differential updates, given (i) the average information freshness and (ii) the average size of the description of changes of properties of certain types that change over time. This assumes the availability of a reliable estimate of average information freshness, and that the size of the description of a change of a certain resource property type does not vary dramatically over time. For instance, the size of a differential update that describes the latest change of a cluster service's number of WaitingJobs

property cannot vary significantly because the property's value is a number (and thus will always be up to a few bytes).

4.8 Discussion

4.8.1 Prefetching Monitoring Strategies

This section is intended to give a concrete idea of how the network overhead that is imposed by monitoring, varies based on whether differential updates and resource importance are used. Four different monitoring strategies are considered:

- (i) retrieving full updates at every visit (*full updates*);
- (ii) retrieving full updates at the first visit to a proxy, followed by differential updates (*differential updates*);
- (iii) as in (i) but using different update frequencies for the topn most important sites (*full updates + importance*);
- (iv) as in (iii) but using differential updates (*differential updates + importance*).

The discussion assumes that 2000 sites are monitored over a 1000 time unit period, every 5 time units (unless otherwise specified). (Once again, the exact values are not significant; the purpose is to explore a wide range of values and demonstrate the relevant trade-offs.) Three values are considered for the following parameters, the mean value being the one that is used in the baseline setting. The average size of full updates is 1, 5, or 10 Kbytes. The average size of differential updates is 0.25, 0.5, 1 Kbyte (which are 5, 10 and 20% of the baseline setting of the size of full updates). In monitoring strategies (iii) and (iv), which vary the frequency of updates based on resource importance, the N most important sites are visited every 5 time units and the remaining of the sites every 10, 15 and 20 time units. The number of most important sites is 5%, 10% and 20% of all 2000 sites (i.e., 100, 200 and 400). Based on the above, the baseline setting is: 5 Kbyte average size of full updates; 0.5 Kbyte average size of differential updates; sites are distinguished in 200 most important and 1800 least important, and they are updated every 5 and 15 time units, respectively.

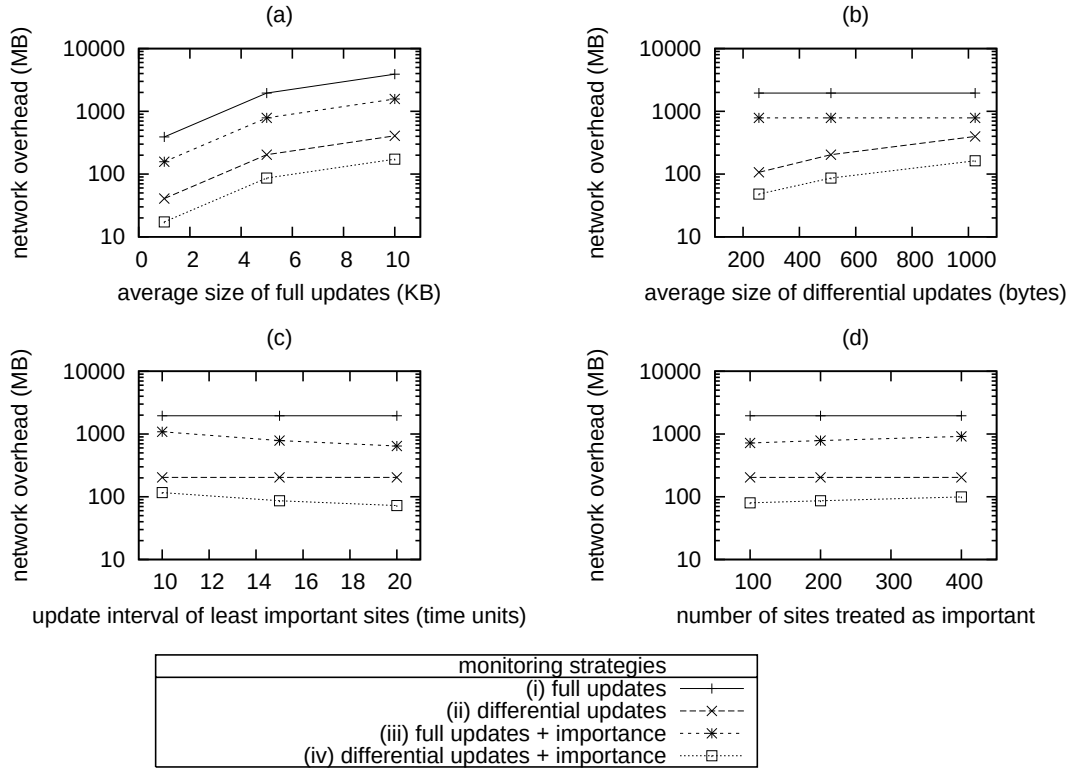

Figure 4.7: Network overhead for various monitoring strategies and settings (y-axis in log-scale).

strategy	i	ii	iii	iv
network overhead (MB)	1953	204	785	86
relative difference from strategy (i)	-	0.90	0.60	0.95

Table 4.2: The network overhead of different strategies in the baseline setting, when the update interval of least important sites is 15 minutes, and the relative difference of all strategies' network overhead against that of strategy (1) (calculated as $\frac{(a-b)}{a}$).

Each of the plots in Figure 4.7 shows the network overhead of all four monitoring strategies, when varying one dimension of the baseline setting. Figure 4.7(a) shows variations of the average size of full updates. As a result all strategies are affected, especially strategies (i) and (iii) as they keep on performing full updates on every visit. Figure 4.7(b) shows variations in the average size of differential updates, thus only (ii) and (iv) are affected. Figure 4.7(c) shows variations in the update interval of least important sites. This affects only importance-aware strategies, namely (iii) and (iv), in which network overhead decreases as the update interval increases. Figure 4.7(d) shows variations in the number of sites considered as important (again, only strategies (iii) and (iv) are affected). The network overhead increases with the number of sites that are visited most often, but the increase is not significant.

Table 4.2 lists the network overhead of the considered monitoring strategies, in the baseline setting of the bottom-left plot in Figure 4.7 (i.e., with a 15 minute update interval for the least important sites). The total network overhead is reduced more significantly when using only differential updates (strategy (ii)), compared to using only importance (strategy (iii)). Yet the benefit of taking into account resource importance is still significant. The network overhead of differential updates only (strategy (ii)) is reduced from 204 Mb to 86 Mb when differential updates is combined with the use of importance (strategy (iv)) (a relative difference of 0.58).

4.8.2 On Modelling, Representation and Protocol Heterogeneity

As already discussed in Chapter 2, existing information monitoring and information services are diverse in terms of the way they model, represent, and communicate resource information (i.e., network protocol).

The information modelling problem arises when the same set of resources are modelled in different ways. The problem is a result of independent efforts that typically have slightly different goals, and thus emphasise different aspects of the modelled resources. The Globus and UNICORE middleware are a typical example, as they both model computational resources, yet the former focuses on the resources themselves, whereas the latter models resources from the user's perspective. As a result, one model describes concepts that have no equivalent in the other, or some concepts exist in both models but are described in subtly different ways. This issue has been addressed by the Grid Interoperability project [BFGG04] by using an ontology that maps the common denominator of the concepts of the Globus and UNICORE models, and having an interpreter responsible for doing the translation between the two schemas. The present work does not claim any contributions for the modelling problem, but instead adopts the GLUE information model, which is widely used by the grid community.

The problem of having different representations of data, e.g. XML and relational, that conform to the same information model can be solved by mapping the two schemas. An example of this task in the prototype is the data manager, which transforms RDF/XML triples to relational records.

The diversity in network protocols is dealt with by having an intermediary that supports the protocols of local information services and provides a consistent network interface. In the presented architecture, this role is served by proxies, which provide information via HTTP.

4.8.3 Relating the Proposal to Other Types of Hierarchies of Aggregation

The proposed architecture is meant to support varying update frequency across grid sites, based on dynamically-calculated resource importance. Applying this approach to prefetching-based hierarchies of aggregation would require dealing with the following complications.

- The calculation of resource importance requires that information about all resources be at one place, thus would have to be performed at the top level of the hierarchy. The importance of grid sites would have to be communicated by the top level node to all other nodes, so that the latter adjust the update frequency of every site based on importance.

- Low-level monitoring nodes are meant to serve not only higher-level nodes but also users. So whatever adjustments would be made to information freshness based on resource importance would also affect users. In general, the monitoring servers in a hierarchy of aggregation belong to different administrative domains; thus it is unlikely that a single monitoring policy can be adopted.

Consider how the prototype compares to prefetching-based hierarchies of aggregation. The prototype is a prefetching-based two-level hierarchy: proxies are leafs, and a monitoring site acts as the top-level node. As far as users are concerned, the latter is a single-node, despite actually being composed of several hosts (crawler, data manager, etc.). Compared to typical prefetching-based hierarchies of aggregation, the prototype's hierarchy differs in important ways: the top-level node is composed of several hosts that are located at a single administrative domain, and every host serves a single purpose (e.g., the crawler contacts proxies, the data manager performs schema transformation, etc.). In contrast, in typical prefetching-based hierarchies of aggregation, every node implements all the functionality that is required to monitor and query resource information.

4.9 Closing Remarks

This chapter has redefined the problem of large-scale monitoring by allowing the information freshness versus network overhead trade-off to be determined on a site-level basis. The proposed approach, importance-aware prefetching, allows to collect information data in advance of query arrivals, thus achieving low query response times, while at the same time having the capability to (i) maintain high information freshness for sites that are sufficiently important; and (ii) control the imposed network overhead.

Importance-aware monitoring requires a definition of resource importance. Resource importance is used to determine the frequency of updates separately

for every grid site. This is further facilitated by the proposed architecture, which (i) has a proxy at every site, to ensure a consistent information representation and network interface; (ii) uses a crawler to contact directly every site (in contrast to multi-level hierarchies of aggregation); and (iii) transforms the collected resource information to relational records, for good query performance.

The definition of resource importance is defined and evaluated in the next chapter; the proposed architecture, as a whole, is evaluated in Chapter 6.

This chapter includes material that has been published in [ZS04].

Chapter 5

A Query-Independent Definition of Resource Importance

The architecture for large-scale grid monitoring that was presented at the previous chapter can use a quantitative definition of resource importance to differentiate the rate of updates across sites. This chapter complements the architectural overview by presenting a definition of resource importance, which allows to reduce the importance of complex resources to a single number. To demonstrate the practical use of the proposed definition of resource importance, the chapter also presents an empirical evaluation of the definition for ranking clusters and sites.

The aim of the empirical evaluation is to demonstrate that the ranking that is yielded by the resource importance definition is sensible. In the context of clusters, we assume "sensible" to mean that larger and faster clusters are more important. It is not clear though how the two factors (host and network speed on the one hand, host capacity[1] and cluster size on the other) should be weighed. Consider, for instance, the following cluster configurations (assume that any properties that are not mentioned are equal):

- A host with a 4 GHz processor and 512 MB RAM, or a host with a 1 GHz processor and 2 GB RAM.
- A 20-node cluster of Pentium4-based hosts or a 100-node cluster of Pentium2-based hosts.

[1] Host capacity refers to host-level properties such as number of CPUs, and amount of primary and secondary memory.

In both cases, the answer to which configuration is more important is problem-specific, i.e., users with different needs may answer differently. Additionally, there is no immediately obvious way to quantify the importance of qualitative properties, such as the geographic locality of a grid site, or the type of middleware via which a resource is available. The proposed resource importance definition addresses these matters by taking into account the supply of and demand for specific resource characteristics, at a global level.

The present chapter makes the following contributions:

- A query-independent definition of resource importance, where importance is a measure of usefulness as expressed by users in resource discovery queries. Resource importance is intended to capture the user-perceived usefulness of a resource in relation to other resources; the importance of a given resource is meant to be used as a relative measure. The definition is applicable for complex resources, ranging from very specific, such as a CPU, up to composite and abstract, such as a site that hosts several clusters. The definition is query-independent in that importance is calculated without reference to a specific query.

- An empirical evaluation of two variations of the resource importance definition, in the context of ranking clusters.

This chapter assumes a basic familiarity with RDF (Section 4.1, page 78). The chapter is structured as follows. Section 5.1 discusses related work. Section 5.2 presents the proposed definition of resource importance, along with a variation of the definition, and implementation remarks. Section 5.3 presents an empirical evaluation of the definition in various settings. Section 5.4 concludes the chapter.

5.1 Related Work

Reference [TD07] proposes *SiteRank*: a way to rank grid resources based on low-level performance metrics and a user-specified ranking function, which is meant to describe an application's requirements in terms of performance metrics. In contrast to the importance definition in this chapter, SiteRank is application-specific and thus query-dependant. The two proposals are complementary, in that SiteRank focuses on computational performance, whereas the importance definition can use any qualitative and quantitative resource information that is

available (which, in principle, can include SiteRank rankings for typical classes of applications).

Quantifying resource importance and ranking have been extensively studied in the context of the world wide web, and its more recent evolution, the semantic web. Given that most queries in web search engines result in thousands or even millions of matched resources, it is crucial that search engines rank matches based not only on relevance but also importance. The importance of web resources is typically estimated based on analysis of the link structure of the web graph, in which a node is a web page and a directed edge is a link from one page to another [LM05]. Two seminal approaches to link-based analysis are HITS and PageRank. HITS [Kle99] distinguishes web resources in authorities and hubs. An authority is a page with many incoming links. A hub, on the other hand, is a page with many outgoing links. The HITS algorithm is intended to be run specifically for a given query. PageRank [PBMW99] determines web page importance not only by the number of incoming links but also by the importance of the pages that contain those incoming links. Also, PageRank differs from HITS in that it is query-independent.

The proposed definition of resource importance differs from the aforementioned methods as follows. First, in our case a resource is an complex entity, as opposed to a web page. Second, resources are described in disjoint documents, thus the approach of link structure analysis is not applicable. The definition proposed in this chapter is similar to PageRank in that it is query-independent.

Resource ranking is also actively studied in the area of the semantic web [BLHL01]. The semantic web is an extension of the existing web in that web resources are annotated using RDF statements (i.e., triples) about some specific domain. One of the promises of the semantic web is to enhance the precision of search engine results, by describing knowledge and facts in a well-specified and unambiguous way. The original web is a graph of interlinked documents, whereas the semantic web is a graph of interlinked triples. A query result in the semantic web is a set of sub-graphs of triples, in which every sub-graph describes a resource match. Similarly to the web, the problem is to determine the ordering of query matches. References [AH06, BM05] define query-independent resource importance by adapting link structure analysis techniques from the web, i.e., based on the number and weight of links that point to a RDF resource.

Others (e.g., [HAMAS04, SRS03]) define ways to rank query results (i.e., query-dependent importance) based on result-specific features, such as the relevance of the matched resources and or properties (e.g., student writes a report on a subject, versus an academic conducting research on the same subject).

The proposed definition of resource importance differs from the approaches in the semantic web in that, as mentioned above, the resources are described in disjoint documents, as opposed to interlinked RDF triples in the semantic web. Furthermore, we assume that all documents conform to a single schema (e.g., the adapted GLUE information model, as mapped to RDF), whereas in the semantic web anyone can make assertions about anything (e.g., one can publish on the web RDF triples about arbitrary domains using a custom or a standard schema).

5.2 A Query-Independent Definition of Resource Importance

Grid resources are diverse, and thus difficult to describe and characterise in terms of their usefulness to users. This section presents a definition of resource importance, which aims to capture the usefulness of resources as typically perceived by end users. Specifically for computer clusters, the assumption about user-perceived importance is that faster nodes and larger clusters are preferred. In addition, service affordability is defined based on the conditions under which a service is made available, if any. Both concepts are quantified in the range $[0, 1]$, where 1 denotes the highest possible importance (or affordability).

The proposed definition of resource importance is applicable to any hierarchical information model that is structured in resource-property-value triples, where a value may be a literal or a reference to another resource. This fits naturally to the RDF information model, except for the restriction that the information model must be hierarchical. The definition is also applicable to hierarchical relational schemas. Specifically, the RDF concepts resource-property-value correspond to the relational concepts table-column-value, where the value is a literal or a reference to another record. Thus, a resource in RDF along with all its properties corresponds to a record in the relational model. Conversely, a relational record of N columns that are scalars (i.e., not references to other records) corresponds to N RDF triples that have the same subject.

The remainder of this section refers to concepts of the adapted GLUE model

(Section 4.4), but the resource importance definition is applicable to any hierarchical schema.

5.2.1 Definition of Resource Importance

Recall that in the RDF information model, every resource has zero or more properties. Every property value is either a literal or a reference to another resource. Properties which take literal values (as opposed to references) are distinguished in qualitative and quantitative. This section defines the importance of:

- values of resource properties that are quantitative;
- values of resource properties that are qualitative;
- resources (i.e., RDF objects), as defined by their properties.

Importance of values of quantitative properties

The importance of the value of a quantitative property, such as nominal bandwidth or RAM capacity, *is proportional to the value itself*. Given the minimum value, min, and the maximum value, max, of a quantitative resource property R_p with a set of known values V, the importance of a value v of that property is (for clarity min, max refer to $min(V), max(V)$ respectively):

$$I(R_p, v) = \begin{cases} \frac{v-min}{max-min}, \text{ if } max \neq min \\ 0, \text{ otherwise} \end{cases} \tag{5.1}$$

for properties where higher values are more important, or

$$I(R_p, v) = \begin{cases} 1 - \frac{v-min}{max-min}, \text{if } max \neq min \\ 0, \text{ otherwise} \end{cases} \tag{5.2}$$

for properties where lower values are more important (such as latency).

Importance of values of qualitative properties

The importance of the value of a qualitative property is based on supply and demand. The *supply* of a value v for a qualitative resource property R_p (e.g., CPU model is Opteron) is the probability that $R_p = v$. In other words, if we consider all resources for which the property R_p is relevant, this probability indicates how

many of those resources have the property $R_p = v$. Such probabilities are drawn from the resource information that is collected about all sites. The probabilities are periodically updated to reflect changes in the supply of resources throughout the grid over time. The *demand* for a value v of a property R_p is the probability that a user will search for the value v whenever R_p is used in a query. The calculation of demand requires the availability of a user query log file. Note that user demand measures the demand for every known value of qualitative resource properties, as opposed to the demand for particular resource instances that have those values (e.g., the demand for Opteron CPUs, instead of the demand for a particular host with an Opteron CPU).

Let V be the set of known values of the qualitative resource property R_p; let demand(v) and supply(v) be the probabilities for demand and supply for the value $v, v \in V$. The following steps are applied:

1. the difference between demand and supply is calculated for all values of V; the new set of values is referred as V_a:

$$v_a = \text{demand}(v) - \text{supply}(v), \forall v \in V \tag{5.3}$$

2. normalise every value v in V_a as follows (min, max refer to $min(V_a), max(V_a)$ respectively):

$$I(R_p, v) = \begin{cases} \frac{v-min}{max-min}, \text{ if } max \neq min \\ 0, \text{ otherwise.} \end{cases} \tag{5.4}$$

Importance of resources

The importance of a resource is the weighed average of the importance of its properties; the result is normalised using Equation 5.4. (The normalisation is necessary to convert all values in the range [0,1].) This definition is recursive, as a resource property's value may be another resource. Unless otherwise specified, equal weights are used. A notable exception is the cluster's importance, calculated as the weighed average of the importance of its subclusters, in which the weight is the number of hosts in every subcluster, normalised by the total number of hosts in that cluster.

The definition of importance is applied bottom-up the hierarchy of resources and resource properties. For instance in Figure 4.4 (page 91), one would first

calculate the importance of "leaf" resources (ApplicationSoftware, OperatingSystem, LocalStorage, Processor), and then proceed with the importance of those resources' parents (Software, Hardware), and so on, all the way up to the hierarchy's root (Site). Based on the level at which importance is calculated, one can informally refer to, for instance, host, cluster service, and site importance.

Qualitative properties: to use or not to use

Consider a resource that has both qualitative and quantitative literal properties, and the quantitative properties suffice to capture the resource's importance. An example would be a hard disk that has the following properties: controller type (e.g., IDE, SCSI); read rate and write rate (the latter two being measured in Mbyte/sec, i.e., quantitative). From the user's perspective, in general, whether the disk is connected via IDE or SCSI is irrelevant as long as it can deliver a minimum required read/write performance. Thus, one can reason that certain qualitative properties are made redundant by the existence of relevant quantitative properties.

A user may specifically look for a host with a SCSI disk, but this is not the general case. The definition of resource importance is intended to serve the general case. Users with special requirements are expected to be explicit while submitting queries (e.g., include a condition that disk type must be SCSI).

Taking into account qualitative properties when there is sufficient quantitative information can affect the results in undesirable ways. For instance, assume that IDE disks are by far more common than SCSI disks and that there is zero user demand for both disk types (e.g., because users do not specify the required disk type in queries). This means that, given an IDE and a SCSI disk, the SCSI disk will have higher importance when both disks have equal performance, or even when the SCSI disk is slower up to some degree (the degree being determined based on the difference of supply of the two disk types).

To avoid this problem, we consider a variant of the resource importance definition, in which the qualitative literal properties of a given resource are not taken into account (i.e., have zero weight) when the resource has literal quantitative properties that can be considered sufficient to capture the resource's importance. Exactly which qualitative properties have to be ignored must be determined manually for a given information schema.

Dealing with incomplete data

Resource providers may choose not to provide all the data that is prescribed in a resource information schema. This can happen for various reasons, such as negligence, or some information being considered not useful. This is not a problem to the extent that an information service implementation includes scripts for automatic generation of resource data. But often administrators have to write custom scripts or manually enter some information.

As a result, any program that processes resource information has to deal with incomplete resource descriptions. The definition of resource importance deals with this problem by using a default importance value for resource properties for which no information is available. The default importance value is zero (i.e., minimum importance) to (i) encourage the publication of resource information; and (ii) discourage the intentional concealment of poor resource characteristics (which would normally yield low but, most probably, higher than zero importance).

5.2.2 Definition of Service Affordability

The more users a cluster service is available to, and the less the price for using it, the more affordable the cluster service is. A cluster service is available via a set of sharing options P, where each sharing option $i \in P$ specifies a price $price_i$ and a number of authorised users $users_i$. We define the affordability of a cluster service as:

$$A(cs) = \sum_{i \in P} \left(w_i \times \frac{\frac{\text{users}_i}{\text{users}_{\max}}}{\frac{1+\text{price}_i}{\text{price}_{\max}}} \right) \tag{5.5}$$

where

- $w_i = \text{users}_i / \sum_{i \in P} \text{users}_i$;
- $\text{users}_{\max}$ is the maximum known number of users a service is available to;
- $\text{price}_{\max}$ is the maximum known price of a service.

One could fake high affordability by defining two sharing options, one with a low price but available only to a few users, and another with a high price available to many users. This is why every sharing option's users/price ratio is weighed by w_i.

Applicability issues

The given definition of service affordability makes the following assumptions.

- *Resource providers can estimate the number of authorised users per sharing option.* Such an estimation is straightforward when the authorised users are actual users (e.g., Nick, John and Mary) as opposed to roles (e.g., members of research groups of the University of Manchester). Estimation in the latter case may require extra administrative effort, which may be considered unacceptable by some resource providers.

- *Resources are offered with a fixed price.* For instance, the S3 service of amazon.com [Ama] provides storage space with fixed pricing per GB that is stored, and per GB of bandwidth that is used for uploading and downloading to and from the service. In contrast, some supercomputing sites adjust pricing based on the utilisation level of their resources. Thus, prices fall when a site's resources are idle, while they increase when there is a lot of demand. In those cases, it is not obvious how to calculate affordability, as resource providers may use different strategies to adjust prices based on the utilisation levels of their resources. At the very least, it would be sensible to assume low affordability for sites that are known to be overloaded (i.e., where all resources are in use and there is a long queue of jobs waiting to be allocated resources).

- *Resource providers are willing to provide sharing options' details to monitoring sites.* This may not be the case when resource providers consider the information about pricing and authorised users as proprietary. Grid sites that choose not to provide information about sharing options may need to be penalised with low affordability.

5.2.3 Implementation Remarks

The calculation of resource importance involves two phases. In the first phase, the importance evaluator: (i) calculates the probability of all the known values of every qualitative literal property in the adapted GLUE model; (ii) finds the minimum and maximum values of every quantitative property; and (iii) calculates user demand, for all known <qualitative property, value> combinations, based on query logs. In the second phase, the importance evaluator calculates

importance starting from the lowest-level resources (e.g., the most indented concepts in Figure 4.3, page 90) and proceeding upwards the hierarchy. The reason for the bottom-up direction is that a resource's importance depends on that of its offspring. Importance can be calculated up to any level of the hierarchy. As previously mentioned, based on the level at which importance is calculated, one can informally refer to, for instance, host, cluster service, and site importance.

Recall that resource importance is the weighed importance of a resource's properties. Properties that are purely informational, such as a site administrator's email or a service's address, are assigned a zero weight as they do not affect the actual usefulness of a resource's importance. Also, anything related to cluster services' sharing options is ignored, since this information is taken into account as part of service affordability.

5.3 Using Resource Importance for Ranking

This section characterises the importance of a number of synthetic site and service descriptions as a means of evaluating the definition of resource importance. The aim of the evaluation is to demonstrate that the definition characterises the importance of resources in a way users would expect. Specifically for clusters, we assume that users perceive importance as proportional to the speed (e.g., CPU clock frequency) and capacity (e.g., RAM size) of cluster nodes, and the total number of nodes in a cluster.

To assess the definition's results, we consider a baseline site description with one service and given characteristics, in three scenarios. In every scenario, all sites initially have the baseline setting, that is, all sites are identical, and thus all properties only have a single value each. Thus, for every quantitative (resp. qualitative) property, $min = max$ in Equation 5.1 (resp. Equation 5.4), and as a result the importance of every site is zero. Certain properties are assigned different values in every scenario, to assess the impact they have to importance.

In the first two scenarios site descriptions are generated that differ from the baseline setting in only one property. The distinguishing property is quantitative in one scenario and qualitative in the other. These two scenarios demonstrate that the definition yields sensible results for the simplest cases.

```
Site
    SiteAvailability (0.1)
    InetAccess (0.1)
      hasNominalBandwidthMbps (1)
    Location (0.1)
    Services (0.7)
        Service
            Policy (i.e., affordability: 0.3)
            LRMS (0.2)
              hasLRMStype (0.5)
              hasLRMSversion (0.5)
            ComputingElement (0.5)
```

Figure 5.1: A subset of the resource information model, in terms of resources and properties, and the weights (indicated in parentheses) that are used in the case studies.

The third scenario concerns a more realistic case where synthetic site descriptions vary from the baseline setting in 7 service properties, of which 2 are qualitative. This scenario is more realistic than the first two in that the varied properties are assigned values based on published analytical models.

The exact details of the baseline setting are not significant because resource importance is not affected by property values that are identical across all sites. For instance, consider that the baseline setting specifies (and thus all sites have) the value "Manchester/UK" for the "SiteLocation" property. On this basis, the property "SiteLocation" is irrelevant to the importance-based ranking of sites because the importance of SiteLocation="Manchester/UK" is the same for all sites.

A clarification of the weights used is in place. Figure 5.1 shows a subset of the resource information model, with a focus on the resources and properties that are relevant to the present evaluation.[2] According to the weights indicated in the figure, the importance of site and service resources are calculated as follows (in the following, "imp" denotes importance):

$$\begin{aligned} \mathrm{imp}_{\mathrm{site}} = \quad & 0.1 \times \mathrm{imp}_{\mathrm{SiteAvailability}} + 0.1 \times \mathrm{imp}_{\mathrm{InetAccess}} + \\ & 0.1 \times \mathrm{imp}_{\mathrm{Location}} + 0.7 \times \mathrm{imp}_{\mathrm{Services}} \end{aligned} \tag{5.6}$$

[2]The notation of Figure 5.1 has been explained in Section 4.4: concepts that are one level of indentation apart have a parent-child relation; concepts at the same level of indentation are siblings. Names starting with "has" indicate properties that take literal values.

site bandwidth in Mbps	value importance
1	0.0000
2	0.0666
4	0.2000
8	0.4666
16	1.0000

Table 5.1: Importance values for a quantitative property.

$$\begin{aligned} \mathrm{imp}_{\mathrm{service}} = \quad & 0.3 \times \mathrm{imp}_{\mathrm{affordability}} + 0.2 \times \mathrm{imp}_{\mathrm{LRMS}} + \\ & 0.5 \times \mathrm{imp}_{\mathrm{ComputingElement}}. \end{aligned} \tag{5.7}$$

The importance of the Local Resource Management System (LRMS) is simply the average of its two qualitative properties:

$$\mathrm{imp}_{\mathrm{LRMS}} = \quad 0.5 \times \mathrm{imp}_{\mathrm{LRMStype}} + 0.5 \times \mathrm{imp}_{\mathrm{LRMSversion}}. \tag{5.8}$$

5.3.1 Simple Scenario With One Quantitative Property

In the first scenario, 5 site descriptions are generated that differ from the baseline setting only with respect to one quantitative property: the nominal bandwidth of the site's Internet connection. The 5 sites are assigned bandwidth values of 1, 2, 4, 8, and 16 Mbps, respectively. Table 5.1 shows the importance of every bandwidth value (which is identical to the importance of sites with that bandwidth, because all 5 sites differ only in terms of bandwidth). Nominal bandwidth is a quantitative property in which higher values imply more importance. The value importance column shows that the the minimum and maximum bandwidth values have 0 and 1 importance, respectively, and the importance of all other bandwidth values is normalised according to Equation 5.1. For instance, the importance of 8 Mbps bandwidth is $\frac{8-1}{16-1} \simeq 0.4667$.

5.3.2 Simple Scenario With One Qualitative Property

In the second scenario, we generate 100 site descriptions that differ from the baseline setting only with respect to one qualitative property: the LRMS version of the only cluster service every site hosts. The LRMS versions and their (randomly generated) frequencies of occurrence are shown in Table 5.2, along with

value	supply	value importance
D	0.52	0
C	0.26	0.5306
B	0.19	0.6734
A	0.03	1

Table 5.2: Importance values for a qualitative property based only on supply.

the corresponding importance values. The demand for the considered LRMS versions is zero. In other words, for the purpose of the scenario being simple, it is assumed that users do not use LRMS version as a condition in resource discovery queries. For instance, the importance of "LRMSversion=B" is calculated as follows: $v_a = 0 - 0.19 = -0.19$ and $\frac{-0.19-(-0.52)}{-0.03-(-52)} = 0.6734$ (where -0.52 and -0.03 are, respectively, the minimum and maximum importance values before normalisation).

5.3.3 Realistic Scenario With Various Properties

The first two scenarios demonstrated that the definition of resource importance yields reasonable results for the simplest cases. In this scenario we consider 100 site descriptions that differ from the baseline setting in 7 properties. Namely, 2 qualitative properties (processor model, LAN technology), and 5 quantitative (processor clock frequency, L1 cache size, number of processors per host, RAM size, and number of hosts per cluster). The data are generated using an implementation of analytical models published in [KCC04]. Specifically, the supply values of processor models in Table 5.3 are derived based on the sample probabilities in Table 1 in [KCC04]. The probabilities in [KCC04] were derived from a study of over 10,000 processors in high-performance clusters. Every processor model is characterised in terms of a performance factor. The performance factor of a processor model is used to tune the analytical models that determine the values of host-level quantitative properties. (For instance, a high-end processor is more likely to have large RAM capacity, compared to a low-end processor.)

For the purpose of this discussion, we distinguish processor models in low, mid and high-end (listed in order of increasing performance): (i) Celeron, Pentium2, Pentium3; (ii) AthlonMP, AthlonXP, Opteron, Pentium4; and (iii) Itanium. This ranking is taken from Table 2 in [KCC04] and has been derived based on floating point performance. The processor model is a strong performance indication, but

value	supply	value importance
Pentium3	0.43	0
Pentium4	0.28	0.3571
AthlonMP	0.14	0.6904
Celeron	0.07	0.8571
AthlonXP	0.03	0.9523
Opteron	0.03	0.9523
Itanium	0.01	1
Pentium2	0.01	1

Table 5.3: Supply, and importance of the values for processor model based only on supply.

overall performance can be significantly affected by other processor properties. For instance, it is possible that a Pentium3 processor may be better than a Pentium4, for specific applications, if it has more cache or more processors on chip. In addition, a cluster of Pentium3 hosts may be more important than one with Pentium4 hosts if the former has a sufficiently larger number of hosts.

Table 5.3 shows the importance of every processor model, based only on supply. (That is, assuming that user demand for all processor models is the same.) The column "value importance" in the table is calculated using Equations 5.3-5.4. Observe that these importance values suggest misleadingly that AthlonMP, Pentium3 and Pentium4 processors are less important than Celeron processors, because they are more common.

Importance values for all 100 sites have been calculated in two ways, hereafter referred as A0 and A1:

A0 always using qualitative properties (more precisely, using the only qualitative property ("processor model") that is varied in the present scenario);

A1 not using the qualitative property "processor model" on the assumption that the processor resource is sufficiently characterised by its quantitative properties.

Tables 5.4 and 5.5 show the details of the top and bottom 10 clusters as ranked in A0. LAN technology is not shown as the property takes two values (Ethernet and Myrinet) that are equally likely, thus does not affect importance. The top 10 clusters (Table 5.4) are based on mid-end processors, with the exception of one Celeron-based cluster. The top cluster (id 13) has moderate processor clock frequency and number of hosts but top settings for cache, RAM and number of

id	importance	model	clock frequency	L1 cache	processors	RAM	cluster hosts
13	1	Pentium4	1745	1024	4	16384	32
74	0.9263	Pentium4	3355	1024	4	4096	20
2	0.8200	Opteron	3125	1024	2	512	5
10	0.8170	Opteron	2780	1024	2	2048	8
67	0.7858	Opteron	1860	1024	2	4096	59
60	0.6959	Pentium4	1745	512	4	8192	23
62	0.6882	AthlonXP	3585	256	2	4096	27
85	0.6830	Pentium4	2205	1024	2	4096	178
95	0.6687	Celeron	1450	256	4	4096	56
98	0.6658	Pentium4	3585	512	2	8192	5

Table 5.4: Importance of the top 10 clusters (A0).

processors. The results do not seem to be significantly affected by the number of hosts per cluster. For instance, the third Opteron cluster (id 67) has more than four times the hosts that the two other Opteron clusters have together (ids 2 and 10), yet the former is ranked lower because of lower clock frequency.

The clusters in the bottom 10 list (Table 5.5) certainly have worst capabilities in terms of cache size, number of processors per host, RAM size, and number of hosts. At first, it may seem surprising to see that all of the bottom 10 clusters have Pentium3 processors, instead of the supposedly inferior Celeron and Pentium2 processors. Table 5.6 lists the details and importance of all the clusters with Celeron and Pentium2 processors, at least a few of which one would expect to be in the bottom 10 list. Looking at Tables 5.5 and 5.6, it becomes apparent that the bottom-10 clusters with Pentium3 processors are actually worst than many clusters with Celeron and Pentium2 processors. This is because clusters with Pentium3 processors are far more common than clusters with Celeron and Pentium2 processors, thus is more likely that there are more clusters with low-end Pentium3 processors than clusters with low-end Celeron processors. Yet there are cases in which the ranking is not sensible. For instance, the last cluster in Table 5.6 (id 48) is not in the bottom 10 list, even though it is worst than the last cluster in Table 5.5 (id 38) in all quantitative properties (except RAM size, which is 1024 MB for both). This is due to the effect of Pentium3 processors being in high supply.

Next, we consider the importance values for the same 100 sites in A1. Tables 5.7 and 5.8 show the details and importance of the top and bottom 10

id	importance	model	clock frequency	L1 cache	processors	RAM	cluster hosts
90	0	Pentium3	650	256	1	256	16
27	0.0409	Pentium3	1100	256	1	512	5
53	0.0585	Pentium3	1200	256	1	1024	5
24	0.0679	Pentium3	1300	256	1	1024	7
51	0.0764	Pentium3	400	512	1	1024	34
1	0.0796	Pentium3	550	512	1	512	30
93	0.0898	Pentium3	500	256	2	1024	3
29	0.0953	Pentium3	650	256	2	512	7
84	0.0967	Pentium3	600	512	1	128	98
38	0.1011	Pentium3	600	256	2	1024	12

Table 5.5: Importance of the bottom 10 clusters (A0).

id	importance	model	clock frequency	L1 cache	processors	RAM	cluster hosts
95	0.6687	Celeron	1450	256	4	4096	56
76	0.5929	Celeron	1300	128	4	2048	101
61	0.5684	Celeron	3000	256	2	2048	5
77	0.5005	Pentium2	850	512	2	2048	9
32	0.4421	Celeron	2150	128	2	1024	27
99	0.3877	Celeron	2200	256	1	1024	8
12	0.2408	Celeron	550	256	1	1024	2
48	0.1892	Celeron	400	128	1	1024	8

Table 5.6: Importance of clusters with a Celeron or Pentium2 processor (A0).

clusters, respectively. Comparing the two top-10 clusters (Tables 5.4 and 5.7), one can see that the AthlonXP- and Celeron-based clusters, and one of the three Opteron-based clusters in A0 are not in the top-10 list of A1. They are ranked high in A0 because of AthlonXP, Celeron and Opteron being processor models with limited supply. Other than that, the relative ranking of clusters with the same processor is identical in both A0 and A1.

The A1 bottom-10 list (Table 5.8), has three entries that were not in the respective A0 list: two Celeron- and one AthlonMP-based clusters. Again, these three clusters were treated preferentially due to being based on processor models in short supply. A1 results in a sensible ranking by ignoring the supply for processor models, as they are adequately described by quantitative properties.

id	importance	model	clock frequency	L1 cache	processors	RAM	cluster hosts
13	1	Pentium4	1745	1024	4	16384	32
74	0.9225	Pentium4	3355	1024	4	4096	20
60	0.6802	Pentium4	1745	512	4	8192	23
85	0.6667	Pentium4	2205	1024	2	4096	178
98	0.6486	Pentium4	3585	512	2	8192	5
59	0.6423	Pentium4	2895	1024	2	2048	7
2	0.6352	Opteron	3125	1024	2	512	5
10	0.6320	Opteron	2780	1024	2	2048	8
55	0.6134	Pentium4	1975	256	4	8192	10
63	0.6064	Pentium4	1975	1024	2	2048	172

Table 5.7: Importance of the top 10 clusters (A1).

id	importance	model	clock frequency	L1 cache	processors	RAM	cluster hosts
48	0	Celeron	400	128	1	1024	8
90	0.0537	Pentium3	650	256	1	256	16
12	0.0542	Celeron	550	256	1	1024	2
27	0.0967	Pentium3	1100	256	1	512	5
88	0.1127	AthlonMP	1260	256	1	512	9
53	0.1152	Pentium3	1200	256	1	1024	5
24	0.1251	Pentium3	1300	256	1	1024	7
51	0.1341	Pentium3	400	512	1	1024	34
1	0.1375	Pentium3	550	512	1	512	30
93	0.1481	Pentium3	500	256	2	1024	3

Table 5.8: Importance of the bottom 10 clusters (A1).

5.4 Closing Remarks

This chapter contributed a query-independent definition of resource importance, which allows to reduce the importance of complex resources, ranging from processors to whole grid sites, to a single number. The chapter also presented an empirical evaluation of the resource importance definition when applied for ranking clusters. Due to the lack of an established benchmark for ranking clusters, the evaluation was limited to an empirical proof-of-concept demonstration. Two variations of the resource importance definition were evaluated. One that takes into account qualitative properties at all times, and another that ignores qualitative properties of resources that are sufficiently described by quantitative properties (labelled A0 and A1, respectively). In the considered scenarios, A1 produces more

sensible results than A0, in that it ignores the supply and demand of processor models, on the basis that the importance of processor models is captured by other quantitative properties. Nevertheless, the concept of supply and demand is still required for capturing the importance of certain qualitative properties that cannot be captured by other quantitative properties, such as Local Resource Management System (LRMS) type and version.

The core idea of the resource importance definition is simple, but a real-world application would require a fair amount of tuning. This is normal as, for instance, Google's PageRank algorithm [PBMW99] despite being conceptually simple is tuned based on hundreds of heuristics (e.g., URL length). The tuning of the resource importance definition for real-world use would mainly involve appropriate weights for resource concepts in the information model, e.g., to balance the significance of a site's location against, say, its availability record.

Chapter 6

Evaluation of the Proposed Architecture

The previous two chapters, among others, described the proposed architecture and the definition of resource importance, which is used to determine the trade-off between information freshness and network overhead on a site-level basis. This chapter presents an experimental evaluation of the proposed architecture based on a prototype implementation. The aims of the chapter are to (i) assess the performance trends of the proposed architecture for various problem settings and grid sizes in realistic conditions; and (ii) examine the problem settings in which the proposed architecture would be more appropriate than just-in-time hierarchies of aggregation (JITHAs).

The present chapter contributes:

- an experimental evaluation of the proposed architecture, based on realistic deployments of the prototype, to measure performance trade-offs in various problem settings;
- a comparison, in terms of query response time and network overhead, between the proposed architecture and JITHAs, based on the results obtained in Chapter 3;
- a comparison of experimental results against estimations obtained using the performance models in Section 4.7.

The chapter is structured as follows. Section 6.1 describes implementation

issues related to the prototype, for the purpose of reproducibility. Such issues include the way configuration, deployment and shutdown of experiments is handled; the modelling issues involved in the generation of synthetic resource information by proxies; and the way the performance metrics are measured. Section 6.2 investigates the performance behaviour of the prototype implementation, i.e., measures the trade-offs of network overhead, information freshness and query response time, in various points of the evaluation space, including different grid sizes. Section 6.3 compares selected results from Chapter 3 for JITHAs, against those of the prototype. Section 6.4 evaluates the previously described performance models (Section 4.7) against experimental results. Section 6.5 concludes the chapter.

6.1 Prototype Implementation and Deployment Issues

The crawler, data manager, importance evaluator, proxies, database front-end and query generator are implemented in Java using JDK version 1.5. The programs that interact with any of the two databases (namely, crawler, data manager, importance generator, and database front-end) use MySQL Connector version 3.1.8. The monitoring site's crawler, data manager, importance evaluator and databases are deployed across 4 hosts, as shown in Figure 4.1, page 83. These hosts are located at the University of Manchester, and are interconnected via a 100 Mbps LAN. The main store is MySQL, version 4.1.12-standard. The PC hosting the main store is a Pentium 4 at 2.4 GHz, with 512 KB cache size, 512 MB RAM, running Linux 2.4.20-31.9.

Planetlab [PACR03] is used to deploy proxies in up to 100 hosts across N. America, Europe, Australia and Asia. Planetlab is a worldwide testbed for experimenting with Internet-scale services. Planetlab had 716 nodes over 349 sites, as of the end of 2006. Every Planetlab node is potentially used by many users, and no guarantees are made about host availability or performance (e.g., hosts may be rebooted or get overloaded at any time). As a result, the successful completion of an experiment that involves a large number of nodes is non-trivial. Planetlab is a suitable evaluation platform for the present work, because it exhibits a dynamic behaviour, similar to what one would expect from geographically distributed servers that host grid information services.

6.1.1 Experiment Launch and Shutdown

To ease the execution of experiments, we run 50 proxy instances per Planetlab host. The collocation of many proxies per host does not have any significant impact on the measured performance metrics because proxies are not computationally intensive. (Proxies' main tasks are to keep track of information regarding local resources, and to respond to fairly infrequent HTTP requests from crawlers.) For efficiency reasons, primarily memory usage, all proxy instances located in one host are running within a single Java Virtual Machine (JVM). Nevertheless, proxy instances within the same JVM are independent: they have distinct state and threads of execution, maintain a separate log file, and listen to a different TCP port.

The first task performed by every JVM is to contact the data manager (which has a thread specifically for this purpose) at the monitoring site, to measure the time difference between local time and the remote host's time (the latter being GMT). This time synchronisation is necessary for the following reasons: (i) information freshness is calculated based on timestamped logs of proxies and the data manager; (ii) Planetlab hosts are located in different time zones; and (iii) although Planetlab hosts are meant to be synchronised (using the Network Time Protocol) this is not always the case. Once the difference between local time and GMT is known, all the timestamps that have to be generated (e.g., in "hasTimestamp" properties, in log entries) are adjusted accordingly.

Let a_1 be the time at which the proxy sends the time request, a_2 be the time at which a response is received (both a_1 and a_2 being sampled from the proxy's clock), and r_1 be the time reported by the remote host (sampled from the remote host's clock). To account for turnaround time, the remote time r_1 is adjusted by the proxy as $r_2 = r_1 - \frac{(a_2 - a_1)}{2}$ and the estimated time difference between a proxy's local time and the remote time is $r_2 - \frac{(a_1 + a_2)}{2}$. This synchronisation phase is monitored manually (via the data manager's logs) to verify that no host takes unreasonably long to launch the proxy and send a synchronisation request. It is acknowledged that the fidelity of the outlined synchronisation method is not as high as that of the standard Network Time Protocol (NTP), but it suffices for the purpose of measuring average information freshness.

The launch script runs remotely a script (via non-interactive ssh) on every Planetlab host; the latter script launches the main Java program that forks a given number of proxy instances on every Planetlab host. Once all proxies are

up and running at a host, the TCP ports at which they listen to are reported to a designated host. This is required as Planetlab hosts are shared by many users, and thus it is not possible to know in advance which TCP ports are not in use at any time. After all the Planetlab hosts have reported back their TCP ports, the crawler is run to perform a full update on all proxies. Once all information for all sites has been retrieved and inserted into the database, one can then launch the actual experiment. This involves:

- Launch of the data manager for processing the RDF-encoded differential updates that are downloaded by the crawler.
- Re-launch of the crawler, this time requesting from proxies exclusively differential updates.
- Launch of the importance evaluator, regardless of whether the considered setting actually uses resource importance. The evaluator is configured to re-evaluate the importance of all grid sites every 15 minutes. This time interval may be too short; it has been chosen to impose extra load to the main RDBMS, so that the measured QRT is pessimistic.
- Initiation of resource event generation: all proxies are notified to start generating resource changes according to the specified mode (described in Section 6.1.2). The proxies implement several different modes of event generation, where every mode specifies the frequency of resource changes for every event type. Contacting all proxies individually can be particularly time-consuming for thousands of proxies. Instead, a single proxy is contacted per host, which in turn notifies the rest of the proxies at that host.
- Launch of the database front-end: a program that accepts identifiers of predefined queries, submits the associated queries to the database, and records the response time to every query. The database front-end is hosted at the same node as the main RDBMS for performance reasons.
- Launch of the query generator, which emulates a specified query workload. A query workload determines the type, number and arrival pattern of queries that should be submitted to the database front-end.

6.1.2 Resource Modelling Settings

For the purpose of the experiments, proxies do not actually collect data from deployed information services, as this would have had many practical complications (most significantly, the permission to use the information services of thousands of sites). Instead, proxies generate synthetic site profiles (i.e., information about a site and the cluster services hosted therein) in RDF/XML according to the adapted GLUE model. The generation of a proxy's site profile in the beginning of an experiment, is followed by events that denote changes in the status of cluster services.

Generation of Site Profiles

For every cluster service, a site profile includes information about (i) sharing policy; (ii) hardware and software configuration of every subcluster;[1] and (iii) the cluster's load status.[2] The information that is generated dynamically concerns the software and hardware configuration of subclusters, and the sharing options of cluster services. Everything else has predefined values based on a template. The hardware configuration that is dynamically generated is about processor model, processor clock frequency, cache size, number of processors per host, RAM capacity, number of hosts per cluster and LAN technology, as described in Section 5.3.3, based on the work of [KCC04]. For simplicity, every cluster is composed of only one subcluster. Every subcluster has up to 3 software packages, which are chosen with equal probability. Every cluster service's operating system is either Linux or Windows XP with respective probability of 0.9 and 0.1. In addition, every cluster service has a number of sharing options that follows a normal distribution with $\mu = 1, \sigma = 1$, provided that there is at least one sharing option per cluster service. Every sharing option defines the list of authorised users and the price (e.g., per hour) that applies to that user group. Instead of producing a list of users, the prototype determines the number of users (based on a normal distribution with $\mu = 5, \sigma = 10$) and the price (normal distribution with $\mu = 30, \sigma = 10$), both bounded to be at least 1. The modelling approach and parameters for sharing options and software were chosen with the sole purpose of producing further variation in service cluster descriptions.

[1] Recall from Section 4.4 that a cluster is composed of one or more subclusters.

[2] "Load status" refers to the five properties of the resource "ComputingElementState" in Figure 4.3, page 90.

Every site profile is effectively a RDF document, in which the baseURI (i.e., the URI with which the contained resources are qualified) is the proxy's URL address and TCP port. For instance, the fully qualified address for the resource SubCluster123 at proxy planetlab2.ewi.tudelft.nl that listens to port 8091 is http://planetlab2.ewi.tudelft.nl:8091/SubCluster123. The name of cluster services is formed by concatenating the resource's class (in this case SubCluster) and the time at which the resource was created (expressed in seconds since epoch time). Combining baseURI, port and resource name allows the unique identification of all resource clusters at a global level.

Resource Changes (Events) Generation

As already mentioned, for the purpose of the experiments, proxies do not actually interact with real deployments of grid information services. Instead, a separate thread in every proxy, the EventGenerator, generates events that denote changes in the status of clusters (i.e., the properties of the ComputingElementState resource). Whenever a new event occurs, the EventGenerator notifies the main proxy thread to update its RDF representation of site profile. A user-supplied event generation mode specifies the average time between consecutive occurrences of every event type. For instance, one event generation mode may specify that properties A, B, C change, on average, every 1, 5 and 10 time units respectively; another may specify that all A, B, C properties change every 5 time units. Some of the experiments have different event generation modes to measure their effect on information freshness.

6.1.3 Performance Metrics

In addition to network overhead and query response time (QRT), we also measure the level of information freshness at successive periods.

Network Overhead

In the case of the prototype, network overhead is measured as the number of bytes that the crawler downloads during an experiment (Equation 4.2, page 93). With reference to the prototype, "network overhead" will be qualified as "total" or "diff" to denote whether it refers to the network overhead of both full and differential updates, or only differential updates, respectively.

In the case of JITHAs, network overhead is calculated according to Equation 3.2 (page 63), where $v_{i,j}$ is the top-level node. In other words, network overhead in JITHAs is the number of bytes collected by the top-level node of a monitoring hierarchy.

Average Information Freshness

The prototype keeps timestamped logs of resource changes in every proxy, and of database updates by the data manager. At the end of an experiment, all proxy logs are collected at a single host. By processing the proxies' logs and the data manager's logs, a script calculates information freshness for every property of every resource, at consecutive periods of 1 minute. Information freshness is calculated as the percentage of resource properties that are up to date at a given point in time, as defined in Equation 4.3 (page 94). Based on the aforementioned values of information freshness at given points in time, one can then calculate time-average or simply average information freshness, according to Equation 4.4 (page 94). The results reported in this chapter refer to average information freshness.

The number of resource properties that change during experiments is fixed to 5. Calculating the freshness of all properties produces significantly high freshness values, as most properties do not change during experiments. Instead, only the 5 properties that do change during experiments are taken into account in the calculation of information freshness.

Average Query Response Time

As described in Section 4.6.3, QRT in the case of the prototype is measured as the server-side cost, i.e., the interval from the receipt of a query identifier by the database front-end until the time at which the query results are ready to be sent back to the query generator. The database front-end keeps a log-file with the response times of the queries accepted during an experiment. The average of the set of QRT values from the database front-end's log-file is what is referred in this chapter as the average QRT of a given experiment.

In the case of JITHAs, QRT is estimated using Equation 3.1 (page 63).

6.2 Exploring the Evaluation Space

This section describes the problem settings that were considered in the experiments, performs a few simple tests to verify that the experiments were run as intended, and presents and discusses the experimental results.

6.2.1 Considered Settings

The evaluation is carried out for several problem settings that are defined in terms of the following parameters: grid size (i.e., number of grid sites and cluster services per site); event generation mode (i.e., frequency of changes per event type); frequency of updates; total number of queries and distribution of query arrivals; and query complexity and selectivity. The evaluation considers 20 different settings: a baseline setting, and 19 settings that differ from the baseline only in terms of one parameter. At least three values are considered for every parameter, so that one can see the performance trend for every parameter.

The rationale for choosing parameter values is as follows. The baseline represents the average case, and at least two more values are considered for every parameter: one more and another less demanding than the baseline setting. The exact parameter values that are chosen are not particularly important, as long as they represent a reasonably wide problem space, and are indicative of large grids in the future. As such, some of the considered parameter values are not necessarily indicative of current grid deployments. For instance, as of mid 2007, most grids have only up to a few tens of grid sites (e.g., TeraGrid [ter] has 23 clusters across approximately 10 grid sites.) In contrast, the prototype is evaluated for grid settings ranging from 500 up to 5000 sites.

The baseline setting has the following parameter values:

grid size 2000 grid sites and 5 cluster services per site;

event generation mode resource changes for 5 event types occur on average every 1, 2.5, 5, 10, 20 minutes, respectively; for instance, property A changes on average every 1 minute, property B every 2.5 minutes, and so on;

update frequency all sites are visited by the crawler every 5 minutes;

number of queries and query arrivals 4000 queries uniformly distributed over the experiment duration;

query complexity and selectivity high complexity query with high selectivity (described below).

To investigate performance trends across the above parameters, several variations of the baseline setting are considered (Table 6.1). In the "setting id" column of the table, every parameter is denoted with a given letter; every value that is considered for a parameter is indicated with a distinctive number. The baseline value for a given parameter is indicated using the number zero, e.g., if s indicates the grid size parameter, $s0$ indicates the baseline setting for grid size.

setting id	value of varied property
grid size (number of grid sites and cluster services per site)	
s1	500 grid sites; 1 cluster service per site
s2	500; 5
s3	500; 10
s4	2000; 1
s0	2000; 5
s6	2000; 10
s7	5000; 1
s8	5000; 5
s9	5000; 10
average frequency of resource changes (per event type)	
r1	all 5 event types every 1 minute
r0	every 1, 2.5, 5, 10, 20 minutes, respectively
r3	all 5 event types every 10 minutes
update frequency of grid sites	
u1	all sites every 1 minute
u0	all sites every 5 minutes
u3	all sites every 15 minutes
u4	10% of sites every 5 minutes; 90% of sites every 15 minutes
u5	determined proportionally to site importance in the interval $[1, 15]$, according to Equation 4.1 in Section 4.3.2
number of queries and query arrivals	
q1	2000 queries uniformly distributed
q0	4000 queries uniformly distributed
q3	2000 queries distributed uniformly over 10 bursts
q4	4000 queries distributed uniformly over 10 bursts
query complexity and selectivity	
c1	low-complexity queries
c0	high-complexity queries with high selectivity (40%)
c3	high-complexity with low selectivity (10%)

Table 6.1: The set of problem settings considered in the evaluation, in terms of how they differ from the baseline setting.

For grid size, we considered 500, 2000 and 5000 sites.[3] Every possible value for the number of grid sites was considered with 1, 5 and 10 cluster services per site (s1-s9 in Table 6.1).

In terms of the frequency of resource changes, in addition to the baseline, two more cases were considered: that all event types occur every 1 minute (r1) or every 10 minutes (r3). Proxies were queried by the crawler for differential updates, every 1 or 15 minutes in the u1 and u3 settings. Resource importance was considered in the u4 and u5 settings (i.e., topn and proportional modes, as described in Section 4.3.2, page 85).

With respect to query workload, we considered 2000 or 4000 queries uniformly distributed throughout the experiment duration (q1 and baseline settings, respectively). In settings q3 and q4, queries are evenly distributed in 10 bursts. For instance, in q3 (resp. q4) every burst consists of 200 (resp. 400) queries. The bursts are evenly distributed across the experiment duration, with the restriction that the last burst must occur 5 minutes before the end of the experiment. The queries of a burst that starts at time t are scheduled at time $t + d$ where d is normally distributed with $\mu = 0, \sigma = 30$ seconds.

The queries used in the evaluation are listed in Figure 6.1. It is assumed that query complexity is affected by the number of conditions in a SELECT query and the potential use of aggregate conditions (i.e., conditions on features that are not explicitly stored and have to be calculated on the fly using GROUP BY and HAVING clauses). The c1 query is considered low-complexity as it has only one condition and one join. c1 matches subclusters of at least 64 hosts each, which are ordered by subcluster size. The c0 and c3 queries have more conditions, two of which are aggregate. c0 (resp. c3) matches sites that host at least 64 (resp. 128) hosts (regardless of whether they belong to more than one cluster service), and have at least 10 TB of aggregate storage and at least 10 GB of aggregate RAM. The queries c0 and c3 are identical, except that c0 selects cluster services at sites that have in total at least 64 hosts, as opposed to 128 in c3. These numbers were chosen to adjust the selectivity of the queries to approximately 40% and 10% respectively.[4]

[3]Experimental evaluation for 5000 sites required the use of 100 Planetlab hosts. Larger grid sizes were not considered due to the difficulty of running experiments with more than 100 Planetlab hosts. (Planetlab hosts are typically shared by many users and tend to be unpredictable and unstable.)

[4]Due to the dynamic generation of resource data on every experiment launch, the selectivity varies slightly across experiments. For this reason, selectivity in practice is up to 5% higher

```
c1: low complexity query
    SELECT ClusterService.siteID, ClusterService.hasClusterServiceName,
            SubCluster.hasNumberOfHosts FROM ClusterService, SubCluster
    WHERE SubCluster.hasNumberOfHosts >= 64
            AND ClusterService.id = SubCluster.serviceID
    ORDER BY SubCluster.hasNumberOfHosts DESC LIMIT 100

c0: high complexity query with high selectivity (approximately 40%)
    SELECT Site.id, Site.totalHosts, ClusterService.hasClusterServiceName
            SUM(SubCluster.hasSizeGB*SubCluster.hasNumberOfHosts)
                                            AS totalStorageGB
            SubCluster.hasRAMsizeMB*SubCluster.hasNumberOfHosts
                                            AS totalRAMsizeMB
    FROM Site, ClusterService, SubCluster
    WHERE Site.totalHosts >= 64
            AND Site.id = ClusterService.siteID
            AND ClusterService.ID = SubCluster.serviceID
    GROUP BY ClusterService.siteID
    HAVING totalStorageGB >= 10000
            AND totalRAMsizeMB >= 10000
    ORDER BY Site.totalHosts DESC LIMIT 100

c3: high complexity query with low selectivity (approximately 10%)
    SELECT Site.id, Site.totalHosts, ClusterService.hasClusterServiceName
            SUM(SubCluster.hasSizeGB*SubCluster.hasNumberOfHosts)
                                            AS totalStorageGB
            SubCluster.hasRAMsizeMB*SubCluster.hasNumberOfHosts
                                            AS totalRAMsizeMB
    FROM Site, ClusterService, SubCluster
    WHERE Site.totalHosts >= 128
            AND Site.id = ClusterService.siteID
            AND ClusterService.ID = SubCluster.serviceID
    GROUP BY ClusterService.siteID
    HAVING totalStorageGB >= 10000
            AND totalRAMsizeMB >= 10000
    ORDER BY Site.totalHosts DESC LIMIT 100
```

Figure 6.1: Definition of the queries used in the evaluation.

setting	total sites	total cluster services	monitored resource changes	occurred resource changes	changes that should have occurred
baseline	2000	10000	348668	1050000	1050000
grid size					
s1	500	500	18812	52500	52500
s2	500	2500	93854	262500	262500
s3	500	5000	188101	525000	525000
s4	2000	2000	71928	210000	210000
s6	2000	20000	377103	2099946	2100000
s7	5000	5000	179130	525000	525000
s8	5000	25000	357036	2624688	2625000
s9	5000	50000	251232	5249314	5250000
frequency of resource changes					
r1	2000	10000	639980	2999866	3000000
r3	2000	10000	228217	300000	300000
frequency of updates					
u1	2000	10000	388575	1050000	1050000
u3	2000	10000	164223	1196250	1050000
u4	2000	10000	159237	1049409	1050000
u5	2000	10000	263956	1050000	1050000
number and distribution of query arrivals					
q1	2000	10000	348290	1050000	1050000
q3	2000	10000	345637	1049621	1050000
q4	2000	10000	344431	1027363	1050000
query complexity					
c1	2000	10000	348399	1050000	1050000
c3	2000	10000	346876	1049265	1050000

Table 6.2: Various metrics of the performed experiments.

6.2.2 Validation

The present evaluation is complicated by the number of considered settings, the complexity of running large-scale experiments over Planetlab, and the large amount of data in thousands of log-files that have to be cross checked. To verify that the prototype behaves as intended for a given configuration (e.g., that proxies generate resource changes at the specified frequency), the metrics in Table 6.2 were calculated. All the values in the table were derived from the proxy and

than the indicated values. (Higher selectivity implies more expensive queries, thus the measured QRT is pessimistic.)

data manager logs, except the last column, which indicates the number of resource changes that should have occurred during every experiment. This number is calculated as follows:

$$\text{nchanges} = S \times \sum_{e \in E} \frac{D}{e} \tag{6.1}$$

where

- S is the total number of cluster services (e.g., 10000 for the baseline setting);
- E is a set, every element of which denotes the average number of minutes between consecutive events of a given event type; e.g., in the r1 setting $E = 1, 1, 1, 1, 1$, whereas in the baseline setting $E = 1, 2.5, 5, 10, 20$;
- D is the experiment duration (60 minutes).

In every experiment, the number of occurred resource changes should always be less or equal to that shown in the last column of Table 6.2. When less, it means that the experiment terminates slightly before the nominal duration, and thus some scheduled events do not occur. This may occur because (i) the experiment terminates when the last query is served; (ii) queries are uniformly distributed over the experiment duration; and (iii) there is no guarantee that the last query will be scheduled at the end of the experiment's nominal duration. In the cases where this happens, the number of events that do not occur is negligible (e.g., the largest difference between the last two columns of the table is 0.02% in q4).[5]

6.2.3 Results

Network overhead

The plots in Figure 6.2 show how the network overhead that is imposed by monitoring is affected by frequency of resource changes, grid size, and frequency of updates (respectively, top, middle and bottom plots). Every plot has one bar per setting, and every bar is separated in two parts, which indicate: (i) the total

[5]There has been a small mis-configuration in the u3 experiment, but with little significance for the measured metrics. In one of the 50 experiment nodes, the 50 proxy instances at that node were configured to host 10 services each (instead of 5). This means that there were 250 more services than intended (i.e., 10250 instead of 10000). The effect of this mis-configuration is negligible for QRT and information freshness (if any), and insignificant for network overhead (in the order of 0.025%).

network overhead, and (ii) only that of differential updates (i.e., excluding the overhead for the initial full updates). More important is the network overhead of differential updates, as the full updates are performed only the first time a proxy is visited.

The middle plot in Figure 6.2 shows that the network overhead increases linearly with the number of monitored cluster services. This is not the case for the frequency of resource changes (top plot). Specifically, the relative difference of the differential updates overhead of the settings r1 and r3 compared to r0 is 0.58 and -0.36, respectively.[6] The relative difference of the corresponding number of monitored resource changes (Table 6.2) is substantially different in the case of r1 (0.84), but similar in r3 (-0.35).

The same holds for the frequency of updates (bottom plot in Figure 6.2). The u1 setting, where updates are performed 5 times as frequently compared to u0, has approximately twice the diff network overhead of u0. Similarly, u3, in which updates are performed 3 times less frequently compared to u0, has less than half the diff network overhead of u0. That is, an increase of a certain order in the frequency of updates does not guarantee an increase of exactly the same order in the number of monitored events (because there may not be as many events occurring in the first place). Likewise, an increase of a certain order in the frequency of events *per se* does not necessarily increase the network overhead at the same order, because the occurrence of more events does not guarantee that they will be monitored.

With respect to the benefits of using resource importance, u4 and u5 have, respectively, less than half, and about two thirds of the differential updates network overhead of the baseline setting.

Average information freshness

The plots in Figure 6.3 show how average information freshness is affected by frequency of resource changes, grid size, and update frequency. In the r1 setting (top plot in Figure 6.3), on average, only 13% of the changing resource information is fresh when resource changes occur 5 times more frequently than updates. In the baseline setting (r0), changes occur on average every 7.7 minutes (the average

[6]The relative difference is calculated as $\frac{(a-b)}{b}$ where b is the value of the baseline setting. The diff network overhead for all settings is listed in the "actual network overhead" column in Table 6.8, page 162.

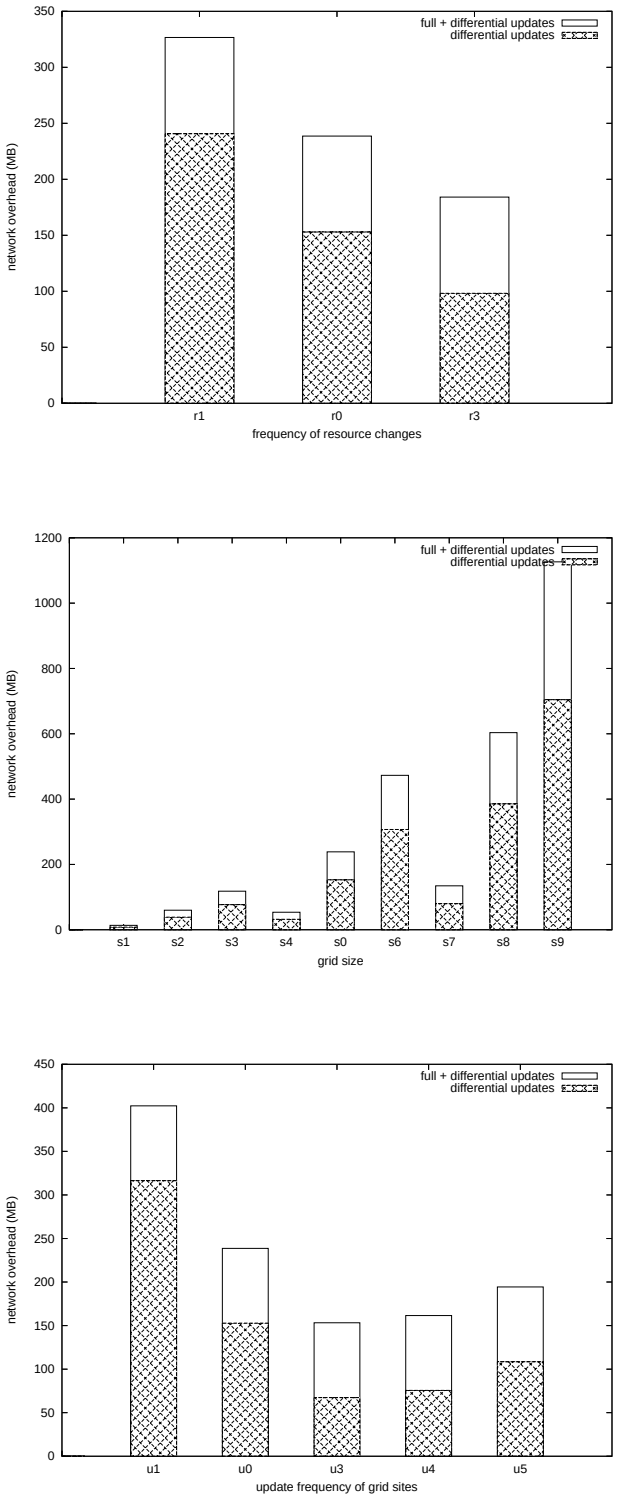

Figure 6.2: Network overhead for several variations of the baseline setting.

of 1, 2.5, 5, 10, and 20 minutes) in which case 56% of the changing resource information is up to date. In r3, where resource changes occur on average every second update, the average information freshness is 77%.

The middle plot of the figure indicates that the prototype cannot deal gracefully with a very large number of monitored service clusters. Specifically, in the settings s6, s8, and s9 (20000, 25000, and 50000 cluster services, respectively), information freshness drops from the expected 58% down to 44%, 39% and 25%, respectively. This is because of the high load at the main store, due to the updates performed by the data manager as well as the query load imposed by the query generator and the importance evaluator.

The bottom plot of Figure 6.3 shows the effect of update frequency to information freshness. In the u1 setting, where updates occur on average 7.7 times faster than changes, average freshness is 75%. Average freshness drops to 56% and 34% for u0 and u3, respectively. The average freshness for u4 and u5 is rather misleading as, in these settings, freshness varies significantly across sites based on site importance.

Instead, consider Figure 6.4, which shows the average freshness of the information about the resources of a site versus the importance of that site, in three different experiment settings: baseline, topn (u4), and inversely proportional (u5).

In the baseline experiment (top plot in Figure 6.4), all sites are updated every 15 minutes. Thus, average freshness is approximately the same for all sites. The middle plot clearly shows that sites are distinguished in two groups, and the smaller group has higher average information freshness. Observe the variability of information freshness among sites that have the same importance. This is due to delays from the point a proxy is queried, until the proxy's update is received and committed to the main store. Such delays depend on the responsiveness of a proxy, and the extent at which the data manager is busy with a backlog of previously downloaded updates. In the u5 experiment (bottom plot in Figure 6.4) the update frequency of a site is inversely proportional to its importance, and is mapped in the range [1, 15] minutes. Observe that even the sites with very low importance, have 35% to 40% freshness, as they are updated on average every 15 minutes.

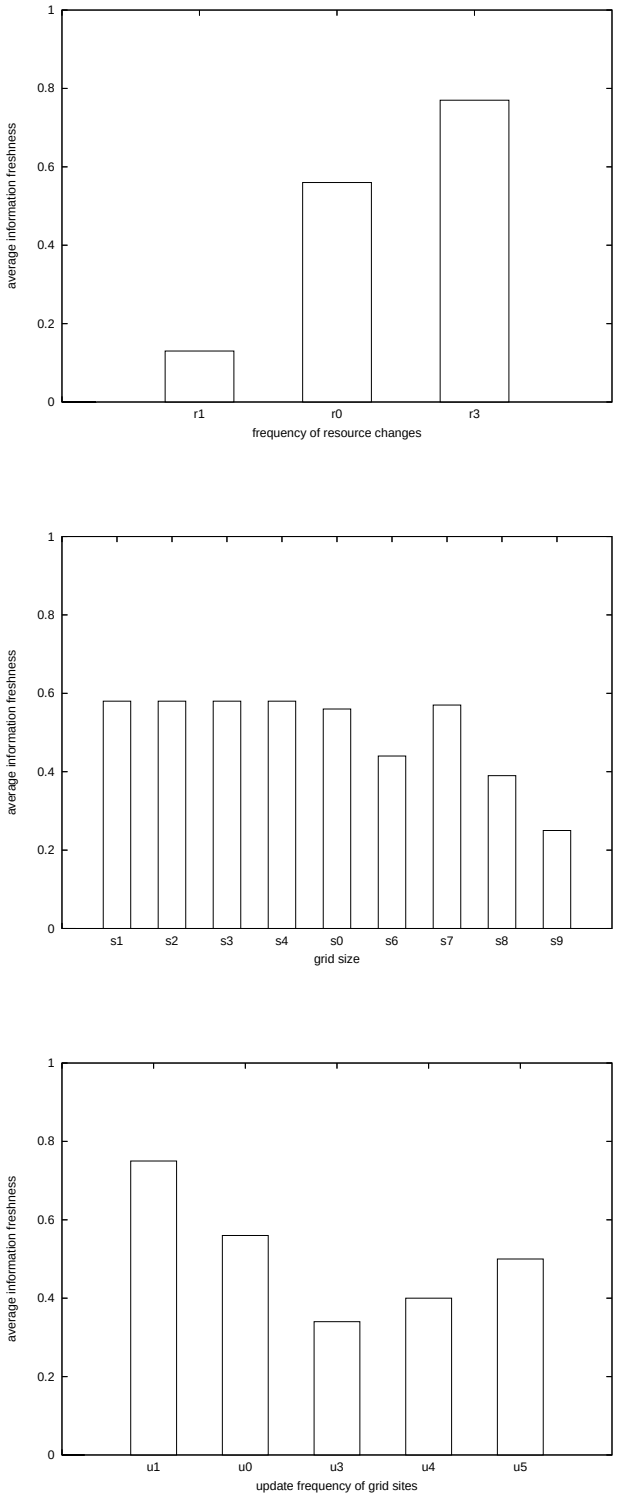

Figure 6.3: Average information freshness for several variations of the baseline setting.

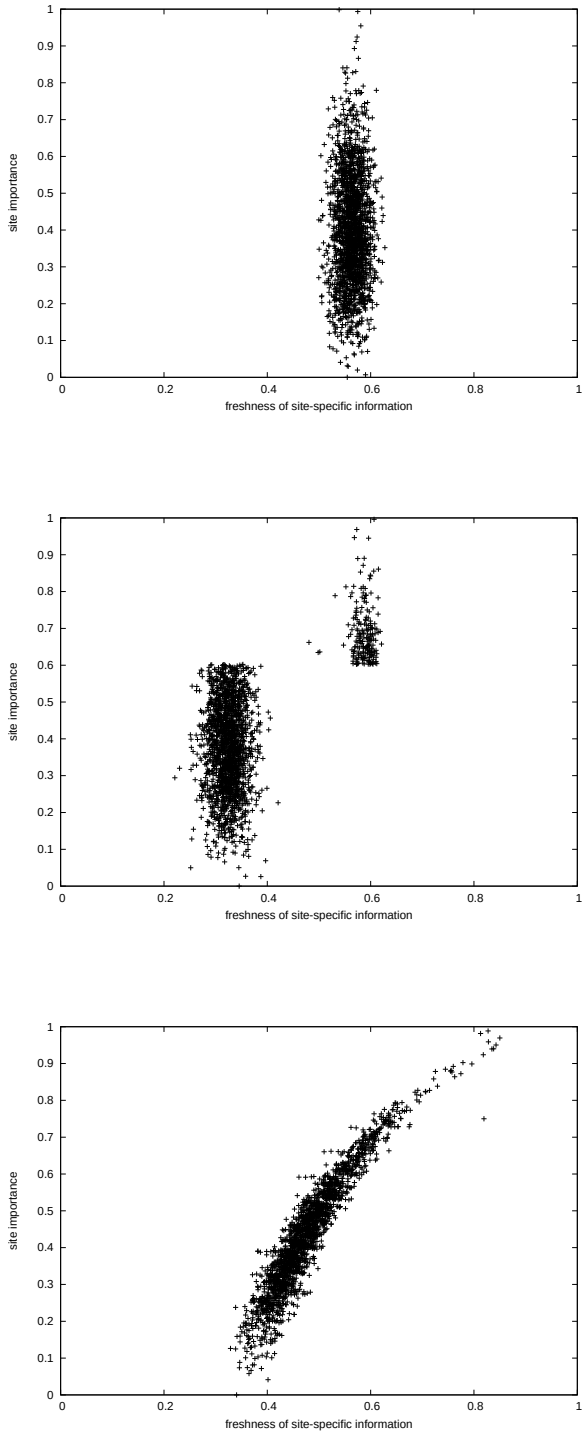

Figure 6.4: Average information freshness per site versus site importance, in the baseline (top), u4 (middle), and u5 (bottom) settings.

Average query response time

Figures 6.5 and 6.6 show how QRT is affected by variations in all of the considered problem settings. It can be seen in the top plot of Figure 6.5 that the difference of QRT between c1 and c0 is significant, as the latter has to calculate some aggregate values on the fly and has more conditions. On the other hand, QRT in c0 is very close to that of c3, despite the difference in query selectivity.

The middle plot shows how QRT is affected by the number and distribution of query arrivals. The relative difference between q1 and q0 on the one hand (2000 and 4000 queries respectively, uniform), and q3 and q4 on the other (2000 and 4000 queries respectively, 10 bursts), is negligible (less than 3%). The pattern of query arrivals however makes a significant difference (17% relative difference between q1 and q3; 15% between q0 and q4).

The bottom plot at Figure 6.5 indicates that the frequency of resource changes (which affects the imposed update load at the main store) does affect QRT but not significantly. The same holds for the effect of update frequency of grid sites (bottom plot of Figure 6.6). Finally, the top plot in Figure 6.6 shows that the relation of grid size (and as a result database size) and QRT tends to be proportional.

6.2.4 Discussion

The following general conclusions are drawn with regards to the performance behaviour of the prototype. Network overhead increases with the frequency of resource changes and that of updates (but the increases are not necessarily of the same order), and the number of monitored service clusters. Average information freshness, in addition to the frequency of updates and changes, is also affected by implementation issues (in the case of the prototype, the capability of the data manager to scale for an increasing number of incoming updates). Finally average query response time is mostly affected by query complexity, grid size, and the timing of query arrivals (and less so by the total number of queries).

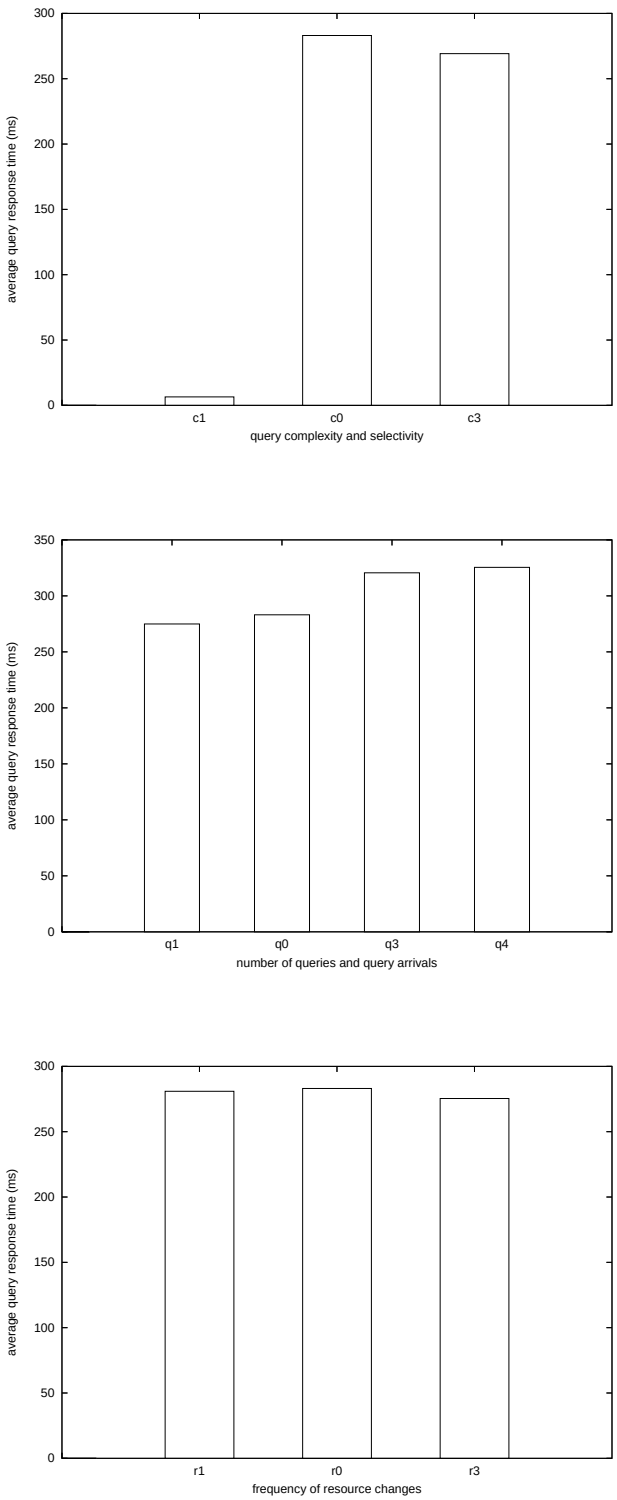

Figure 6.5: Average QRT for variations of the baseline setting in terms of query complexity and selectivity (top), number of queries and query arrivals (middle), and frequency of resource changes (bottom).

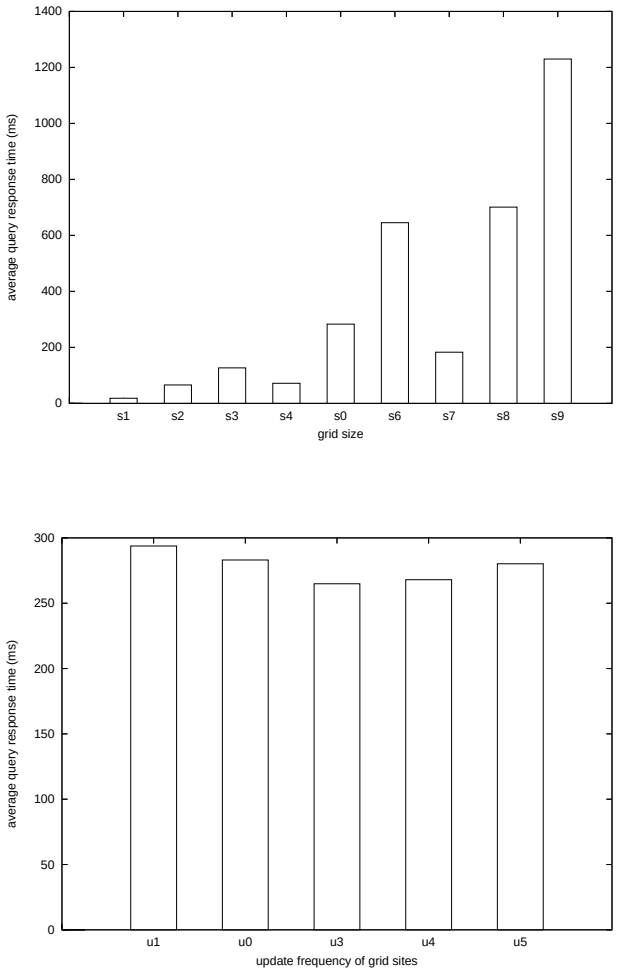

Figure 6.6: Average QRT for variations of the baseline setting in terms of grid size (top) and update frequency of grid sites (bottom).

6.3 Comparison Against Just-In-Time Hierarchies of Aggregation

This section compares selected performance results of just-in-time hierarchies of aggregation (JITHAs), from Chapter 3, against the prototype results of the previous section, in identical problem settings. Among all the JITHAs that were considered in Chapter 3, the present comparison includes those that delivered the lowest QRT in at least one setting.

6.3.1 Average Query Response Time

Tables 6.3-6.5 compare the two approaches for 40% query selectivity and 500, 2000 and 5000 grid sites, with 5 cluster services each. The data is grouped in different tables based on the number of levels of a JITHA: Table 6.3 lists two-level JITHAs; Table 6.4 lists three-level JITHAs; and Table 6.5 lists both four- and five-level JITHAs. In addition, Table 6.6 compares JITHAs of all levels against the prototype, for 10% query selectivity and only for 2000 sites (because that is the only grid size for which the prototype was evaluated with query selectivity of 10%). In the case of JITHAs, results in all aforementioned tables are for 20% cache hit ratio.

The structure of Tables 6.3-6.6 is as follows. The results are organised in groups; every group's first line is about the prototype (indicated with double horizontal lines). The remaining of the lines in a group, list QRT data for selected JITHAs in a given setting. Different JITHAs in the same group are separated by a single horizontal line. Every group compares one or more JITHAs to the equivalent prototype experiment (equivalent in terms of problem settings).

- The first column shows whether a line refers to the prototype ("prototype") or one of the considered JITHAs (where nsites-$A \times B \times C$ indicates the considered grid size setting and JITHA).

- The second column shows the considered setting.[7] The settings of every group are identical in terms of grid size, number of queries, and query selectivity. The remaining parameters of a problem setting are specific to one of the two approaches (e.g., update frequency in the case of the

[7]The notation for settings of JITHAs is explained in Section 3.5.5, page 70.

prototype; query processing time in the case of JITHAs). As an example, the first group in Table 6.3 is about monitoring 500 sites with 5 cluster services each, and responding to 4000 queries with query selectivity of 40%.

- The third column shows the average QRT of the considered system in the given setting.
- The fourth column shows the relative difference between the average QRT of the JITHA at the corresponding line, and the prototype entry in that group.[8] A positive relative difference indicates that the prototype has lower QRT.
- The last column shows the standard deviation of QRT for the system at that line. Variability in query performance is mainly due to varying load at the main store of the monitoring site in the case of the prototype, and cache misses in the case of JITHAs.

The QRT of the two approaches is compared in terms of average and standard deviation. The prototype's average QRT is smaller in most cases. A few JITHAs have slightly smaller average QRT, when evaluated with the smallest resource match size (0.1 KB). These JITHAs are:

- In Table 6.4 (three-level JITHAs):
  - $1 \times 16 \times 125$ with -0.14 relative difference, when monitoring 2000 sites;
  - $1 \times 128 \times 39$ with up to -0.46 relative difference, when monitoring 5000 sites;
  - $1 \times 32 \times 156$ with up to -0.55 relative difference, when monitoring 5000 sites.
- In Table 6.5 (four-level JITHAs):
  - $1 \times 4 \times 16 \times 31$ with -0.16 relative difference, when monitoring 2000 sites;
  - $1 \times 4 \times 4 \times 125$ with -0.15 relative difference, when monitoring 2000 sites;

[8] The relative difference is calculated as $\frac{qrt_{JITH} - qrt_{prototype}}{qrt_{prototype}}$.

	setting	average QRT (ms)	relative difference	std dev of QRT (ms)
prototype	s2	65.80		25.02
500-1 × 500 × 5	0.1-0.40-25	241.51	2.67	109.62
	1.0-0.40-25	372.06	4.65	175.72
	2.0-0.40-25	556.73	7.46	269.22
	0.1-0.40-100	376.21	4.72	139.85
	1.0-0.40-100	506.76	6.70	205.94
	2.0-0.40-100	691.43	9.51	299.44
prototype	baseline	238.10		44.14
2000-1 × 2000 × 5	0.1-0.40-25	579.81	1.44	280.90
	1.0-0.40-25	977.81	3.11	482.41
	2.0-0.40-25	1177.61	3.95	583.57
	0.1-0.40-100	714.51	2.00	311.13
	1.0-0.40-100	1112.51	3.67	512.64
	2.0-0.40-100	1312.31	4.51	613.80
prototype	s8	700.90		172.03
5000-1 × 5000 × 5	0.1-0.40-25	1034.33	0.48	511.03
	1.0-0.40-25	2210.82	2.15	1106.69
	2.0-0.40-25	5956.00	7.50	3002.90
	0.1-0.40-100	1169.03	0.67	541.26
	1.0-0.40-100	2345.52	2.35	1136.92
	2.0-0.40-100	6090.70	7.69	3033.12

Table 6.3: Prototype against two-level JITHAs with 40% selectivity and 20% cache hit ratio.

  - $1 \times 4 \times 32 \times 39$ with up to -0.64 relative difference, when monitoring 5000 sites.

- In Table 6.5 (five-level JITHAs):

  - $1 \times 2 \times 4 \times 16 \times 39$ with up to -0.60 relative difference, when monitoring 5000 sites.

In all the above cases, the relative difference is rather small (up to -0.64). The average QRT is many times higher than that of the prototype, in the case of two-level JITHAs (Table 6.3), and in most cases with settings of 1 or 2 KB resource match size (Tables 6.3-6.6). Furthermore, the average QRT of JITHAs of all levels, for 10% query selectivity, is up to two orders of magnitude larger than that of the prototype (Table 6.6). Finally, the standard deviation of QRT of JITHAs tends to be several times larger than that of the prototype, especially in the settings with 1 and 2 KB resource match size.

Overall, the results indicate that JITHAs deliver a higher QRT, compared to that of the prototype, for most of the considered settings. Also, recall that the JITHA's performance estimates are optimistic, due to the assumptions that were made in Sections 3.4.1 and 3.5.1.

	setting	average QRT (ms)	relative difference	std dev of QRT (ms)
prototype	s2	65.80		25.02
500-1 × 16 × 31 × 5	0.1-0.40-25	161.50	1.45	77.43
	1.0-0.40-25	338.21	4.14	162.38
	2.0-0.40-25	364.80	4.54	175.77
	0.1-0.40-100	344.01	4.23	138.07
	1.0-0.40-100	520.73	6.91	221.76
	2.0-0.40-100	547.32	7.32	235.00
500-1 × 4 × 125 × 5	0.1-0.40-25	163.12	1.48	85.49
	1.0-0.40-25	225.69	2.43	113.99
	2.0-0.40-25	447.78	5.81	253.48
	0.1-0.40-100	345.63	4.25	145.70
	1.0-0.40-100	408.21	5.20	174.63
	2.0-0.40-100	630.30	8.58	313.83
prototype	baseline	238.10		44.14
2000-1 × 16 × 125 × 5	0.1-0.40-25	203.72	−0.14	102.98
	1.0-0.40-25	487.58	1.05	239.84
	2.0-0.40-25	894.34	2.76	460.61
	0.1-0.40-100	386.23	0.62	163.59
	1.0-0.40-100	670.09	1.81	299.10
	2.0-0.40-100	1076.85	3.52	520.59
2000-1 × 4 × 500 × 5	0.1-0.40-25	264.47	0.11	155.22
	1.0-0.40-25	618.96	1.60	338.49
	2.0-0.40-25	1043.87	3.38	568.13
	0.1-0.40-100	446.98	0.88	214.83
	1.0-0.40-100	801.48	2.37	399.12
	2.0-0.40-100	1226.38	4.15	628.75
prototype	s8	700.90		172.03
5000-1 × 128 × 39 × 5	0.1-0.40-25	378.97	−0.46	182.87
	1.0-0.40-25	956.87	0.37	473.21
	2.0-0.40-25	1679.80	1.40	838.69
	0.1-0.40-100	561.49	−0.20	242.02
	1.0-0.40-100	1139.38	0.63	530.01
	2.0-0.40-100	1862.31	1.66	894.60
5000-1 × 32 × 156 × 5	0.1-0.40-25	315.97	−0.55	157.08
	1.0-0.40-25	887.02	0.27	440.20
	2.0-0.40-25	1728.07	1.47	872.78
	0.1-0.40-100	498.48	−0.29	217.50
	1.0-0.40-100	1069.53	0.53	498.03
	2.0-0.40-100	1910.58	1.73	930.87

Table 6.4: Prototype against three-level JITHAs with 40% selectivity and 20% cache hit ratio.

6.3.2 Network Overhead

The plots in Figures 6.7 and 6.8 compare the network overhead (y axis; log scale) of JITHAs and the prototype when processing 4000 queries (x axis). (Again, 20% cache hit ratio is assumed in the case of JITHAs). The figures show the network overhead for grids of 500, 2000 and 5000 sites (500 and 2000 sites, respectively, in the top and bottom rows of Figure 6.7; 5000 sites in Figure 6.8) with 5 cluster services per site. In the case of JITHAs, the left and right columns of Figures 6.7-6.8 show the network overhead for 10% and 40% query selectivity, respectively. (The values for the prototype in plots of the same row are identical, because in the

	setting	average QRT (ms)	relative difference	std dev of QRT (ms)
prototype	s2	65.80		25.02
500-1 × 4 × 4 × 31 × 5	0.1-0.40-25	162.90	1.48	91.02
	1.0-0.40-25	228.18	2.47	119.83
	2.0-0.40-25	314.59	3.78	162.27
	0.1-0.40-100	383.91	4.83	182.23
	1.0-0.40-100	449.19	5.83	210.98
	2.0-0.40-100	535.60	7.14	252.66
prototype	baseline	238.10		44.14
2000-1 × 4 × 16 × 31 × 5	0.1-0.40-25	200.65	−0.16	112.35
	1.0-0.40-25	589.74	1.48	320.03
	2.0-0.40-25	886.45	2.72	463.08
	0.1-0.40-100	421.66	0.77	203.73
	1.0-0.40-100	810.74	2.41	409.95
	2.0-0.40-100	1107.46	3.65	550.90
2000-1 × 4 × 4 × 125 × 5	0.1-0.40-25	202.06	−0.15	117.43
	1.0-0.40-25	502.13	1.11	261.86
	2.0-0.40-25	957.64	3.02	523.01
	0.1-0.40-100	423.06	0.78	208.46
	1.0-0.40-100	723.14	2.04	351.44
	2.0-0.40-100	1178.65	3.95	612.98
2000-1 × 2 × 4 × 8 × 31 × 5	0.1-0.40-25	298.97	0.26	210.77
	1.0-0.40-25	663.57	1.79	388.75
	2.0-0.40-25	1067.38	3.48	600.16
	0.1-0.40-100	550.29	1.31	330.11
	1.0-0.40-100	914.89	2.84	507.82
	2.0-0.40-100	1318.71	4.54	717.03
prototype	s8	700.90		172.03
5000-1 × 4 × 32 × 39 × 5	0.1-0.40-25	254.46	−0.64	138.30
	1.0-0.40-25	1061.17	0.51	552.64
	2.0-0.40-25	1842.43	1.63	945.55
	0.1-0.40-100	475.46	−0.32	229.62
	1.0-0.40-100	1282.17	0.83	639.79
	2.0-0.40-100	2063.44	1.94	1029.95
5000-1 × 2 × 4 × 16 × 39 × 5	0.1-0.40-25	279.79	−0.60	165.63
	1.0-0.40-25	1282.41	0.83	715.42
	2.0-0.40-25	2281.45	2.26	1238.42
	0.1-0.40-100	531.12	−0.24	285.92
	1.0-0.40-100	1533.73	1.19	831.46
	2.0-0.40-100	2532.78	2.61	1350.89

Table 6.5: Prototype against four- and five-level JITHAs with 40% selectivity and 20% cache hit ratio.

prototype's case query selectivity is not relevant to network overhead.) The plots show the network overhead as a function of the number of processed queries. The lines for the prototype are flat, as the imposed network overhead is independent from the number of queries.

In the prototype, the total overhead (i.e., full plus differential updates) is, respectively, 59.9, 238.6, and 603.5 MB for the settings s2 (500 sites, 2500 cluster services), baseline (2000 sites, 10000 cluster services), and s8 (5000 sites, 25000 cluster services). The overhead for the differential updates is 38.5, 152.9 and

	setting	average QRT (ms)	relative difference	std dev of QRT (ms)
prototype	c1	6.50		13.06
2000-1 × 2000 × 5	0.1-0.10-25	579.02	88.08	280.50
	1.0-0.10-25	602.90	91.75	292.59
	2.0-0.10-25	644.29	98.12	313.55
	0.1-0.10-100	713.72	108.80	310.73
	1.0-0.10-100	737.60	112.48	322.82
	2.0-0.10-100	778.99	118.84	343.78
2000-1 × 16 × 125 × 5	0.1-0.10-25	187.64	27.87	96.09
	1.0-0.10-25	369.77	55.89	181.55
	2.0-0.10-25	391.90	59.29	192.44
	0.1-0.10-100	370.15	55.95	156.60
	1.0-0.10-100	552.28	83.97	241.47
	2.0-0.10-100	574.41	87.37	252.23
2000-1 × 4 × 500 × 5	0.1-0.10-25	244.09	36.55	145.67
	1.0-0.10-25	309.84	46.67	174.76
	2.0-0.10-25	391.68	59.26	213.00
	0.1-0.10-100	426.60	64.63	204.99
	1.0-0.10-100	492.36	74.75	234.94
	2.0-0.10-100	574.20	87.34	273.56
2000-1 × 4 × 16 × 31 × 5	0.1-0.10-25	183.81	27.28	105.78
	1.0-0.10-25	247.94	37.14	133.46
	2.0-0.10-25	332.81	50.20	174.43
	0.1-0.10-100	404.82	61.28	197.03
	1.0-0.10-100	468.95	71.15	224.75
	2.0-0.10-100	553.82	84.20	265.02
2000-1 × 4 × 4 × 125 × 5	0.1-0.10-25	185.38	27.52	111.01
	1.0-0.10-25	250.37	37.52	138.81
	2.0-0.10-25	335.91	50.68	179.26
	0.1-0.10-100	406.38	61.52	201.78
	1.0-0.10-100	471.38	71.52	230.02
	2.0-0.10-100	556.92	84.68	270.08
2000-1 × 2 × 4 × 8 × 31 × 5	0.1-0.10-25	275.82	41.43	201.23
	1.0-0.10-25	356.93	53.91	235.97
	2.0-0.10-25	460.73	69.88	285.48
	0.1-0.10-100	527.15	80.10	320.19
	1.0-0.10-100	608.26	92.58	355.92
	2.0-0.10-100	712.05	108.55	405.51

Table 6.6: Prototype (c1 setting) against two-, three-, four- and five-level JITHAs with 10% selectivity and 20% cache hit ratio.

385.4 MB for s2, s0, and s8, respectively. In the case of JITHAs, network overhead ranges from 77.1 to 15,522 MB for 10% query selectivity, and from 308.5 to 62,088 MB for 40% selectivity.

It can be seen that the prototype's total network overhead is smaller than that of JITHAs in all of the considered settings. However, the network overheads of the prototype are fairly similar to those of JITHAs that have 0.1 KB resource match size and 10% query selectivity (left column plots).

The plots also indicate the minimum number of queries for which the prototype is more appropriate compared to JITHAs, in the considered settings. In the left-column plots, it is about 3000 queries for 0.1 KB resource match size, and and less than 500 queries for 1 and 2 KB resource match size. In other words,

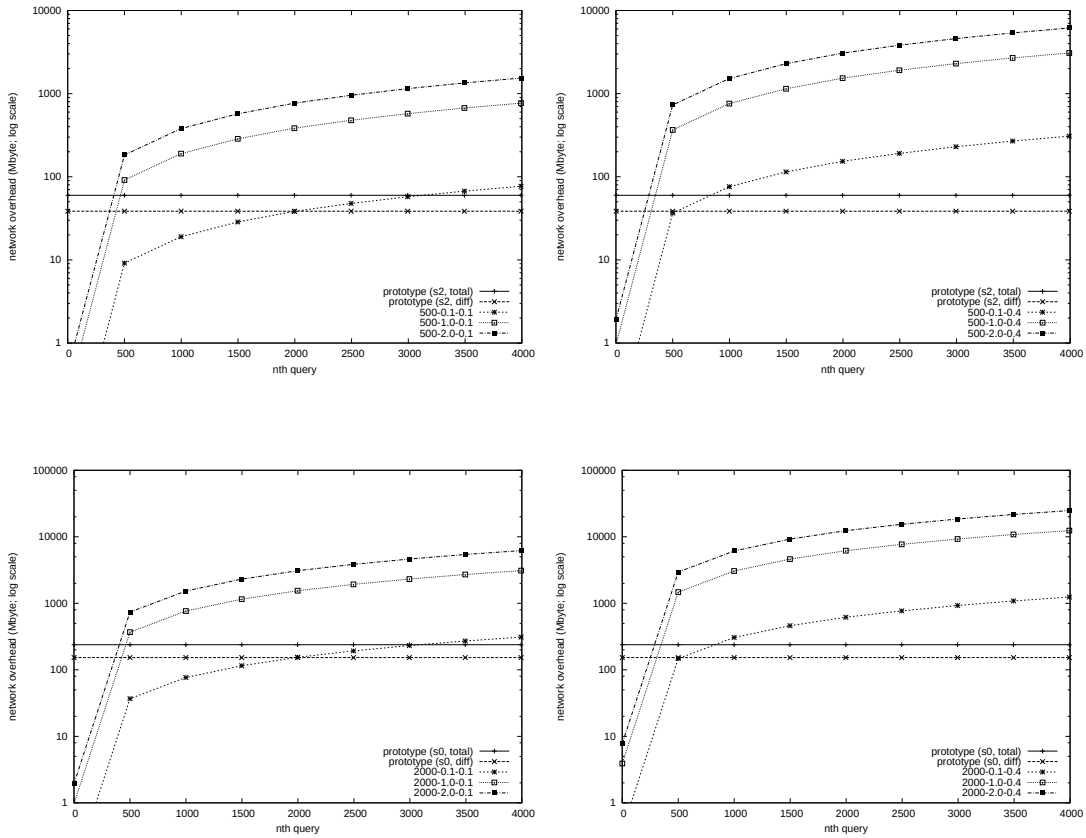

Figure 6.7: Comparison of the prototype against JITHAs in terms of network overhead, when receiving 4000 queries, and monitoring 500 and 2000 sites (top and bottom rows, respectively) for 10% and 40% query selectivity (left and right columns, respectively), and for 20% cache hit ratio in the case of JITHAs.

with 10% query selectivity, the prototype is more appropriate against a JITHA that has 0.1 KB match size, when at least 3000 queries are submitted within the considered interval (in this case per hour).

In the case of 40% selectivity (right column plots), the network overhead per query in JITHAs is higher. Thus, the minimum number of queries that justifies the prototype's total network overhead (compared to that imposed by JITHAs) is smaller: approximately 800 queries for 0.1 KB resource match size, and 200 queries for 1 and 2 KB resource match size.

Overall, in terms of network overhead, JITHAs should be preferred when the average number of expected queries over a certain time interval is rather small and with small selectivity.

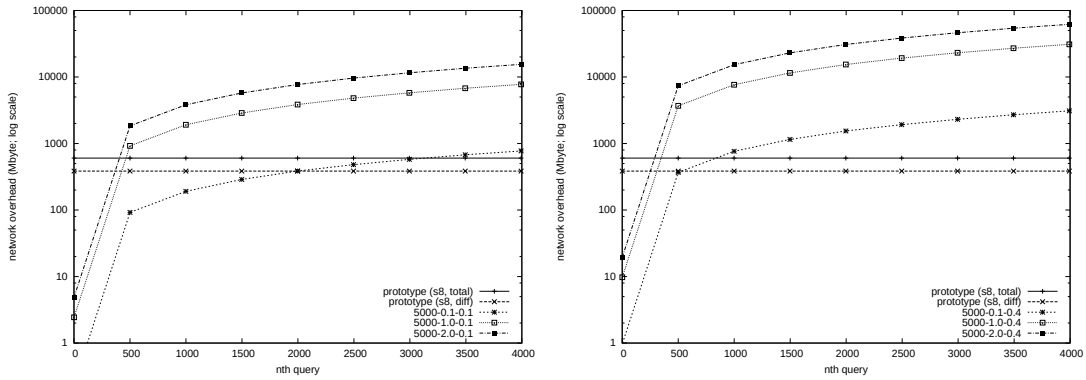

Figure 6.8: Comparison of the prototype against JITHAs in terms of network overhead, when receiving 4000 queries, and monitoring 5000 sites for 10% and 40% query selectivity (left and right columns, respectively), and for 20% cache hit ratio in the case of JITHAs.

6.3.3 Discussion

The prototype outperforms JITHAs in terms of average QRT in most of the considered settings (in some of which it clearly outperforms by orders of magnitude). The prototype also outperforms JITHAs in terms of network overhead (in all of the considered settings, but assuming a minimum number of queries), at the cost of varying and lower average information freshness. JITHAs were evaluated for 10% and 20% cache hit ratio in Chapter 3; only the latter (more optimistic) setting was considered in the present comparison. With sufficiently high values of cache hit ratio, JITHAs may outperform the proposed architecture, in terms of average QRT (although, in any case, JITHAs, which cross administrative boundaries, cannot provide any guarantees for QRT). The proposed architecture may also be problematic when the average number of queries per given time interval is not sufficiently large to justify the imposed network overhead (even though it can be reduced by compromising information freshness). In any case, the proposed architecture is intended for large grids, with many users (which implies a large number of resource discovery queries). In general, the prototype performs well for the envisaged uses, except for the data manager being a bottleneck in very large grid settings. In a production-quality implementation, this could be addressed with data partitioning across several (potentially replicated) databases.

6.4 Model-Based Estimations Compared to Experimental Results

This section compares performance estimations, based on the information freshness and network overhead models in Section 4.7, against the results obtained from the Planetlab experiments. The aim is to examine the accuracy of the models' estimations.

6.4.1 Average Information Freshness

Table 6.7 compares average information freshness values, as measured in Planetlab experiments, against estimations based on the average information freshness model in Section 4.7 (page 95). Average information freshness is not a useful metric when site-level freshness varies significantly, thus the comparison in Table 6.7 does not include settings u4, and u5. (u5 is discussed separately at the end of the present section.) Table 6.7 is organised as follows.

N	setting	D	Mm	Mc list	freshness		
			(approx.) (mins)	(mins)	estimation	actual	diff.
1	baseline	60	4.9	1,2.5,5,10,20	0.61	0.56	−0.05
2	c1	59	4.9	1,2.5,5,10,20	0.61	0.60	−0.01
3	c3	59	4.9	1,2.5,5,10,20	0.61	0.60	−0.01
4	q1	59	4.9	1,2.5,5,10,20	0.61	0.60	−0.01
5	q3	55	4.9	1,2.5,5,10,20	0.61	0.60	−0.01
6	q4	55	4.9	1,2.5,5,10,20	0.61	0.60	−0.01
7	s1	59	4.9	1,2.5,5,10,20	0.61	0.58	−0.03
8	s2	59	4.9	1,2.5,5,10,20	0.61	0.58	−0.03
9	s3	59	4.9	1,2.5,5,10,20	0.61	0.58	−0.03
10	s4	59	4.9	1,2.5,5,10,20	0.61	0.58	−0.03
11	s6	59	4.9	1,2.5,5,10,20	0.61	0.44	−0.17
12	s7	59	4.9	1,2.5,5,10,20	0.61	0.57	−0.04
13	s8	59	4.9	1,2.5,5,10,20	0.61	0.39	−0.22
14	s9	65	4.9	1,2.5,5,10,20	0.61	0.25	−0.36
15	r1	59	4.9	1,1,1,1,1	0.17	0.13	−0.04
16	r3	59	4.9	10,10,10,10,10	0.67	0.77	0.10
17	u1	60	0.9	1,2.5,5,10,20	0.89	0.75	−0.14
18	u3	54	14.9	1,2.5,5,10,20	0.34	0.34	−0.00

Table 6.7: Average information freshness: model estimations versus experiment results.

- The first and second columns indicate the identifiers with which an experiment may be referred in this discussion.
- The third column (D) indicates the duration of every experiment, in minutes. The actual experiment duration may vary slightly from the nominal

duration of 60 minutes, because experiments are terminated as soon as the last query is served (see the second paragraph of Section 6.2.2, page 137).

- Mm indicates the average number of minutes between consecutive visits to proxies. The listed values are adjusted for a 5 second window (Section 4.3.2). Thus, for Mm=5 minutes, the effective Mm is:

$$(Mm \times 60 - 5)/60 \simeq 4.91666.$$

- Mc list indicates the list of frequencies of resource changes for the 5 different property types that do change during experiments.

- The freshness columns indicate the average information freshness during an experiment, as follows:

  Estimation The expected freshness of a collection of k properties is calculated using Equation 4.5, in which $x = \frac{Rc_{avg}}{Rm}$. Rc_{avg} is the average of the weighed frequencies of change of all resource property types P, in which the weighting factor is the number of properties of type p (nprops_p) in a collection of resource properties.

$$Rc_{avg} = \frac{1}{k} \sum_{p \in P} Rc_p \times \text{nprops}_\text{p} \tag{6.2}$$

  where $k = \sum_{p \in P} \text{nprops}_\text{p}$. In the case of the Planetlab experiments, every property type that changes has one instance per cluster service (i.e., every cluster service has 5 changing properties, each one of a different property type). Thus, nprops_p = nservs for all 5 property types (i.e., Rc_{avg} is the average of all 5 Rc_p, weighed equally).

  For instance, the average information freshness for the baseline setting, taking into account Equation 6.2, is estimated as follows:

$$x = \frac{Rc_{avg}}{Rm} = \frac{Mm}{Mc_{avg}} = \frac{4.9}{\frac{1+2.5+5+10+20}{5}} \simeq 0.64$$
$$f(x) = \frac{1}{1+x} \simeq 0.61.$$

  Measurement is measured from the experiment logs using Equation 4.4 and according to Section 6.1.3.

Difference is the absolute difference between estimation and actual freshness.

Settings 1-14 in Table 6.7 are, in principle, identical in terms of information freshness, as they all have the same frequencies of monitoring updates and resource changes (i.e., Mm and Mc list).[9] In practice, the actual freshness varies between 0.56 and 0.60 in settings 1-10 and 12, and drops significantly in large grid settings (because the Data Manager gets overwhelmed with incoming differential updates). With the exception of the large grid settings, the estimated freshness is quite accurate.

In the last group of settings (15-18 in Table 6.7), the freshness estimates for r1 and u3 are fairly accurate. The estimate for r3 is quite pessimistic compared to the actual measurement. It is believed that this discrepancy is a side-effect of the combination of the following conditions. First, in r3, resource changes occur only every 10 minutes. Second, all proxies are contacted for updates at the beginning of every experiment, which implies that freshness is close to ideal for the first Mc minutes. Third, experiment duration (D) is not sufficiently larger than Mc, to smooth the aforementioned effect of ideal freshness during the first Mc minutes.

Finally, the u1 estimate is 0.14 higher than the actual, because the model does not account for the possibility of the data manager being overwhelmed with incoming updates (which is the case in u1 because of the low Mm, i.e., high update frequency).

The setting u5 is particularly useful to the present discussion because it determines Mm individually for every site (using the inversely proportional mapping method in Section 4.3.2). As a result, the u5 experiment provides data for 2000 different values of Mm, in the range $[1, 15]$, and the corresponding (site-level) average freshness values. Figure 6.9 compares estimations of average information freshness (using Equation 4.5, page 98), against actual measurements of average information freshness, at the site-level. The estimated trend by the model fits well with the experimental results of average information freshness given a certain update frequency, although most of the experimental freshness measurements are smaller than the corresponding estimation (especially for small values of Mm).[10]

[9] The numbers 1-14 refer to the first column of Table 6.7.

[10] A possible explanation for the discrepancy with smaller values of Mm is the data manager being slow with processing downloaded updates. Assume that an average of 1 minute is required to commit to the main store every downloaded update, because, on average, the crawler downloads updates faster than the data manager commits them to the main store. The

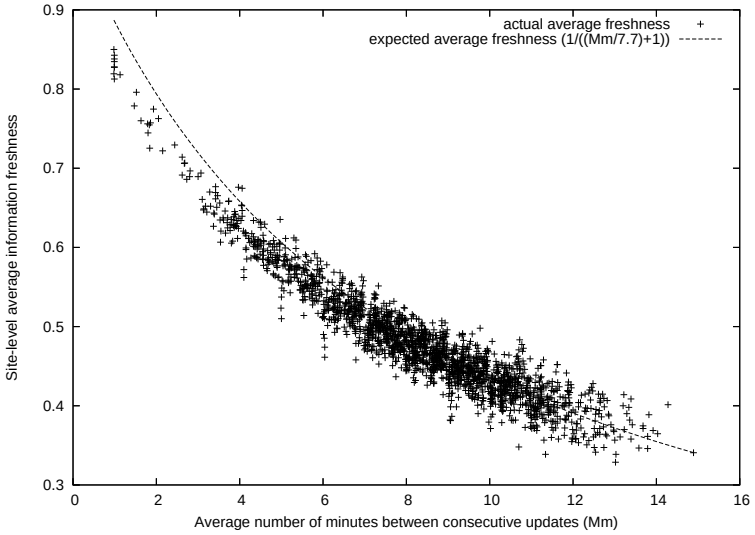

Figure 6.9: Average time interval between consecutive updates Mm (x-axis) vs site-level average information freshness (y-axis) in u5; crosses indicate experimental values; line indicates model estimate.

In general, the absolute difference between the 2000 site-level average information freshness measurements in u5 and the corresponding model estimations is, on average, 2% with a standard deviation of 1.5%.

6.4.2 Network Overhead of Differential Updates

Table 6.8 compares the estimation of network overhead due to differential updates, and other relevant metrics, against experimental results.

- The nupdates columns indicate the number of updates (including empty ones) that the crawler downloads during an experiment.

  Estimation According to Equation 4.8 the total number of received updates is $D \times Rm \times$ nsites. The estimations in Table 6.8 increase the outcome of Equation 4.8 by nsites, because sites are not only updated at the end of consecutive time intervals of Mm minutes, but also at

impact of 1 extra minute is more significant to the freshness of more frequently updated sites. For instance, let Mc=7.7, and two different Mm values, 2 and 12 minutes. The difference in freshness between Mm=2 and Mm=3 is 0.074, whereas between Mm=12 and Mm=13 is 0.019.

the beginning of every experiment.[11]

Measurement Calculated based on experiment logs.

- The nmchanges columns indicate the number of resource changes that are monitored during an experiment.

  Estimation Calculated according to Equation 4.11.

  Measurement Calculated based on experiment logs.

- Network overhead indicates the network overhead of all the differential updates that are downloaded during an experiment.

  Estimation Calculated according to Equation 4.13, in which the average size of a description of a resource change is 460 bytes, increased by a fixed overhead of 80 bytes per update.

  Measurement Calculated based on experiment logs, according to Equation 4.2.

In all three columns, the relative difference is calculated as $\frac{estimated-actual}{actual}$.

The estimations in the table are fairly accurate in most settings (with a relative difference of up to 0.06). In settings u1, s6, s8, and s9, the number of monitored changes (nmchanges) is significantly overestimated, because in these settings the data manager is overwhelmed with incoming updates, many of which are not committed by the end of the experiment. (The metric nmchanges refers to the updates that have been downloaded *and* committed to the database). In the setting u3, both nupdates and nmchanges are overestimated, and as a result the same holds for the estimated diff network overhead.

Figure 6.10 shows site-level diff network overhead (x axis) versus site-level average information freshness (y axis), in the u5 experiment. The figure compares measured values of site-level diff network overhead (shown as a continuous line) against model-based estimations (shown as a dashed line). It can be seen that the model estimations are fairly accurate: the relative difference between the 2000 site-level measurements and the corresponding model estimations is, on average, 7.2% with a standard deviation of 4.2%. The model appears to slightly underestimate the diff network overhead in cases where average information is

[11]For examples of using the equations cited in this section, see Appendix C.

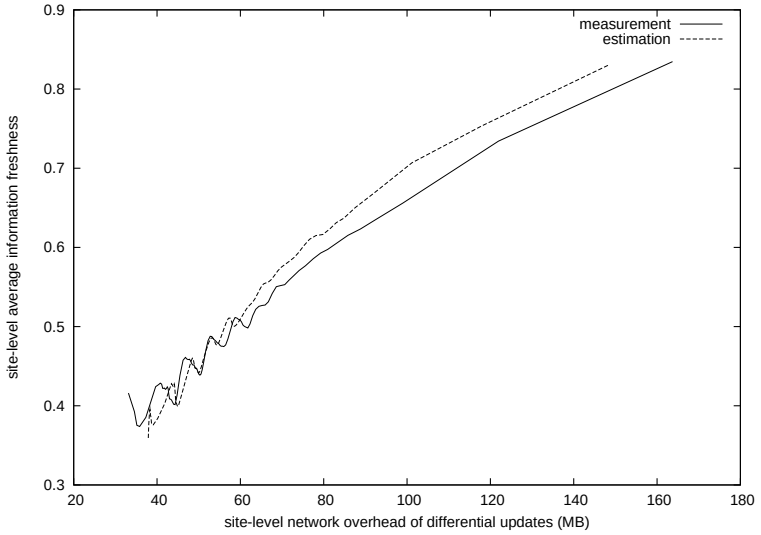

Figure 6.10: Comparison of estimated site-level diff network overhead against measured site-level network overhead (x axis) versus measured site-level average information freshness (y axis), in the u5 setting.

high. A likely explanation for this is that the estimated average information freshness (based on which diff network overhead is estimated) is slightly higher than the measured.

6.4.3 Discussion

Looking at Figures 6.9 and 6.10, it is clear that the models for average information freshness and network overhead of differential updates are fairly accurate across a wide range of monitoring frequencies (as is the case in the u5 experiment). The average information freshness model tends to be slightly more optimistic compared to the experimental measurements, as it does not account for the delay that is introduced by the data manager. The diff network overhead model is based on the average information freshness model, thus any errors in the latter propagate to the former.

6.5 Closing Remarks

The present chapter evaluated a prototype implementation of the proposed architecture, in Planetlab. Several problem settings were considered, including grids of up to 5000 sites and 50000 cluster services. The results demonstrate the performance trade-offs between network overhead, frequency of resource changes, and frequency of updates. The evaluation shows that the prototype delivers low QRT and maintains the expected information freshness, except for very large settings, for which further tuning is needed to deal with the large number of updates.

The prototype was compared against selected just-in-time hierarchies of aggregation (those which were found to deliver the lowest QRT in at least one setting in Chapter 3). Overall, the prototype delivers lower QRT in most cases, except for the most optimistic JITHA-specific settings (i.e., low resource match size). The prototype is especially appropriate for high selectivity queries, as its QRT performance is far less sensitive than that of JITHAs, in terms of query selectivity. The comparison assumed a 20% cache hit ratio for JITHAs. JITHAs may outperform the prototype when higher cache hit ratio applies.

The prototype imposes smaller network overhead compared to JITHAs, in all the examined settings. This, however, may not be the case with higher frequencies of resource changes than those considered, with smaller query selectivity, and with higher cache hit ratio. The prototype's network overhead is fixed for a given time interval (i.e., independent from the number of queries), which means that the architecture scales well for a large number of queries, but also that it imposes unnecessary overhead when there are only a few queries. In either case, network overhead can be reduced by varying information freshness based on site importance.

Finally, the average information freshness and network overhead models were tested against experimental results. Both models were shown to be fairly accurate across a broad range of settings.

N	setting	D	nupdates			nmchanges			network overhead (MB)		
			esti-mation	measure-ment	rel. diff.	esti-mation	measure-ment	rel. diff.	esti-mation	measure-ment	rel. diff.
1	b	60	26406	25872	−0.02	332281	348668	0.05	147.8	152.9	0.03
2	c1	59	26000	25851	−0.01	327163	348399	0.06	145.5	153.7	0.05
3	c3	59	26000	25976	−0.00	327163	346876	0.06	145.5	154.0	0.05
4	q1	59	26000	25939	−0.00	327163	348290	0.06	145.5	154.2	0.06
5	q3	55	24372	23996	−0.02	306688	345637	0.11	136.4	142.2	0.04
6	q4	55	24372	23995	−0.02	306688	344431	0.11	136.4	141.7	0.04
7	s1	59	6500	6498	−0.00	16358	18812	0.13	7.7	8.0	0.04
8	s2	59	6500	6500	0.00	81790	93854	0.13	36.4	38.5	0.06
9	s3	59	6500	6499	−0.00	163581	188101	0.13	72.3	76.8	0.06
10	s4	59	26000	25994	−0.00	65432	71928	0.09	30.7	32.2	0.05
11	s6	59	26000	25991	−0.00	654326	377103	−0.74	289.0	307.1	0.06
12	s7	59	65000	64364	−0.01	163581	179130	0.09	76.7	79.7	0.04
13	s8	59	65000	64820	−0.00	817907	357036	−1.29	363.8	385.4	0.06
14	s9	65	71101	69972	−0.02	1789372	251232	−6.12	790.4	704.2	−0.12
15	r1	59	26000	25702	−0.01	540140	639980	0.16	238.9	240.8	0.01
16	r3	59	26000	25977	−0.00	214245	228217	0.06	96.0	98.2	0.02
17	u1	60	132909	130708	−0.02	684001	388575	−0.76	310.2	316.4	0.02
18	u3	61	10178	8000	−0.27	181612	164223	−0.11	74.5	67.4	−0.19
19	u4	60	10044	11119	−0.10	179220	159237	0.13	79.3	75.6	0.05

Table 6.8: Network overhead of differential updates: model estimations versus experiment data.

Chapter 7

Conclusions

This chapter concludes the dissertation with a summary of the studied problem and the proposed thesis (Section 7.1), a summary and critique of contributions (Section 7.2), and directions for future work (Section 7.3).

7.1 Problem and Thesis Summary

This dissertation set to investigate monitoring for large-scale grid information services. That is, services that, similarly to web search engines, (i) are expected to resolve resource discovery queries against information about resources throughout the Grid, within a second; (ii) deliver up-to-date results; and (iii) do not impose unacceptably high network overhead.

Existing grid monitoring systems are incapable to scale gracefully for an increasing number of monitored resources, because the imposed network overhead (in prefetching-based hierarchies of aggregation) or the query response time (in just-in-time hierarchies of aggregation) become unacceptably high. The typical solution to the scalability problem is to compromise the freshness of information about all monitored resources (by manually reducing the update frequency in prefetching, or by increasing the cache time-to-live in just-in-time evaluation). In doing so, i.e., treating all resources as equally important, existing systems ignore that resources in a large heterogeneous Grid are unlikely to have equal capabilities (and thus to be of equal usefulness or importance to users).

The proposed thesis addressed the problem by enabling monitoring performance trade-offs to be dealt with at the level of sites as opposed to grid-wide.

Specifically, the thesis re-defined the large-scale grid monitoring problem, by providing:

- a way to quantify the importance of grid resources, which is used as a basis to discriminate sites based on the resources they host;
- the periodic adjustment of update frequency (and thus information freshness), at a site level, based on the importance of resources at every site (as opposed to a static, manually defined, update frequency for all sites);
- an extension of the performance metrics space, with explicit measurements of information freshness and network overhead (in addition to query response time), to quantify the effect of importance-based resource discrimination.

7.2 Review of Contributions

The present dissertation made the following contributions.

- A taxonomy of grid monitoring systems, and its application to a wide range of existing systems and proposals.
- A performance model of monitoring hierarchies of aggregation that use just-in-time evaluation. The model estimates best case QRT performance and imposed network overhead for given monitoring and grid settings.
- A query-independent definition of the relative importance of complex resources that are described using a hierarchical information schema.
- A new approach to large-scale grid monitoring that varies the freshness of information about grid resources based on the importance of the grid site the resources are located at. The investigation of the proposed approach, in addition to the resource importance definition, led to the following contributions:
  - An architecture, based on the web-crawling paradigm, and a prototype implementation of the architecture that realises the proposed approach.

- An experimental evaluation of the proposed monitoring approach, using the prototype implementation, for the purpose of: (i) an empirical study of performance trends; (ii) a comparison of the proposed architecture against JITHAs (just-in-time hierarchies of aggregation); and (iii) an investigation of practical issues to be addressed in a production-quality implementation of the proposed architecture.
- Models for the estimation of the proposed architecture's performance in terms of information freshness and network overhead in given grid and monitoring settings, which were tested against experimental results.

7.2.1 A Taxonomy of Grid Monitoring Systems

The grid monitoring community has invested considerable effort in producing functionality and supporting the ever-changing middleware technology. With the exception of a few experimental studies (e.g., [ZFS05]), comparisons between grid monitoring systems are typically limited to a discussion about functionality and supported middleware, without actual analysis of what makes a certain proposal better over another.

The taxonomy in Chapter 2 is an attempt to identify architectural (i.e., implementation-independent) choices, and use that as a basis to classify and compare the large number of existing grid monitoring and information systems. The taxonomy is useful for understanding how monitoring systems relate in terms of architecture, which to a large extent affects scalability. However, the taxonomy is not meant to be an extensive feature-complete classification (For such a classification see [GWB+04].)

7.2.2 A Performance Model of Just-In-Time Hierarchies of Aggregation

The model in Chapter 3 is intended for studying the performance of just-in-time hierarchies of aggregation (JITHAs), in terms of query response time and network overhead, in various different settings. As is the case with the taxonomy in Chapter 2, the JITHAs model is focused on the performance impact of architectural choices (as opposed to implementation-specific ones). Chapter 3 used the model

for a comparative evaluation of a total of 34 JITHAs in grid settings of 500, 2000 and 5000 grid sites. The main lessons of the study are:

- the average query response time of JITHAs with the same number of levels can vary significantly;
- deep JITHAs (i.e., with many levels) tend to have better average query response time (compared to, say, two-level JITHAs), but up to a certain number of levels.

The model can, in principle, be used by grid administrators to evaluate alternative configurations of JITHAs, for instance, by comparing their corresponding best-case query response times, before actual deployment. In practice, however, a monitoring system's implementation choices may have performance implications that are more significant than those of the chosen JITHA. Anyhow, one can use the model's optimistic estimations of query response time as a baseline.

7.2.3 A Query-Independent Definition of Resource Importance

Chapter 5 defined and evaluated empirically a query-independent definition of resource importance (query-independent means that importance is not defined in relation to a specific query). The proposed definition of resource importance is in no way specific to importance-aware monitoring: it could be used, for instance, for ranking query results. The definition applies to resources of varying complexity (e.g., from CPUs to whole grid sites), but requires that all resources are described using a single hierarchical schema. The latter restriction means that the definition is not applicable for unstructured resource descriptions, such as natural language annotations of web services, or web pages.

7.2.4 Importance-Aware Grid Monitoring using Prefetching

The core contribution of the thesis is the adaptation of the web crawling paradigm in the context of grid monitoring, for the purpose of large-scale grid information services. This adaptation had to take into account the major differences between the two contexts, most notably, the differences between computational grid resources (clusters) and web resources (pages). These differences include:

Frequency of resource changes and monitoring updates The web has billions of pages, most of which typically change infrequently. Moreover, the frequency of change of web pages varies significantly, ranging from many times a day to once every few years, if at all. Web crawlers exploit this diversity to determine the frequency of updates on a page-level basis. In contrast, computational grids, even those envisaged in the present dissertation, are significantly smaller in size. Furthermore, certain properties of clusters (mostly those related to load status) typically change many times a day. In the grid context, frequency of resource changes is not a good basis to determine the frequency of updates, because:

- the variation in frequency of change may not be sufficiently significant (as mentioned above, cluster load changes on a regular basis);
- it would be too expensive to update resources as often as they change because certain cluster properties (most notably the ones related to current status and load) change much more frequently compared to typical web pages.

The present thesis on this issue is that frequency of updates should be based on the importance of the monitored resource, and not on the resource's frequency of changes. This approach raises the problem of measuring grid resource importance in a query-independent way, which motivated the definition in Chapter 5.

Transfer protocol and representation The web is based on a standard transfer protocol (HTTP) and web resources are typically represented using a standard markup language (HTML). In grids, however, there is no consensus neither on transfer protocols nor on resource representation. Proxies in the proposed architecture are intended to mask the heterogeneity in transfer protocols and resource representation, with RDF/XML over HTTP. The thesis did not attempt to address the problem of information modelling heterogeneity, as it is to a large extent, a political matter.

Query performance Given the structured nature of grid resource information, users expect to be able to submit more expressive queries than a list of keywords (which is the case in web search engines). The prototype supports SQL queries, even though production sites may choose to limit the

complexity of user queries (e.g., by limiting the number of conditions in a query). SQL is supported despite grid resources being represented in RDF, by automatic transformation of RDF triples to relational records.

The evaluation in Chapter 6 demonstrated that the prototype implementation of the proposed architecture scales well in terms of query response time, and allows site-level trade-offs between information freshness and imposed network overhead. In very large grid settings (e.g., with more than 20000 monitored clusters) the updates may arrive faster than the data manager can commit them in the database, which underlines the need for methods that enable parallelism across many co-located hosts (e.g., data partitioning across several databases). The evaluation also indicated that the crawling process is not a bottleneck, which eliminates the requirement for distributed crawling, originally set in [DIS04, DSI06].

The experimental evaluation in Planetlab was especially time-consuming, which underlines the need for the proposed models for information freshness and network overhead. The models for average information freshness and the network overhead of differential updates were shown to yield fairly accurate estimates in a broad range of settings, except for certain cases due to implementation issues (e.g., the data manager implementation being a bottleneck in very large grid sizes).

7.3 Future Work

The work in this dissertation can be extended as follows.

- Investigate means to improve the scalability for updating and querying resource information, in very large grid settings. One may consider data partitioning across several databases, or inverted indexes (e.g., [TGMS94]), which are typically used in web search engines for fast keyword matching. The use of inverted indexes is challenged by the need to support range queries (e.g., select a cluster with at least 50 nodes) and frequent updates (e.g., for cluster status information). Another interesting line of investigation is the combined use of inverted indexes and relational databases, for infrequently and frequently changing resource information, respectively.

- The prototype currently assumes that the information provided by proxies

is accurate, even though grid resource providers may modify that information to boost their grid site importance. This suggests that the proposed architecture should be extended to take into account issues of trust between grid sites and monitoring sites. At a more practical level, one may also have to provide confidentiality of potentially sensitive resource information (e.g., current queue size, sharing policy). Similar issues occur in the web, where for instance (i) malicious web sites present different content to users on the one hand, and web crawlers on the other, to boost their page rankings; and (ii) subscription-only web sites allow authorised web crawlers to access their content, on the condition that is only used for indexing, and is otherwise treated confidentially.

- The prototype currently accepts resource discovery queries expressed in SQL. In practice, suitable tools would be needed to support the user to construct such queries, and possibly impose upper bounds in query complexity.

- Investigate adaptations of the definition of resource importance so that it can be applied to resources that are described using schemas of all kinds of graphs (as opposed to the current definition which only supports hierarchical schemas).

- Investigate other areas where the definition of resource importance might be useful. For instance, the definition might be used for the relative ranking of people's profiles in web sites for social networking, job hunting, and dating. One potential extension of the resource importance definition, is the support for quantitative properties in which the most important values lie within a certain range (as opposed to importance being (inversely) proportional to the property's value). A further possible complication in these cases, is that such a range might not be fixed but dependent on the value of one or more other properties.

Appendix A

Network Measurements for the Just-In-Time Hierarchies of Aggregation Model

Table A.1 lists results from the network measurements that were performed as part of the evaluation in Chapter 3. The column "M" denotes the number of parallel transfers; "bytes" denotes the number of bytes of every transfer; and "time" denotes the average time required (in ms) to perform M transfers of the indicated size. For every combination of number of parallel transfers and transfer size, the shown time is the average of up to 240 measurements, evenly spread over 16 evenings.

M	bytes	time (ms)	M	bytes	time (ms)	M	bytes	time (ms)
2	12697	30	4	127795	67	8	640000	622
2	12800	30	4	128000	67	8	1024000	871
2	50790	33	4	253952	111	8	1277952	1033
2	51200	33	4	255590	111	8	1280000	1035
2	64000	34	4	256000	111	8	2560000	1855
2	126976	43	4	319488	132	16	1587	67
2	127795	43	4	320000	132	16	1996	66
2	128000	43	4	507904	200	16	3174	67
2	203161	56	4	512000	201	16	3993	67
2	204800	56	4	638976	241	16	6348	68
2	253952	64	4	640000	241	16	6400	68
2	256000	64	4	1015808	369	16	7987	69
2	507904	108	4	1024000	372	16	12697	72
2	511180	109	4	1277952	458	16	15872	76
2	512000	109	4	1280000	460	16	15974	77
2	640000	130	4	2031616	715	16	19968	82
2	1015808	198	4	2048000	720	16	25600	89
2	1024000	199	4	2555904	893	16	31744	98
2	1277952	238	4	2560000	894	16	39936	109
2	1280000	240	4	5111808	1775	16	63488	290
2	2031616	368	4	5120000	1778	16	63897	300
2	2048000	371	8	1587	220	16	64000	296
2	2555904	456	8	3174	223	16	79872	324
2	2560000	457	8	3993	221	16	126976	321
2	4063232	712	8	6348	224	16	128000	323
2	4096000	718	8	6400	221	16	159744	339
2	5111808	891	8	7987	223	16	253952	426
2	5120000	892	8	12697	222	16	256000	440
2	10223616	1786	8	12800	222	16	319488	504
2	10240000	1784	8	15872	224	16	512000	762
4	1587	35	8	15974	225	16	638976	934
4	3174	35	8	25600	227	16	1277952	1818
4	6348	35	8	31744	232	31	51	74
4	6400	36	8	31948	234	31	204	73
4	12697	36	8	32000	232	31	512	73
4	12800	36	8	39936	238	31	1024	73
4	15872	37	8	51200	247	31	2048	73
4	25395	37	8	63488	253	31	4096	76
4	25600	38	8	63897	255	32	1996	73
4	31744	38	8	64000	256	32	3174	74
4	31948	38	8	79872	251	32	7987	83
4	32000	38	8	126976	294	32	12697	91
4	50790	41	8	128000	294	32	19968	111
4	51200	41	8	159744	314	32	31744	295
4	63488	45	8	253952	380	32	31948	218
4	63897	46	8	256000	380	32	39936	217
4	64000	46	8	319488	417	32	63488	347
4	101580	58	8	320000	418	32	79872	372
4	102400	58	8	512000	540	32	126976	410
4	126976	67	8	638976	621	32	159744	493

32	253952	759	250	4096	510
32	319488	929	312	51	187
32	638976	1811	312	204	190
39	51	77	312	512	193
39	204	77	312	1024	196
39	512	77	312	2048	202
39	1024	78	312	4096	535
39	2048	77	500	51	240
39	4096	81	500	204	247
62	51	815	500	512	252
62	204	817	500	1024	268
62	512	820	500	2048	411
62	1024	819	500	4096	643
62	2048	823	625	51	329
62	4096	829	625	204	330
64	3993	842	625	512	321
64	15974	1093	625	1024	327
64	39936	1460	625	2048	616
64	79872	1707	625	4096	852
64	159744	1861	1000	51	416
64	319488	2529	1000	204	435
78	51	788	1000	512	439
78	204	825	1000	1024	448
78	512	804	1000	2048	548
78	1024	826	1000	4096	990
78	2048	823	1250	51	466
78	4096	833	1250	204	485
125	51	114	1250	512	486
125	204	113	1250	1024	494
125	512	115	1250	2048	598
125	1024	116	1250	4096	1033
125	2048	120	2000	51	671
125	4096	356	2000	204	672
128	1996	122	2000	512	701
128	7987	338	2000	1024	753
128	19968	413	2000	2048	1172
128	39936	810	2000	4096	1423
128	79872	1064	2500	51	765
128	159744	1969	2500	204	803
156	51	127	2500	512	829
156	204	128	2500	1024	918
156	512	132	2500	2048	1407
156	1024	133	2500	4096	2296
156	2048	136	5000	51	1143
156	4096	354	5000	204	1243
250	51	157	5000	512	1192
250	204	160	5000	1024	1790
250	512	174	5000	2048	2721
250	1024	165	5000	4096	7426
250	2048	172			

Table A.1: The network measurements that were used in the evaluation in Chapter 3.

Appendix B

Query Response Time of Various Just-In-Time Hierarchies of Aggregation

Figures B.1-B.4 show the average QRT to 2000 queries, for all the just-in-time hierarchies of aggregation that are considered in Chapter 3, and for all the settings where cache hit ratio is set to 20% and QPT to 25 ms. Specifically, figures B.1-B.4 show the performance of two-, three-, four-, and five-level JITHAs, respectively. Every figure has six plots in three rows. The setting of every plot is indicated on top of it. For instance the top left plot in Figure B.1 is 0.1-0.10-25, which stands for 0.1 KB match size, 10% query selectivity and QPT=25 ms. In general, in all four figures: (i) the plots on the left hand side have query selectivity of 10%, whereas the plots on the right hand side have query selectivity of 40%; and (ii) the plots of the first, second and third row have 0.1, 1, and 2 KB match size, respectively.

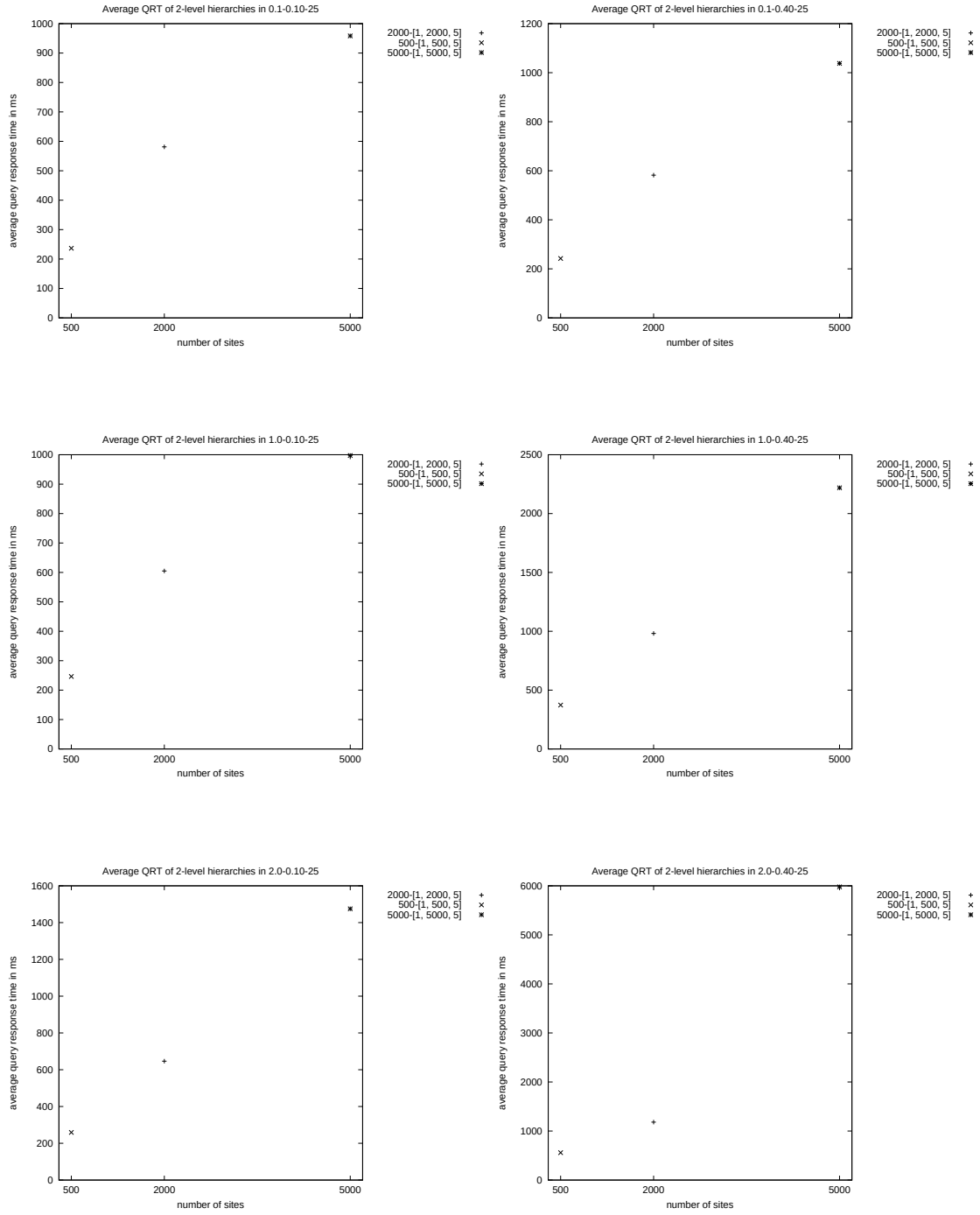

Figure B.1: Number of sites versus QRT, with 2-level hierarchies and 20% cache hit ratio.

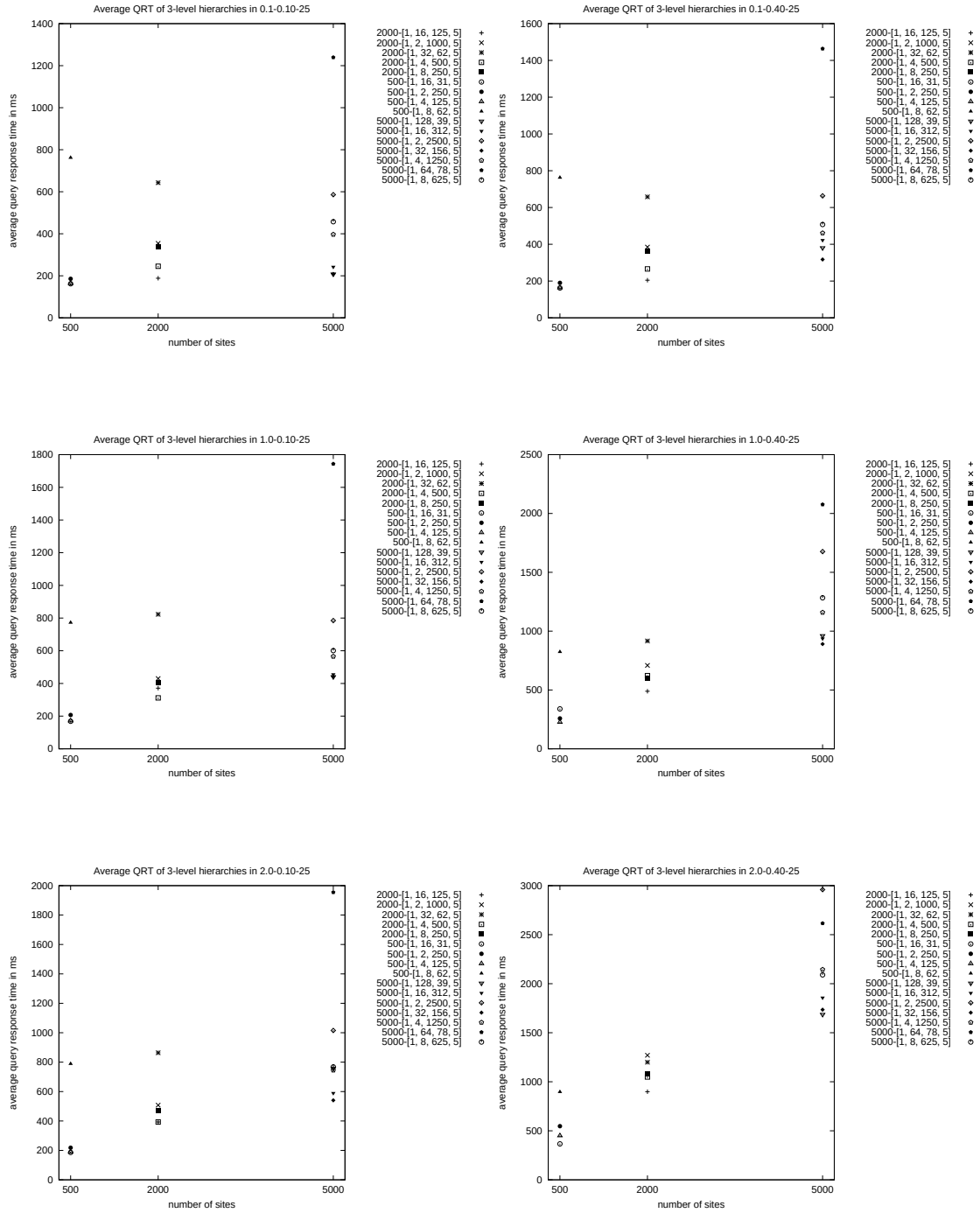

Figure B.2: Number of sites versus QRT, with 3-level hierarchies and 20% cache hit ratio.

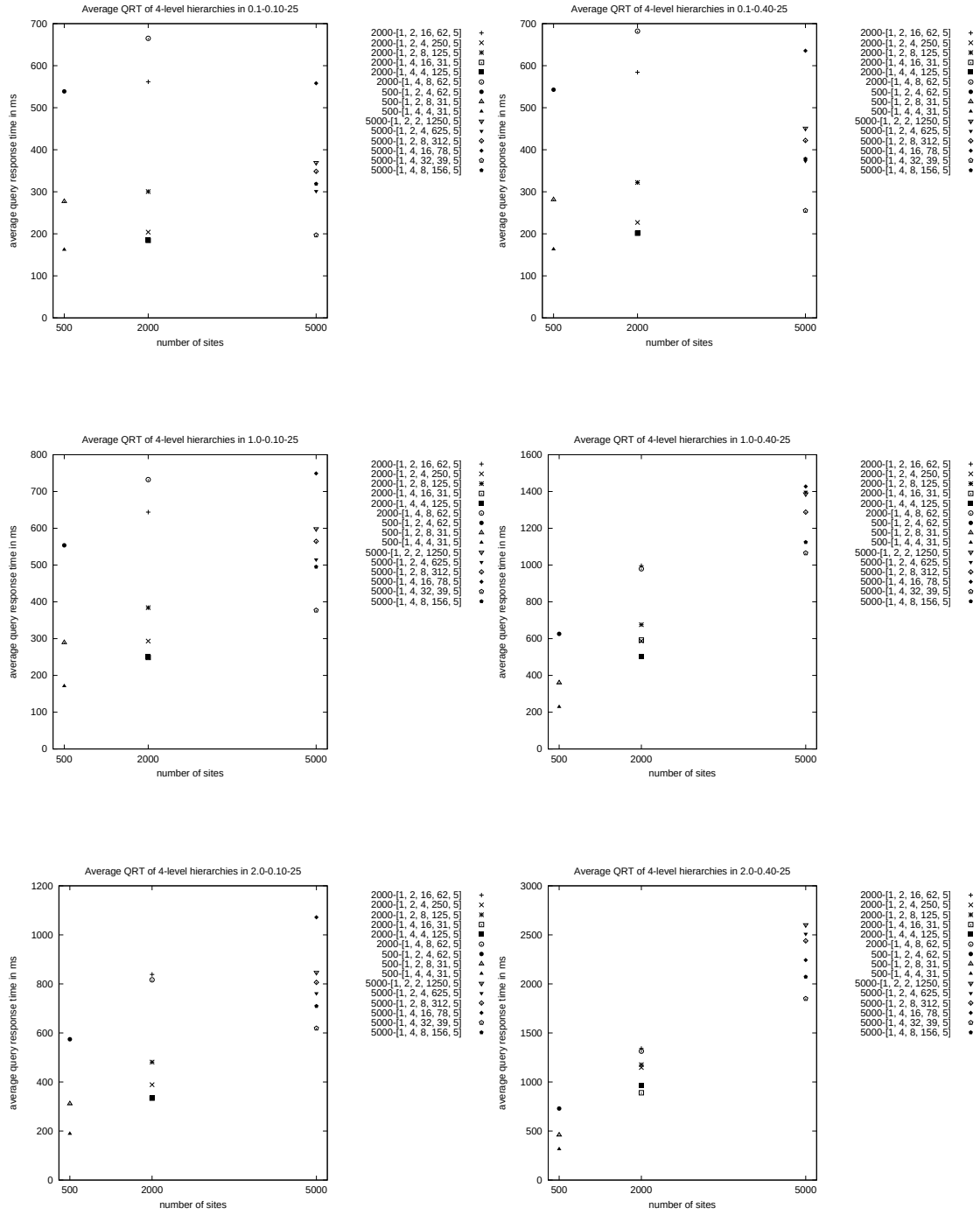

Figure B.3: Number of sites versus QRT, with 4-level hierarchies and 20% cache hit ratio.

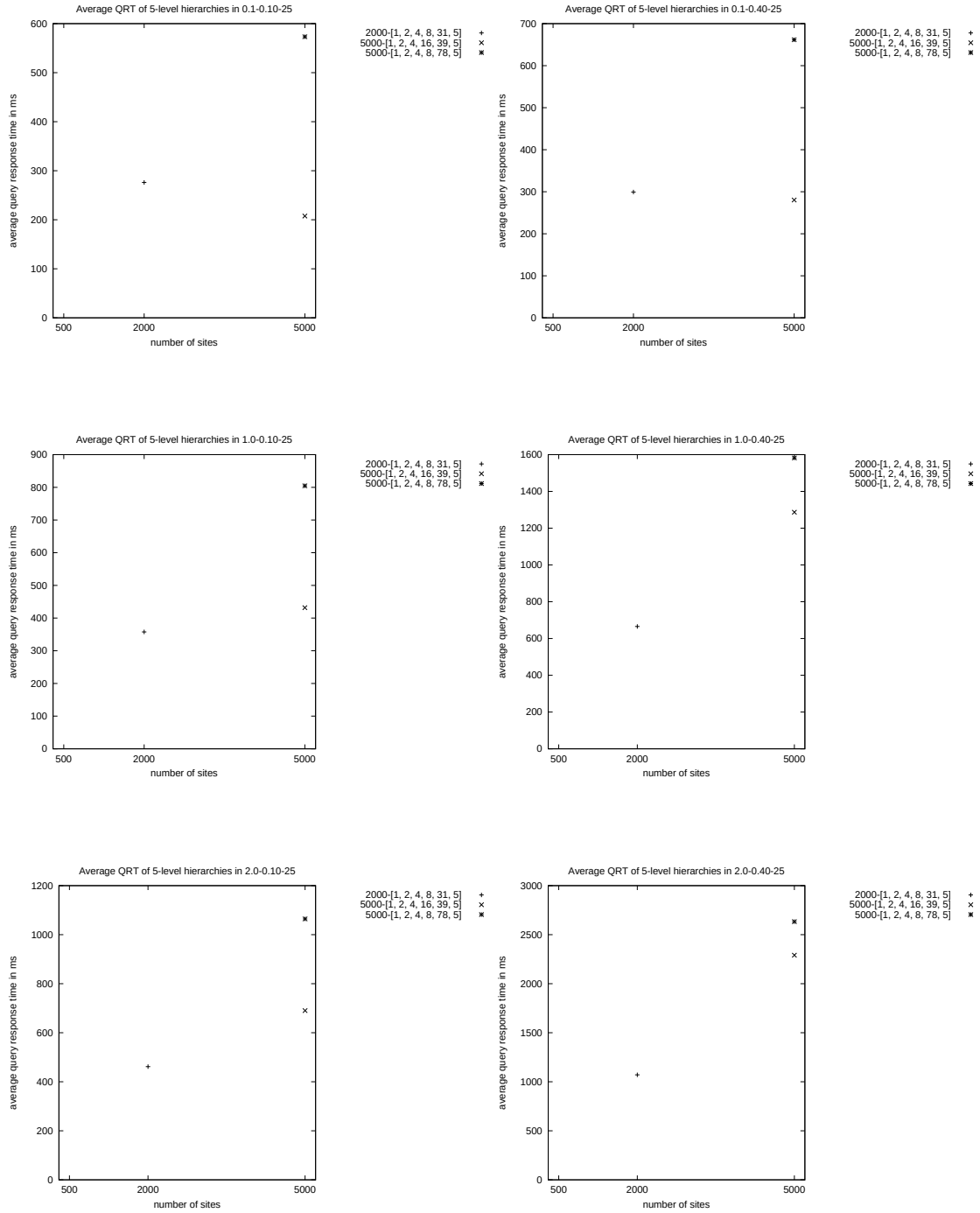

Figure B.4: Number of sites versus QRT, with 5-level hierarchies and 20% cache hit ratio.

Appendix C

Example Calculations of the Network Overhead of Differential Updates

This appendix is an extended version of Section 6.4.2.

Table 6.8 compares the estimation of network overhead due to differential updates, and other relevant metrics, against experimental results.

- The nupdates columns indicate the number of updates (including empty ones) that the crawler downloads during an experiment.

  Estimation According to Equation 4.8 the total number of received updates is $D \times Rm \times$ nsites. The estimations in Table 6.8 increase the outcome of Equation 4.8 by nsites, because sites are not only updated at the end of consecutive time intervals of Mm minutes, but also at the beginning of every experiment. For instance, in the baseline setting

$$\begin{aligned} \text{nupdates} &= D \times Rm \times nsites + nsites \\ &= 60 \times \tfrac{1}{Mm} \times 2000 + 2000 \\ &= 60 \times \tfrac{1}{4.91666} \times 2000 + 2000 \\ &= 26406. \end{aligned}$$

  Measurement Calculated based on experiment logs.

- The nmchanges columns indicate the number of resource changes that are monitored during an experiment.

Estimation Calculated according to Equation 4.11. For instance, in the c1 setting:

$$\begin{aligned} \text{nmchanges} &= \textstyle\sum_{i \in P} \text{nupdates} \times (1 - \text{F}_{\text{i,Rm}}) \times \text{nservs}_\text{s} \\ &= \textstyle\sum_{i \in P} 26000 \times (1 - \text{F}_{\text{i,Rm}}) \times 5 \\ &\simeq 327163 \end{aligned}$$

where $P = [1, 2.5, 5, 10, 20]$ and $Rm = \frac{1}{4.91666}$.

Measurement calculated based on experiment logs.

- Network overhead indicates the network overhead of all the differential updates that are downloaded during an experiment.

  Estimation Calculated according to Equation 4.13, in which the average size of a description of a resource change is 460 bytes, increased by a fixed overhead of 80 bytes per update. For instance, for the c3 setting:

$$\begin{aligned} \text{overhead} = \; & \text{nupdates} \times 80 + \text{nmchanges} \times \text{size}_\text{avg} \\ & = 26000 \times 80 + 327163 \times 460 \\ & \simeq 145.5\,\text{Mbytes}. \end{aligned}$$

  Measurement Calculated based on experiment logs, according to Equation 4.2.

In all three columns, the relative difference is calculated as $\frac{estimated - actual}{actual}$.

The estimations in the table are fairly accurate in most settings (with a relative difference of up to 0.06). In settings u1, s6, s8, and s9, the number of monitored changes (nmchanges) is significantly overestimated, because in these settings the data manager is overwhelmed with incoming updates, many of which are not committed by the end of the experiment. (The metric nmchanges refers to the updates that have been downloaded *and* committed to the database). In the setting u3, both nupdates and nmchanges are overestimated, and as a result the same holds for the estimated diff network overhead.

Bibliography

[ABF+] S. Andreozzi, S. Burke, L. Field, S. Fisher, B. Konya, M. Mambelli, J. M. Schopf, M. Viljoen, and A. Wilson. GLUE Schema Specification, version 1.2. http://infnforge.cnaf.infn.it/glueinfomodel.

[ABF+03] S. Andreozzi, N. De Bortoli, S. Fantinel, A. Ghiselli, G. Tortone, and M.C. Vistoli. GridICE: a Monitoring Service for the Grid. In *3rd Cracow Grid Workshop*, Cracow, Poland, October 27–29 2003.

[ABF+05] S. Andreozzi, N. De Bortoli, S. Fantinel, A. Ghiselli, G. L. Rubini, G. Tortone, and M. C. Vistoli. GridICE: a Monitoring Service for Grid Systems. *Future Generation Computer Systems*, 21(4):559–571, 2005.

[ACGM+01] A. Arasu, J. Cho, H. Garcia-Molina, A. Paepcke, and S. Raghavan. Searching the Web. *ACM Transactions on Internet Technology*, 1(1):2–43, 2001.

[ADD+03] G. Allen, K. Davis, K. N. Dolkas, N. D. Doulamis, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, T. Radke, M. Russell, E. Seidel, J. Shalf, and I. Taylor. Enabling Applications on the Grid - A GridLab Overview. *International Journal of High Performance Computing Applications*, 17(4):449–466, 2003.

[AH06] S. Decker A. Hogan, A. Harth. ReConRank: A Scalable Ranking, Method for Semantic Web Data with Context. In *Second International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2006)*, November 5 2006.

[Ama] Amazon.com. Amazon Simple Storage Service (Amazon S3). http://www.amazon.com/gp/browse.html?node=16427261.

[AX02] A. Andrzejak and Z. Xu. Scalable, efficient range queries for grid information services. In *P2P '02: Proceedings of the Second International Conference on Peer-to-Peer Computing*, page 33, Washington, DC, USA, 2002. IEEE Computer Society.

[Ayd92] R. Aydt. The Pablo Self-Defining Data Format. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, March 1992. (Revised at 17 March 2003).

[BBS+04] B. Balis, M. Bubak, T. Szepieniec, R. Wismüller, and M. Radecki. Monitoring Grid Applications with Grid-enabled OMIS Monitor. In F. F. Rivera, M. Bubak, A. G. Tato, and R. Doallo, editors, *Proceedings of the 1st European Across Grids conference*, volume 2970 of *Lecture Notes in Computer Science*, pages 230–239, Santiago de Compostela, Spain, February 13-14 2004. Springer-Verlag.

[BCC+01] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS Project: Software Support for High-Level Grid Application Development. *International Journal of High Performance Computing Applications*, 15(4):327–344, July 2001.

[BCC+05] R. Byrom, B. Coghlan, A. Cooke, R. Cordenonsi, L. Cornwall, M. Craig, A. Djaoui, A. Duncan, S. Fisher, A. Gray, S. Hicks, S. Kenny, J. Leake, O. Lyttleton, J. Magowan, R. Middleton, W. Nutt, D. OCallaghan, N. Podhorszki, P. Taylor, J. Walk, and A. Wilson. Fault Tolerance in the R-GMA Information and Monitoring System. In *Advances in Grid Computing - European Grid Conference*, volume 3470 of *Lecture Notes in Computer Science*, pages 751–760. Springer, July 2005.

[BDK+02] M. Bhide, P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and . Shenoy. Adaptive Push-Pull: Disseminating Dynamic Web Data. *IEEE Trans. Comput.*, 51(6):652–668, 2002.

[BFGG04] J. Brooke, D. Fellows, K. Garwood, and C. Goble. Semantic Matching of Grid Resource Descriptions. In *Proceedings of the 2nd European Across Grids Conference*, Nicosia, Cyprus, January 28-30 2004.

[BFH03] F. Berman, G. Fox, and T. Hey, editors. *Grid Computing: Making the Global Infrastructure a Reality.* John Wiley & Sons, 2003.

[BG01] Z. Balaton and G. Gombás. GridLab Monitoring: Detailed Architecture Specification. Technical Report GridLab-11-D11.2-01-v1.2, EU Information Society Technologies Programme (IST), 2001.

[BG03] Z. Balaton and G. Gombás. Resource and Job Monitoring in the Grid. In *Proceedings of the 9th International Euro-Par Conference*, volume 2790 of *Lecture Notes in Computer Science*, pages 404–411, Klagenfurt, Austria, August 26-29 2003. Springer-Verlag.

[BH00] B. Buck and J. K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[BHP02] F. Bonnassieux, R. Harakaly, and P. Primet. MapCenter: an Open GRID Status Visualization Tool. In *ISCA 15th International Conference on Parallel and Distributed Computing Systems*, Louisville, Kentucky, USA, September 2002.

[BHP04] F. Bonnassieux, R. Harakaly, and P. Primet. Automatic Services Discovery, Monitoring and Visualization of Grid Environments: the MapCenter Approach. In F. F. Rivera, M. Bubak, A. G. Tato, and R. Doallo, editors, *Proceedings of the 1st European Across Grids conference*, volume 2970 of *Lecture Notes in Computer Science*, pages 222–229, Santiago de Compostela, Spain, February 13-14 2004. Springer-Verlag.

[BKP01] Z. Balaton, P. Kacsuk, and N. Podhorszki. Application Monitoring in the Grid with GRM and PROVE. In *International Conference on Computational Science (ICCS2001), Part I*, volume 2073 of *Lecture Notes in Computer Science*, page 253, San Francisco, CA, USA, May 28-30 2001. Springer-Verlag.

[BKPV01] Z. Balaton, P. Kacsuk, N. Podhorszki, and F. Vajda. From Cluster Monitoring to Grid Monitoring Based on GRM. In R. Sakellariou et al., editors, *Proceedings of the 7th International Euro-Par Conference*, volume 2150 of *Lecture Notes in Computer Science*, pages 874–881, Manchester, UK, August 2001. Springer-Verlag.

[BLC01] T. Berners-Lee and D. Connolly. Delta: An Ontology for the Distribution of Differences Between RDF Graphs, 2001. (12th May 2006 revision).

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web, May 2001.

[Blo70] B. H. Bloom. Space/Time Trade-Offs in Hash Coding With Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[BM05] B. Bamba and S. Mukherjea. *Semantic Web and Databases*, volume 3372 of *Lecture Notes in Computer Science*, chapter Utilizing Resource Importance for Ranking Semantic Web Query Results, pages 185–198. Springer-Verlag, 2005.

[BS03] M. A. Baker and G. Smith. GridRM: An Extensible Resource Monitoring System. In *IEEE Cluster Computing Conference*, page 207, Hong Kong, December 2003.

[CFCS03] M. Cai, M. Frank, J. Chen, and P. Szekely. MAAN: A Multi-Attribute Addressable Network for Grid Information Services. In *GRID '03: Proceedings of the Fourth International Workshop on Grid Computing*, page 184, Phoenix, Arizona, November 17 2003. IEEE Computer Society.

[CFCS04] M. Cai, M. R. Frank, J. Chen, and P. A. Szekely. MAAN: A Multi-Attribute Addressable Network for Grid Information Services. *J. Grid Comput.*, 2(1):3–14, 2004.

[CFFK01] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid Information Services for Distributed Resource Sharing. In *Proceedings of the 10th IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*, pages 181–194, San Francisco, CA, 7-9 August 2001. IEEE Computer Society Press.

[CFSD90] J. D. Case, M. Fedor, M. L. Schoffstall, and C. Davin. Simple Network Management Protocol (SNMP), RFC 1157, May 1990.

[CGM03a] J. Cho and H. Garcia-Molina. Effective Page Refresh Policies for Web Crawlers. *ACM Trans. Database Syst.*, 28(4):390–426, 2003.

[CGM+03b] A. Cooke, A. J. G. Gray, L. Ma, W. Nutt, J. Magowan, M. Oevers, P. Taylor, R. Byrom, L. Field, S. Hicks, J. Leake, M. Soni, A. Wilson, R. Cordenonsi, L. Cornwall, A. Djaoui, S. Fisher, N. Podhorszki, B. Coghlan, S. Kenny, and D. O'Callaghan. R-GMA: An Information Integration System for Grid Monitoring. In *Proceedings of the 10th International Conference on Cooperative Information Systems*, 2003.

[CGN+04] A. W. Cooke, A. J. G. Gray, W. Nutt, J. Magowan, M. Oevers, P. Taylor, R. Cordenonsi, R. Byrom, L. Cornwall, A. Djaoui, L. Field, S. M. Fisher, S. Hicks, J. Leake, R. Middleton, A. Wilson, X. Zhu, N. Podhorszki, B. Coghlan, S. Kenny, D. O. Callaghan, and J. Ryan. The Relational Grid Monitoring Architecture: Mediating Information about the Grid. *Journal of Grid Computing*, 2(4):323–339, December 2004.

[cro] The EU CrossGrid Project. (accessed on 27th November 2003).

[CZH+99] S. E. Czerwinski, B. Y. Zhao, T. D. Hodes, A. D. Joseph, and R. H. Katz. An architecture for a secure service discovery service. In *MobiCom '99: Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 24–35, New York, NY, USA, 1999. ACM Press.

[dBKB02] M. den Burger, T. Kielmann, and H. E. Bal. TopoMon: A Monitoring Tool for Grid Network Topology. In *International Conference on Computational Science (ICCS 2002)*, volume 2230, pages 558–567. Springer-Verlag, 2002.

[DGK+01] P. Dinda, T. Gross, R. Karrer, B. Lowekamp, N. Miller, P. Steenkiste, and D. Sutherland. The Architecture of the Remos System. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10*

2001), pages 252–265, San Francisco, CA, USA, August 7–9 August 2001. IEEE Computer Society Press.

[DGL+98] T. DeWitt, T. Gross, B. Lowekamp, N. Miller, P. Steenkiste, J. Subhlok, and D. Sutherland. ReMoS: A Resource Monitoring System for Network-Aware Applications. Technical Report CMU-CS-97-194, School of Computer Science, Carnegie Mellon University, December 1998.

[DIS04] M. Dikaiakos, Y. Ioannidis, and R. Sakellariou. Search Engines for the Grid: A Research Agenda. In F. F. Rivera, M. Bubak, A. G. Tato, and R. Doallo, editors, *Proceedings of the 1st European Across Grids Conference*, volume 2970 of *Lecture Notes in Computer Science*, pages 49–58, Santiago de Compostela, Spain, February 13-14 2004. Springer-Verlag.

[DKP+01] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive Push-Pull: Disseminating Dynamic Web Data. In *Proceedings of the 10th International Conference on World Wide Web*, pages 265–274, Hong Kong, Hong Kong, 2001. ACM Press.

[DL05] P. Dinda and D. Lu. Fast Compositional Queries in a Relational Grid Information Service. *Journal of Grid Computing*, 3:131–150, June 2005.

[DO99] P. Dinda and D. O'Hallaron. An Extensible Toolkit for Resource Prediction in Distributed Systems. Technical Report CMU-CS-99-138, School of Computer Science, Carnegie Mellon University, July 1999.

[DSI06] M. Dikaiakos, R. Sakellariou, and Y. Ioannidis. Information Services for Large-scale Grids: A Case for a Grid Search Engine. In J. Dongarra, H. Zima, A. Hoisie, L. Yang, and B. DiMartino, editors, *In Engineering the Grid: Status and Perspectives*. American Scientific Publishers, January 2006.

[FFK+97] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A Directory Service for Configuring High-performance Distributed Computations. In *Proceedings of the 6th*

IEEE Symposium on High Performance Distributed Computing, pages 365–375. IEEE Computer Society Press, 1997.

[FK99a] I. Foster and C. Kesselman. Globus: A Toolkit-Based Grid Architecture. In *The Grid: Blueprint for a Future Computing Infrastructure*, pages 259–278. Morgan-Kaufmann, 1999.

[FK99b] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan-Kaufmann, 1999.

[FK03] I. Foster and C. Kesselman, editors. *The Grid 2: Blueprint for a Future Computing Infrastructure*. Morgan-Kaufmann, 2003.

[FKNT02] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Global Grid Forum, June 2002.

[FKT01] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputing Applications*, 15(3):115–128, 2001.

[FKTT98] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *ACM Conference on Computers and Security*, pages 83–91. ACM Press, 1998.

[GMB05] G. Gombás, C. A. Marosi, and Z. Balaton. Grid Application Monitoring and Debugging Using the Mercury Monitoring System. In *Advances in Grid Computing – European Grid Conference 2005*, volume 3470, pages 193–199. Springer, 2005.

[gra93] Grand Challengers 1993: High Performance Computing and Communications. Technical report, 1993. Committee on Physical, Mathematical, and Engineering Sciences, Federal Coordinating Council for Science, Engineering, and Technology, Office of Science and Technology Policy. Supplement to the President's Fiscal Year 1993 Budget.

[Gra03] Jim Gray. Distributed Computing Economics. Technical Report MSR-TR-2003-24, Microsoft Research, March 2003.

[GTJ+02] D. Gunter, B. Tierney, K. Jackson, J. Lee, and M. Stoufer. Dynamic Monitoring of High-Performance Distributed Applications. In *Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing, HPDC-11*, page 163, Edinburgh, Scotland, 23-26 July 2002. IEEE Computer Society Press.

[GTTV03] D. Gunter, B. L. Tierney, C. E. Tull, and V. Virmani. On-Demand Grid Application Tuning and Debugging with the NetLogger Activation Service. In *GRID '03: Proceedings of the Fourth International Workshop on Grid Computing*, page 76, Phoenix, Arizona, November 17 2003. IEEE Computer Society.

[GWB+04] M. Gerndt, R. Wismüller, Z. Balaton, G. Gombás, Z. Németh, N. Podhorski, H.-L. Truong, T. Fahringer, M. Bubak, E. Laure, and T. Margalef. Performance Tools for the Grid: State of the Art and Future. Technical report, Lehrstuhl fuer Rechnertechnik und Rechnerorganisation, Technische Universitaet Muenchen (LRR-TUM), January 2004.

[HAMAS04] C. Halaschek, B. Aleman-Meza, I. Arpinar, and A. Sheth. Discovering and Ranking Semantic Associations over a Large RDF Metabase. In M. Nascimento, editor, *Proceedings of the 30th VLDB Conference*. Morgan Kaufman, 2004.

[haw] Hawkeye, A Monitoring and Management Tool for Distributed Systems. www.cs.wisc.edu/condor/hawkeye/ (accessed on 27th October 2003).

[IF01] A. Iamnitchi and I. Foster. On Fully Decentralized Resource Discovery in Grid Environments. In *Proceedings of the Second International Workshop on Grid Computing*, pages 51–62, London, UK, 2001. Springer-Verlag.

[IF04] A. Iamnitchi and I. Foster. *A Peer-To-Peer Approach to Resource Location in Grid Environments*, pages 413–429. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[IW97] T. Ludwig II and R. Wismüller. OMIS 2.0 - A Universal Interface for Monitoring Systems. In M. Bubak, J. Dongarra, and J. Wasniewski,

editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 4th European PVM/MPI Users' Group Meeting*, volume 1332 of *Lecture Notes in Computer Science*, pages 267–276, Crakow, Poland, November 3-5 1997. Springer-Verlag.

[KCC04] Y. S. Kee, H. Casanova, and A. A. Chien. Realistic Modeling and Synthesis of Resources for Computational Grids. In *Supercomputing 2004*, November 6–12 2004.

[KD03a] H.N.L.C. Keung, J.R.D. Dyson, S.A. Jarvis , and Nudd , G.R. Performance Evaluation of a Grid Resource Monitoring and Discovery Service. *IEE Proceedings-Software*, 150(4):243–251, August 2003.

[KD03b] H.N.L.C. Keung, J.R.D. Dyson, S.A. Jarvis , and Nudd , G.R. Predicting the Performance of Globus Monitoring and Discovery Service (MDS-2) Queries. In *Fourth International Workshop on Grid Computing*, pages 176–183. IEEE, November 17 2003.

[Kle99] J. M. Kleinberg. Authoritative Sources in a Hyperlinked Environment. *J. ACM*, 46(5):604–632, 1999.

[KLH+05] Y.-S. Kee, D. Logothetis, R. Huang, H. Casanova, and A. Chien. Efficient Resource Description and High Quality Selection for Virtual Grids. In *Proceedings of the 5th IEEE Symposium on Cluster Computing and the Grid (CCGrid'05)*. IEEE, 2005.

[LCG] LCG – LHC Computing Grid Project. http://lcg.web.cern.ch/LCG/.

[LLM88] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, San Jose, California, June 13-17 1988.

[LM05] A. Langville and C. Meyer. A Survey of Eigenvector Methods of Web Information Retrieval, 2005.

[MCC04] M. L. Massie, B. N. Chun, and D. E. Culler. Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7), July 2004.

[MS92] M. Mansouri-Samani. Monitoring Distributed Systems (A Survey). Technical Report DOC92/23, Imperial College, London, UK, 1992.

[MSS93] M. Mansouri-Samani and M. Sloman. Monitoring Distributed Systems. *IEEE Network*, 7(6):20–30, November 1993.

[NLB01] H. B. Newman, I. C. Legrand, and J. J. Bunn. A Distributed Agent-based Architecture for Dynamic Services. In *Computing in High Energy and Nuclear Physics (CHEP01)*, 2001.

[NLG+03] H. B. Newman, I. C. Legrand, P. Galvez, R. Voicu, and C. Cirstoiu. MonALISA: A Distributed Monitoring Service Architecture. In *Computing in High Energy and Nuclear Physics (CHEP03)*, La Jolla, California, 2003.

[PACR03] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. *SIGCOMM Comput. Commun. Rev.*, 33(1):59–64, 2003.

[PBG04] N. Podhorszki, Z. Balaton, and G. Gombás. Monitoring Message-Passing Parallel Applications in the Grid with GRM and Mercury Monitor . In *European Across Grids Conference*, volume 3165 of *Lecture Notes in Computer Science*, pages 179–181. Springer, 2004.

[PBMW99] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report SIDL-WP-1999-0120, Stanford Digital Library Technologies Project, November 11 1999.

[PDH+02] B. Plale, P. Dinda, M. Helm, G. von Laszewski, and J. McGee. Key Concepts and Services of a Grid Information Service. In *Proceedings of 15th International Conference on Parallel and Distributed Computing Systems (PDCS 2002)*, Louisville, KY, September 2002.

[PJJ+04] B. Plale, C. Jacobs, S. Jensen, Y. Liu, C. Moad, R. Parab, and P. Vaidya. Understanding Grid Resource Information Management through a Synthetic Database Benchmark/Workload. In *4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2004)*, pages 277–284, April 19-22 2004.

[PJL+03] B. Plale, C. Jacobs, Y. Liu, C. Moad, R. Parab, and P. Vaidya. Benchmark Details of Synthetic Database Benchmark/Workload for Grid Resource Information. Technical Report TR583, Indiana University, Computer Science, August 2003.

[PK00] N. Podhorszki and P. Kacsuk. Design and Implementation of a Distributed Monitor for Semi-on-line Monitoring of VisualMP Applications. In *Proceedings of the 3rd Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS2000): From Instruction Parallelism to Cluster Computing*, pages 23–32, Hungary, 2000.

[Pow03] S. Powers. *Practical RDF*. O'Reilly, 2003.

[rdf04a] RDF Vocabulary Description Language 1.0: RDF Schema (W3C Recommendation), February 10 2004. http://www.w3.org/TR/rdf-concepts/.

[rdf04b] RDF/XML Syntax Specification (Revised) (W3C Recommendation), February 10 2004. http://www.w3.org/TR/rdf-syntax-grammar/.

[rdf04c] Resource Description Framework (RDF): Concepts and Abstract Syntax (W3C Recommendation), February 10 2004. http://www.w3.org/TR/rdf-concepts/.

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-Addressable Network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM Press.

[RHB06] R. Ranjan, A. Harwood, and R. Buyya. A Study on Peer-to-Peer Based Discovery of Grid Resource Information. Technical Report GRIDS-TR-2006-17, P2P Networks Group and Grid Computing and Distributed Systems Laboratory, The University of Melbourne, November 10 2006.

[Rit01] Jordan Ritter. Why Gnutella Can't Scale. No, Really., February 2001. http://www.darkridge.com/ jpr5/doc/gnutella.html.

[RLS98] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, page 140, Chicago, Illinois, USA, 28-31 July 1998. IEEE Computer Society Press.

[RM05] D. A. Reed and C. L. Mendes. Intelligent Monitoring for Adaptation in Grid Applications. *Proceedings of the IEEE*, 93(2):426–435, February 2005.

[RVSR98] R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed. Autopilot: Adaptive Control of Distributed Applications. In *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing*, pages 172–179, July 1998.

[SC92] Larry Smarr and Charles E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992.

[SFK+98] P. Stelling, I. Foster, C. Kesselman, C. Lee, and G. von Laszewski. A Fault Detection Service for Wide Area Distributed Computations. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, pages 268–278, Chicago, IL, 28-31 1998.

[SGQ02] W. Smith, D. Gunter, and D. Quesnel. A Simple XML Producer/Consumer Protocol, GWD-Perf-8-2, Global Grid Forum, 2002.

[SKL+03] R. Sundaresanand, T. Kurcand, M. Lauria, S. Parthasarathyand, and J. Saltz. A Slacker Coherence Protocol for Pull-based Monitoring of On-line Data Sources. In *Proceeedings of 3rd International Symposium on IEEE/ACM Cluster Computing and the Grid (CCGrid03)*, pages 250–257, Tokyo, Japan, May 12–15 2003. IEEE Computer Society Press.

[SLK+03] R. Sundaresanand, M. Lauriaand, T. Kurcand, S. Parthasarathyand, and J. Saltz. Adaptive Polling of Grid Resource Monitors Using a Slacker Coherence Model. In *12th IEEE International Symposium on High Performance Distributed*

Computing (HPDC'03), page 260, Seattle, Washington, June 22-24 2003. IEEE Computer Society Press.

[Smi01] W. Smith. A Framework for Control and Observation in Distributed Environments. Technical Report NAS-01-006, NASA Advanced Supercomputing Division, NASA Ames Research Center, July 2001.

[Smi02] W. Smith. A System for Monitoring and Management of Computational Grids. In *Proceedings of the 31st International Conference on Parallel Processing (ICPP2002)*, page 55. IEEE Computer Society Press, August 2002.

[SMLN+03] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a Scalable Peer-To-Peer Lookup Protocol For Internet Applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[SRP+06] J. M. Schopf, I. Raicu, L. Pearlman, N. Miller, C. Kesselman, I. Foster, and M. D'Arcy. Monitoring and Discovery in a Web Services Framework: Functionality and Performance of Globus Toolkit MDS4. Technical Report MCS Preprint ANL/MCS-P1315-0106, Mathematics and Computer Science Division, Argonne National Laboratory, January 2006.

[SRS03] N. Stojanovic and L. Stojanovic R. Studer. An Approach for the Ranking of Query Results in the Semantic Web. In *International Semantic Web Conference*, pages 500–516, October 2003.

[SWMY00] W. Smith, A. Waheed, D. Meyers, and J. Yan. An Evaluation of Alternative Designs for a Grid Information Service. In *Proceedings of the Ninth International Symposium on High-Performance Distributed Computing*, pages 185–192. IEEE, 2000.

[TAG+02] B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, and R. Wolski. A Grid Monitoring Architecture, GWD-Perf-16-3, Global Grid Forum, August 2002.

[TCF+03] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, and P. Vanderbilt.

Open Grid Services Infrastructure (OGSI), Version 1.0. Technical report, Global Grid Forum, June 2003.

[TD07] G. Tsouloupas and M. D. Dikaiakos. Grid Resource Ranking Using Low-level Performance Measurements. In *13th International Euro-Par Conference*, volume 4641 of *Lecture Notes in Computer Science*, pages 467–476, Rennes, France, August 28–31 2007. Springer-Verlag.

[ter] TeraGrid Hardware Resources: Compute and Visualization Resources. http://www.teragrid.org/userinfo/hardware/resources.php.

[TF03a] H. L. Truong and T. Fahringer. SCALEA: A Performance Analysis Tool for Parallel Programs. *Concurrency and Computation: Practice and Experience*, 15(11-12):1001–1025, 2003.

[TF03b] H. L. Truong and T. Fahringer. SCALEA-G: A Unified Monitoring and Performance Analysis System for the Grid. Technical report, Institute for Software Science, University of Vienna, November 2003.

[TF04] H. L. Truong and T. Fahringer. SCALEA-G: A Unified Monitoring and Performance Analysis System for the Grid. *Scientific Programming*, 12(4):225–237, 2004.

[TG03] B. Tierney and D. Gunter. NetLogger: A Toolkit for Distributed System Performance Tuning and Debugging. In G. S. Goldszmidt and J. Schönwälder, editor, *IFIP/IEEE 8th International Symposium on Integrated Network Management (IM 2003)*, volume 246 of *IFIP Conference Proceedings*, pages 97–100. Kluwer, March 2003.

[TGL+03] C. E. Tull, D. Gunter, W. Lavrijsen, D. Quarrie, and B. Tierney. GMA Instrumentation of the Athena Framework using NetLogger. In *Computing in High Energy and Nuclear Physics (CHEP03)*, La Jolla, California, June 2003.

[TGMS94] A. Tomasic, H. Garcia-Molina, and K. Shoens. Incremental Updates of Inverted Lists for Text Document Retrieval. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on*

Management of Data, pages 289–300, New York, NY, USA, 1994. ACM Press.

[VAMR01] F. Vraalsen, R. A. Aydt, C. L. Mendes, and D.A. Reed. Performance Contracts: Predicting and Monitoring Grid Application Behavior. In *Proceedings of the 2nd International Workshop on Grid Computing*, Lecture Notes in Computer Science, Denver, CO, November 12 2001. Springer-Verlag.

[Wal99] J. Waldo. The Jini Architecture for Network-Centric Computing. *Communications of the ACM*, 42(7):76–82, 1999.

[WHK97] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3), RFC 2251, December 1997.

[WMOS03] R. Wolski, L. J. Miller, G. Obertelli, and M. Swany. *Grid Resource Management, State of the Art and Future Trends*, chapter Performance Information Services for Computational Grids, pages 193–213. Kluwer Academic Publishers, 2003.

[Wol03] R. Wolski. Experiences with Predicting Resource Performance On-Line in Computational Grid Settings. *SIGMETRICS Performance Evaluation Review*, 30(4):41–49, 2003.

[WSH99] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computer Systems*, 15(5-6):757–768, October 1999.

[ZFS03] X. Zhang, J. Freschl, and J. Schopf. A Performance Study of Monitoring and Information Services for Distributed Systems. In *Proceedings of 12th IEEE High Performance Distributed Computing (HPDC-12 2003)*, pages 270–282, Seattle, WA, USA, 22-24 June 2003. IEEE Computer Society Press.

[ZFS05] X. Zhang, J. L. Freschl, and J. M. Schopf. Scalability Analysis of Three Monitoring and Information Systems: MDS2, R-GMA, and Hawkeye. Technical Report Preprint ANL/MCS-P1294-1005, Mathematics and Computer Science Division, Argonne National Laboratory, 2005.

[ZKJ01] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report CSD-01-1141, University of California at Berkeley, Berkeley, CA, USA, 2001.

[ZS04] S. Zanikolas and R. Sakellariou. Towards a Monitoring Framework for Worldwide Grid Information Services. In *10th International Euro-Par Conference*, volume 3149 of *Lecture Notes in Computer Science*, pages 417–422, Pisa, Italy, August 31–September 3 2004. Springer-Verlag.

[ZS05] S. Zanikolas and R. Sakellariou. A Taxonomy of Grid Monitoring Systems. *Future Generation Computer Systems*, 21(1):163–188, January 2005.