Parallel Processing Letters, Vol. 18, No. 3 (2008) 347–370

World Scientific
www.worldscientific.com

AN IMPORTANCE-AWARE ARCHITECTURE FOR LARGE-SCALE GRID INFORMATION SERVICES

SERAFEIM ZANIKOLAS and RIZOS SAKELLARIOU
School of Computer Science, The University of Manchester
Oxford Road, Manchester M13 9PL, U.K.
{*zanikolas,rizos*}*@cs.man.ac.uk*

Received May 2008
Revised July 2008
Communicated by P. Fragopoulou

ABSTRACT

This paper is concerned with the scalability of large-scale grid monitoring and information services, which are mainly used for the discovery of resources of interest. 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 paper is motivated by the observation that existing grid monitoring systems can only be manually configured for a trade-off among the three metrics, which applies equally to all monitored resources; this implies that all resources in a grid are considered to be of equal importance. Assuming that in a large-scale grid setting this is unlikely to hold, the paper proposes an importance-based monitoring architecture for large-scale grid information services, based on an adaptation of the web crawling paradigm. The main idea is that, since not all resources are of equal importance, one can vary the trade-off based on the relative importance of the monitored resources. The proposed architecture is described and evaluated based on large-scale deployments of a prototype implementation on PlanetLab.

Keywords: Grid Information Services, Large-Scale Grids, Importance-Aware Prefetching, Grid Monitoring

1. Introduction

The state of Grid computing, despite significant advances, is far from the original vision of a unified, worldwide Grid which would serve as a single platform for online resource sharing. Numerous grids have been deployed across the world, but are being used in isolation. The fragmentation of existing grid deployments reduces the overall value of the Grid as a sharing platform. This fragmentation of grids is due to a number of both technical and political issues that need to be addressed, such as *scalability* and *interoperability*. Generally speaking, existing middleware appears to be incapable of operating seamlessly for increasing numbers of grid administrative domains, resources and users, whereas, there is a lack of integration between

competing or overlapping software development efforts.

Of the above issues, this paper focuses on the scalability problem, specifically for grid monitoring and information services. 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 key observation underpinning the work in this paper is that existing grid monitoring systems are essentially configured to provide a specific 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. However, in a large grid this may not be likely to hold; some resources may be more important than others.

Thus, the main contribution of this paper is a proposal for an importance-based monitoring architecture for large-scale grid information services, which is based on an adaptation of the web crawling paradigm. Since not all resources are of equal importance, the key idea is that one can vary the trade-off separately for each resource, depending 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 the sites hosting the least important resources. A prototype of the proposed architecture has been implemented and experimentally evaluated.

The remainder of the paper is organised as follows. Section 2 provides some background related to the proposal in this paper and highlights some relevant work from the literature. Section 3 presents the proposed architecture for importance-aware grid monitoring and describes the functionality of different components. Section 4 evaluates experimentally the proposed architecture, based on large-scale deployments of a prototype implementation on PlanetLab. Section 5 reviews the paper's contributions, and discusses potential future work.

2. Background and Related Work

Existing grid monitoring and information services typically have a hierarchical (i.e., tree) structure. High level nodes collect and republish resource information from their child nodes (i.e., immediate descendants). Nodes at the lowest level (i.e., leaf nodes of the tree) are typically responsible for collecting resource information from a specific grid site. Users (with the right permissions) can query any of the nodes in the hierarchy for specific resource information. Every node maintains a local cache, where it caches resource information that it receives from its child nodes. A node may be configured to periodically collect all the information that is available from its child nodes, or may collect (and cache) resource information only upon the arrival of a query. The former approach answers queries making use of *prefetching*, whereas the latter approach is based on *just in time* evaluation of the query.

The performance of grid monitoring and information services is mainly measured in terms of: (i) how quickly queries are answered (*query response time*); (ii) how

much network traffic a service generates to answer user queries (*network overhead*); and (iii) 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 at the same time: design choices affect 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.

The grid community has developed several monitoring and information services [8, 17, 25, 24, 4, 1, 7]. These can be manually configured (e.g., via a static configuration file) for either prefetching or just-in-time evaluation (or a combination of the two). They are also configured to support a trade-off among the aforementioned performance metrics; this trade-off applies *equally* to all monitored grid resources. The performance of grid monitoring and information services has been the subject of several experimental studies. Some studies evaluate a single implementation [21, 16, 15], while others compare several systems [20, 19, 28, 29]. Some of the findings indicate that current hierarchical structures cannot guarantee a consistently low Query Response Time. Although prefetching may improve Query Response Time it may impose a significant network overhead. Experimental studies indicate that the latter 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. To strike a balance for overall good performance, the predominant view is that hierarchical structures for grid information services should be "rather narrow and deep" [29].

Going beyond the current state-of-the-art, the architecture proposed in this paper aims to provide additional flexibility by using centrally-controlled prefetching, which makes use of a dynamically adjusted frequency of updates for each grid site, depending on the site's importance. The proposals in [14, 12] are similar to the one in this paper in that they also collect grid-wide resource information at a central relational database. However, both proposals collect detailed host and network related information (as opposed to the collections per grid site used in our proposal) and do not prescribe how monitoring is performed nor do they cope with varying frequencies of updates. Sundaresan *et al.* [23, 22] propose a host monitor in which the frequency of updates is dynamically adjusted, and information freshness is explicitly measured. The frequency of updates 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; thus, the frequency of updates becomes correlated to the frequency of resource changes, something that may not be desirable in the case of frequently changing properties. In contrast, our proposal provides more flexibility by adjusting the frequency of updates based on specific information about the importance of the resources of a grid site. None of the cited systems has a notion of resource importance, and hence none allows to treat resources differently based

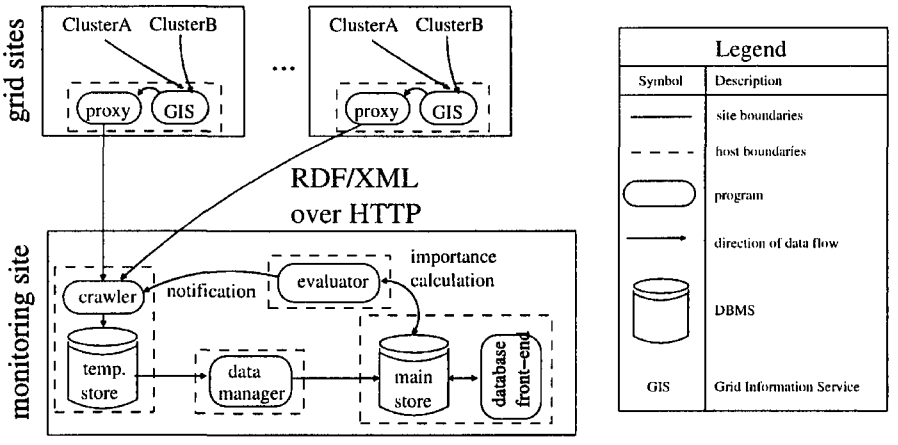

Fig. 1. Overall system architecture.

on their importance.

An important feature of this paper's proposal 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 [9, 5] 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 [9, 5] 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 paper 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.

3. An Architecture Based on Importance-Aware Prefetching

Assuming that in a large-scale grid one may reasonably expect that not all resources are of equal 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 (hence, maintaining higher freshness) for the sites that host the most important resources, while lower freshness is maintained for the sites hosting the least important resources. Furthermore, the frequency of updates per site can be dynamically adjusted by periodically re-evaluating the importance of all known grid resources.

This approach has been realized by the architecture shown in Figure 1. In its

abstract form, this can be seen as an adaptation of the web crawling paradigm [3] for the purpose of large-scale grid monitoring [10, 11, 27]. 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 explained next.

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. 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 1). The purpose of proxies is to provide a contact point with a consistent network interface and resource representation for every grid site. The proposed architecture is intended to operate on top of existing information services; thus, to overcome interoperability problems due to the diversity of existing grid information services, a simplified version of the GLUE information model [2] has been adapted (see [26, Section 4.4]).

Proxies provide information about their local resources upon request, via standard HTTP (HyperText Transfer Protocol). Four types of queries to proxies can be made: (i) provide all available resource information; (ii) provide all available information about a specific resource; (iii) provide all available resource information, excluding data that has not changed since a specified timestamp; and (iv) provide all available information about a specific resource, excluding data (about this resource) that has not changed since a specified timestamp. The response to queries of the first two types is a *full update* (either for all the resources or for specific resources), whereas the response to queries of the last two types is a *differential update*. Differential updates are implemented using a standard if-modified-since HTTP header. Their purpose is to minimize the network overhead by not collecting the same information more than once; that is, information that is already available at the monitoring site and has not changed since the last query to the proxy was made (hence, it is 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.

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

```
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 }
```

Fig. 2. High-level pseudo-code of the crawler.

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. This work does not explicitly deal with proxy discovery by the monitoring sites. In the simplest case, it can be assumed that grid site administrators manually submit their sites' proxy addresses to a monitoring site. Alternatively, a monitoring site could use an automated method that scans IP addresses for certain services (e.g., [6]).

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 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 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 2) retrieves from the main store (as discussed in the paragraph "Storage" of this section, this is a relational database): (i) the address of every site's proxy; (ii) the importance of every site; and, (iii) 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. The crawling process is described in lines 3-13 of Figure 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 (the `window` is meant to mitigate potential delays between requesting an update from a proxy until a response has been received.) 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. These values are between 0 and 1 and they are used to re-calculate the update interval of every site (lines 14-17 in Figure 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.

The crawler implements two ways of mapping site importance to frequency of updates: *topn* and *proportional*. In *topn mode*, the crawler operator specifies the frequency of 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 remaining 90% every 20 minutes). In *proportional mode*, the crawler operator specifies the minimum and maximum allowed frequency (say min_f and max_f) and the crawler dynamically maps importance (say imp) to an appropriate frequency (say f) between the minimum and maximum. This is based on standard linear mapping, i.e., $f = min_f + (1 - imp) \times (max_f - min_f)$.

Data Manager The resource information, as provided by proxies, is encoded in RDF/XML, 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 calculates the importance of all sites based on a predetermined criterion. This criterion is orthogonal to the description of the architecture and the reader is referred to [26, Ch. 5] for further information. In this paper, we only need to assume that importance values range from 0 (least important) to 1 (most important). 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.

3.3. *Example Operation*

To exemplify, this section describes the operation of the described architecture in a hypothetical scenario. The following assumptions are made: (i) the monitoring site has a list of proxy addresses; (ii) 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; and, (iii) 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 each 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. Evaluation of the Proposed Architecture

4.1. *Prototype Implementation and Deployment Issues*

The proposed architecture has been implemented as a prototype. 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 1. 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. The prototype's behaviour has been validated, for a number of settings [26, Section 6.2.2].

In order to assess the performance trends of the proposed architecture for various problem settings and grid sizes in realistic conditions, PlanetLab [18] has been used to deploy proxies in up to 100 hosts across North America, Europe, Australia and Asia. PlanetLab is a worldwide testbed for experimenting with Internet-scale services, with more than 800 nodes over about 400 sites as of mid-2007. 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.

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 synchronize its clock with the clock of the data manager (which has a thread specifically for this purpose) at the monitoring site. 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.

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 and the TCP ports at which they listen to are known, 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: (i) Launch of the data manager for processing the RDF-encoded differential updates that are downloaded by the crawler. (ii) Re-launch of the crawler, this time requesting from proxies exclusively differential updates. (iii) Launch of the importance evaluator. (iv) Initiation of resource event generation: all proxies are notified to start generating resource changes according to a pre-specified setting. (v) 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. (vi) Launch of the query generator, which emulates a specified query workload (submitted from the users). A query workload determines the type, number and arrival pattern of queries that should be submitted to the database front-end. Finally, it is noted that all experiments terminate after 60 minutes.

For the purpose of the experiments, proxies do not actually collect data from

deployed information services, as this would have had many practical complications. 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. For every cluster service of a grid site, the site profile includes information about properties of this cluster service, such as hardware and software configuration, load status, etc. The work in [13] has been used to generate the hardware configuration, whereas a number of basic assumptions have been made with respect to other properties. It is noted that, at this stage, the aim is to evaluate the behaviour of the architecture with respect to changes of these properties as opposed to the precise fine-grain modelling of these properties.

With respect to events that denote changes in the status of resources and their properties, a separate thread in every proxy, the EventGenerator, is used. Whenever a new event occurs, the EventGenerator notifies the main proxy thread to update its RDF representation of site profile. The average time between consecutive occurrences of every event type is predetermined.

4.2. *Performance Metrics*

As already stated in Section 2, the performance of a grid information service is measured primarily according to three performance metrics: network overhead, information freshness and query response time. In the context of our experiments these are measured as follows.

Network Overhead Network overhead is measured as the number of bytes that the crawler downloads during an experiment. Essentially, this is the sum of the number of bytes downloaded for each update of the monitoring site carried out by the crawler. We further distinguish the network overhead to the network overhead due to: (i) full updates that the crawler carries out when it first visits every site; and, (ii) differential updates that the crawler carries out after it has visited at least once a site and where only information that has changed since the last visit is collected. Clearly, the total network overhead is the sum of the two.

Information Freshness A value for a particular resource property stored at a monitoring site is considered fresh, at a given point in time, if it is synchronised with its real-world equivalent value at the grid site (that is, both values are the same). Thus, information freshness, at time t, of a collection of resources C that has k resource property values p_j is given by:

$$F_{C,t} = \frac{1}{k} \sum_{j=0}^{k-1} F_{p_j,t} \tag{1}$$

where $F(p_j, t)$ is 1, if the property value p_j is up to date at time t, and 0 otherwise. This implies that, at a given time, the information freshness of a collection of properties indicates the percentage of resource properties that are up to date at

that point in time. To measure information freshness throughout an experiment, it is useful to define time-average information freshness or simply average freshness as the average of the information freshness at j consecutive equally-distanced time points during a time interval T (e.g., the duration of the experiment). Thus:

$$\bar{F}_{C,T} = \frac{1}{j} \sum_{t=0}^{j-1} F_{C,t}. \tag{2}$$

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, using the formulas above. The results reported next refer to average information freshness. It is noted that only the 5 properties that do change during experiments are taken into account in the calculation of freshness. This makes it easier to study the behaviour of this metric; including all properties would produce significantly higher values for freshness and smaller deviations between different measurements.

Query Response Time Query Response Time 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 user who set the query. The database front-end keeps a log-file with the response time of each query processed during an experiment. Same as before, it is helpful to use the average query response time to denote the average value of the set of Query Response Time values in the database front-end's log-file at the end of a given experiment.

4.3. *Exploring the Evaluation Space / 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. It is noted that the number of possible problem settings is potentially too large. The evaluation considers 20 different settings: a baseline setting, and 19 settings that differ from the baseline only in terms of one parameter. The purpose of this is to investigate performance trends for each parameter separately.

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.

The baseline setting has the following parameter values: (i) *grid size:* 2000 grid

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	between 1 and 15, determined proportionally to site importance
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%)

Fig. 3. Problem settings considered in the evaluation.

sites and 5 cluster services per site; (ii) *event generation mode:* resource changes for 5 event types (each one corresponding to a different resource property) 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; (iii) *update frequency:* all sites are visited by the crawler every 5 minutes; (iv) *number of queries and query arrivals:* 4000 queries uniformly distributed over the experiment duration; (v) *query complexity and selectivity:* high complexity query with high selectivity (described below).

The 19 variations of the baseline setting are shown in Figure 3. In the "setting id" column of the table, every parameter is denoted with a different letter; every value that is considered for a parameter is indicated with a distinctive number. The baseline value for a given parameter is indicated with the number zero. For example, the letter 's' indicates the grid size parameter, and 's0' indicates the baseline setting for grid size.

For grid size, we considered 500, 2000 and 5000 sites. Each of these values was also considered with 1, 5 and 10 cluster services per site (s1-s9 Figure 3). 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 3.2). 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 4. 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.

4.4. *Results*

Network overhead The plots in Figure 5 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 network overhead, and (ii) only the overhead 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.

As expected, the middle plot in Figure 5 shows that the network overhead increases linearly with the number of monitored cluster services. Conversely, the network overhead decreases linearly as the frequency of resource changes increases. The same is also true when it comes to the frequency of updates (bottom plot in Figure 5). It is interesting to note here that an increase of a certain order in the frequency of updates does not guarantee an increase of exactly the same order in the

```
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
```

Fig. 4. Definition of the queries used in the evaluation.

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. For example, the u1 setting, where updates are performed 5 times more frequently compared to u0, has approximately twice the differential network overhead of u0. Similarly, u3, in which updates are performed 3 times less frequently compared to u0, has less than half the differential network overhead of u0. Finally, with respect to making use of 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 show how information freshness is affected by frequency of resource changes, grid size, and update frequency. In Figure 6 (top plot) it can be seen that freshness increases proportionally

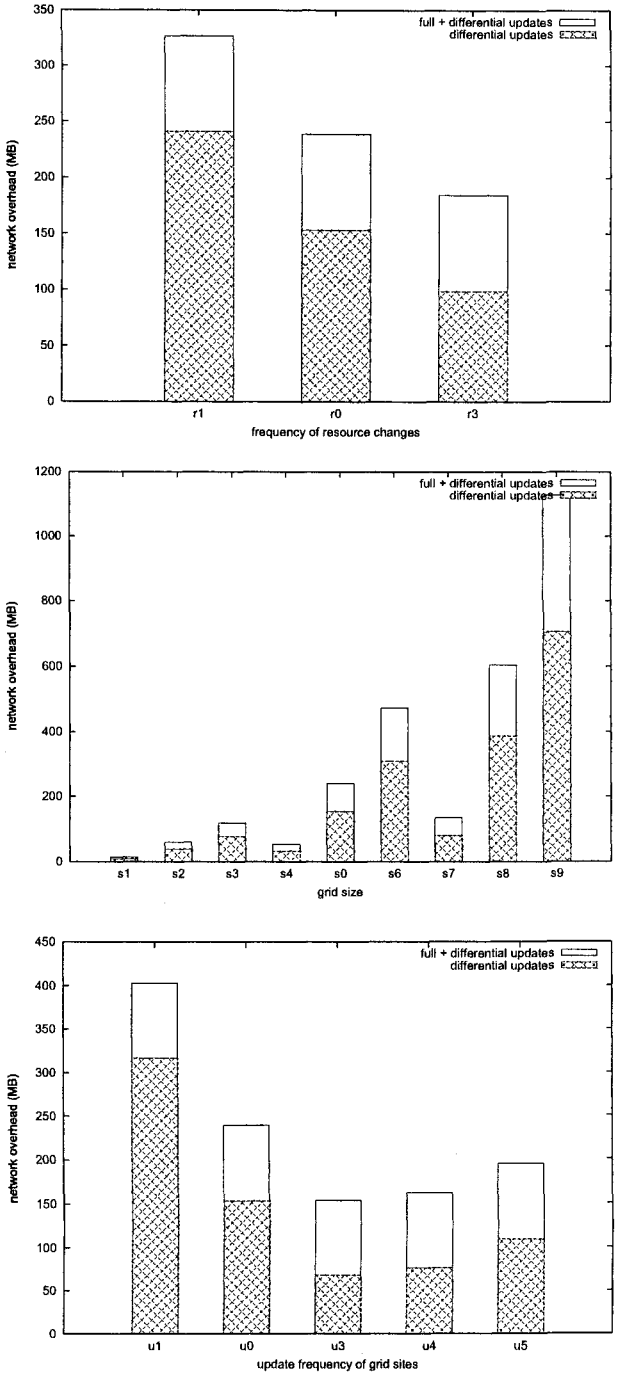

Fig. 5. Network overhead for several variations of the baseline setting.

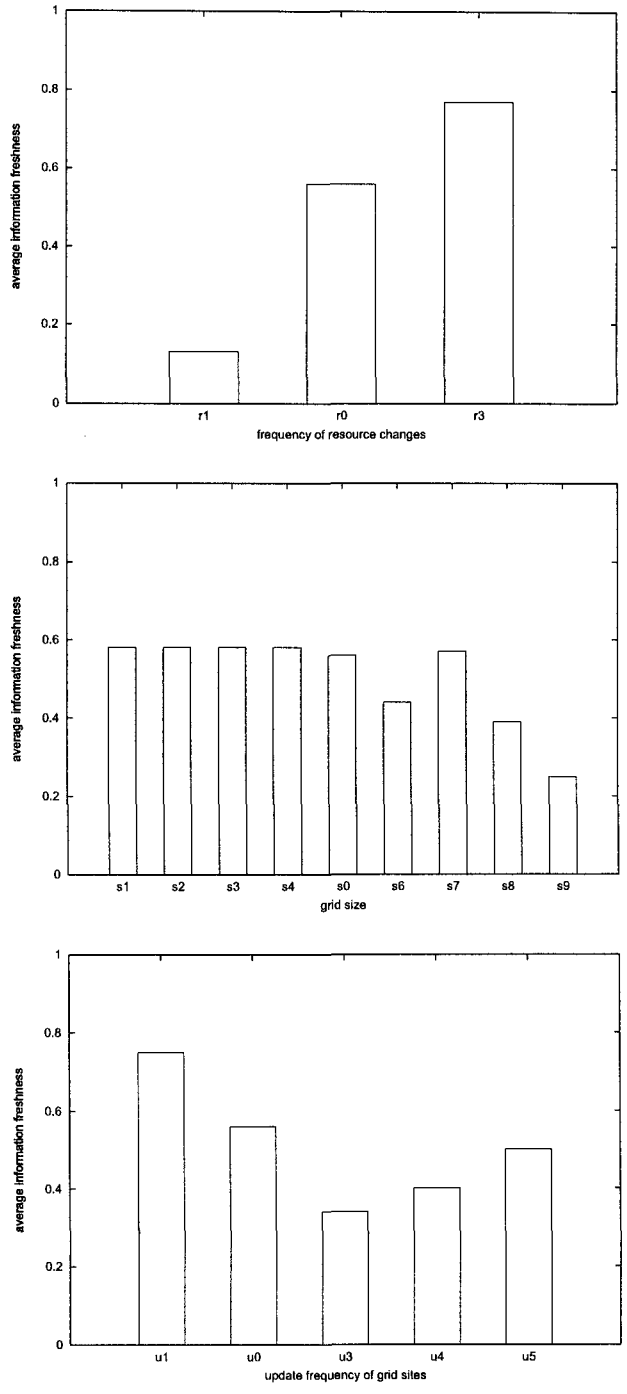

Fig. 6. Average information freshness for several variations of the baseline setting.

to decreases of the frequency of resource changes. The middle plot indicates that freshness remains constant for different grid sizes, except when there is a large number of monitored service clusters, in which case it drops (settings s6, s8, s9). It has been found that this is implementation-specific and it is caused by high load at the main store. The bottom plot of Figure 6 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 deserves further analysis as, in these settings, freshness varies significantly across sites based on site importance.

Thus, Figure 7 shows the average information freshness of the resources of a site compared to the importance of that site, in three different experiment settings: baseline (u0), topn (u4), and inversely proportional (u5). In the baseline experiment (top plot in Figure 7), all sites are updated every 15 minutes. Thus, average freshness is approximately the same for all sites. The middle plot shows that sites are distinguished in two groups; the smaller group has higher average information freshness. In the u5 experiment (bottom plot in Figure 7) the update frequency of a site is inversely proportional to its importance, and is mapped in the range [1, 15] minutes. It is interesting to observe that the average information freshness in this case shows some power-law connection with the site's importance: very few sites (with high importance) exhibit high information freshness, whereas many sites (with low importance) exhibit lower information freshness.

Average query response time Figures 8 and 9 show how QRT is affected by variations in all of the considered problem settings. It can be seen in the top plot of Figure 8 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 8 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 9). Finally, the top plot in Figure 9 shows that the relation of grid size (and as a result database size) and QRT tends to be proportional.

Discussion This section evaluated a prototype implementation of the proposed architecture using PlanetLab. Several problem settings were considered, including configurations of up to 5000 sites and 50000 cluster services. The results of the

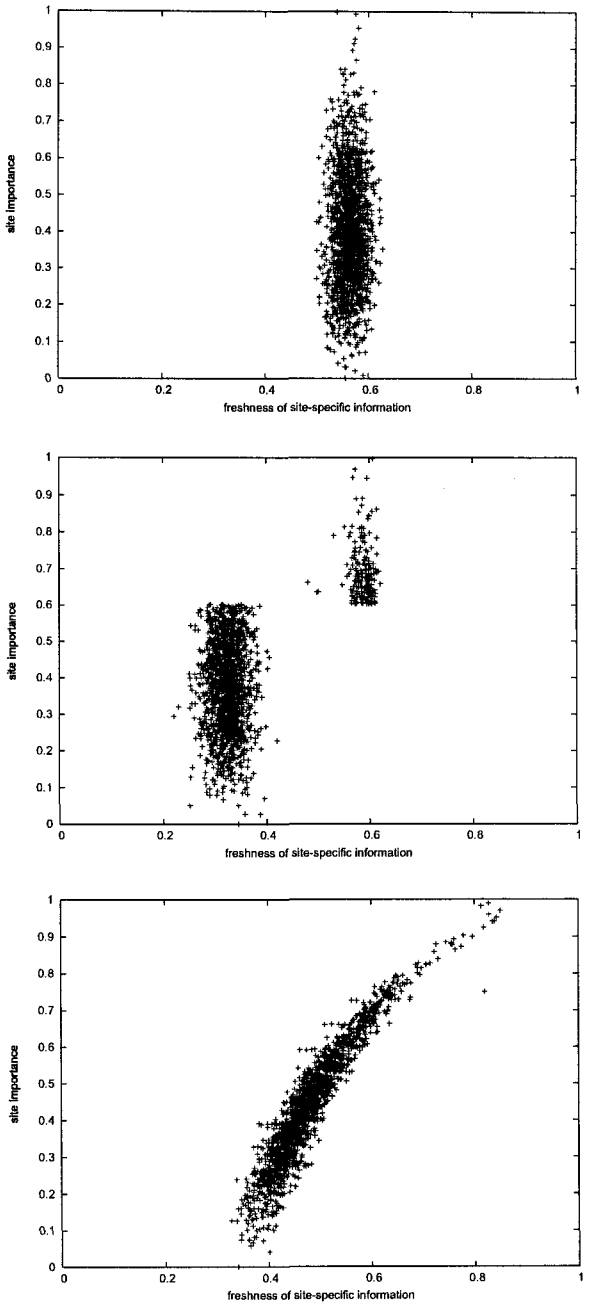

Fig. 7. Average information freshness per site versus site importance, in the baseline (top), u4 (middle), and u5 (bottom) settings.

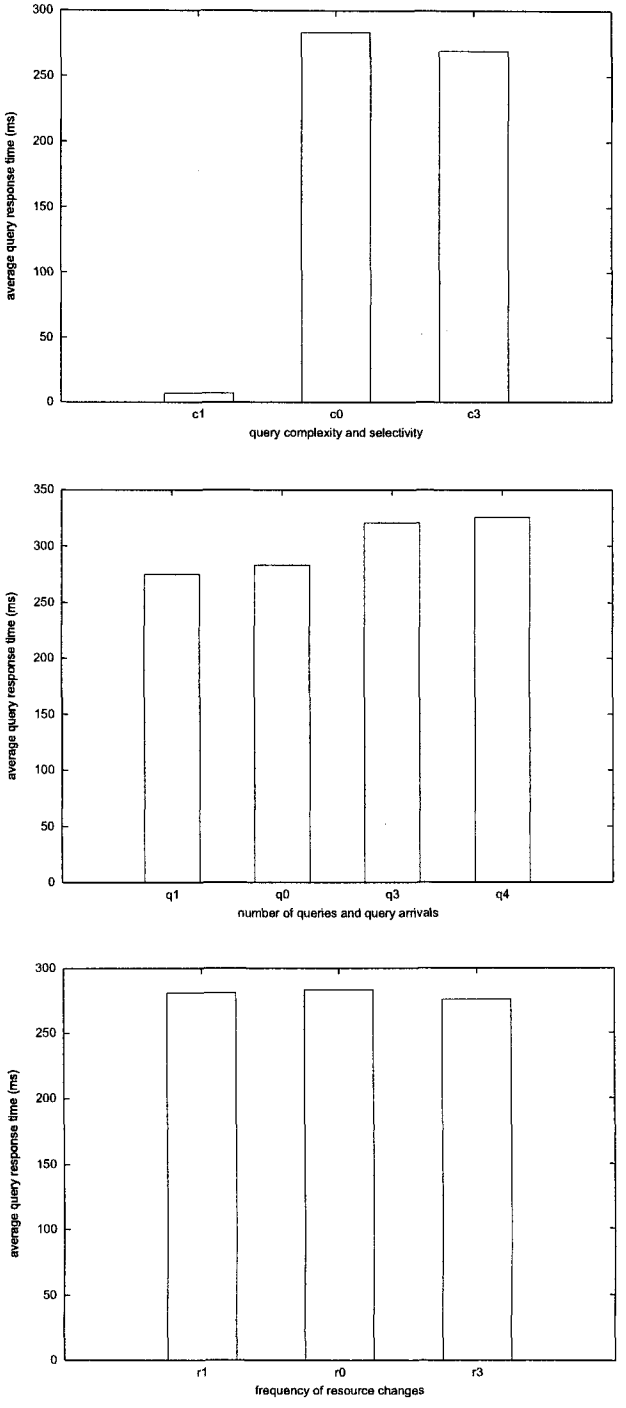

Fig. 8. Average QRT for variations of the baseline setting in terms of query complexity and selectivity (top), number of queries and query arrivals (middle), and frequency of resource changes (bottom).

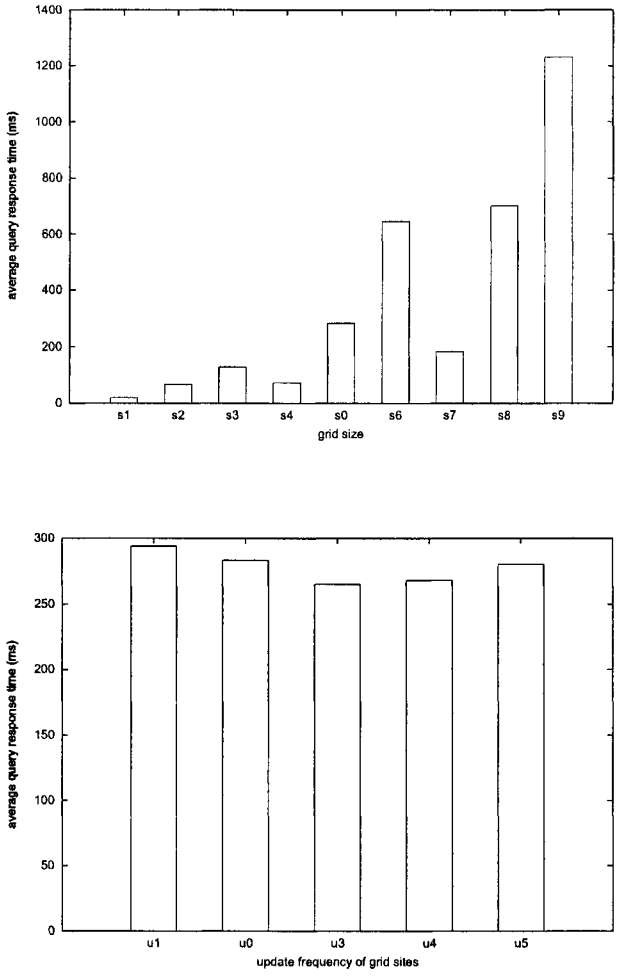

Fig. 9. Average QRT for variations of the baseline setting in terms of grid size (top) and update frequency of grid sites (bottom).

PlanetLab experiments have been promising. Despite the highly volatile nature of PlanetLab (whose hosts are shared at the same time by many users), the prototype behaves as expected. Thus: (i) the network overhead increases proportionally with the frequency of resource changes, the inverse of the frequency of updates as well as with the number of sites and services; (ii) the average information freshness depends on the frequency of resource changes and updates; and, (iii) the query response time is primarily affected by query complexity and selectivity. In addition, implementation decisions appear to make an impact only on information freshness and large sizes (this is related to the ability of the data manager to scale with an

increasing number of incoming updates; however, this could be improved by using multiple databases). Finally, the results obtained when updates are adjusted based on resource importance demonstrate the potential of the architecture to provide high information freshness for those resources deemed to be important.

5. Conclusion

This paper has redefined the problem of large-scale grid information services by making it possible to adjust the information freshness versus network overhead trade-off on a site-level basis. The proposed architecture, based on importance-aware prefetching, collects information in advance of query arrivals. Thus, it achieves low query response times, while at the same time it has the capability to: (i) maintain high information freshness for sites that are considered to be important; and (ii) control the imposed network overhead. Evaluation results of a prototype implementation on PlanetLab demonstrate the performance trade-offs between network overhead, frequency of resource changes, and frequency of updates for several settings.

Further work could consider additional evaluation settings; as already mentioned the possible evaluation space is potentially too large. Such work could provide more informative answers to questions such as 'what is the required frequency of updates to attain a certain level of information freshness', or 'how to choose update frequencies so that the network overhead does not exceed a certain value'. Porting the implementation onto existing grid platforms and comparing with existing implementations of grid information services would also provide more insight. Early experiences based on performance models [26] suggest that the approach proposed in this paper has the potential to outperform existing hierarchical, just-in-time, implementations of grid information services, both in terms of the network overhead and query response time; however, this also needs further investigations. Finally, an important direction that has not been addressed in this paper relates to different definitions of resource importance that can be adapted on-the-fly as it may be required.

Acknowledgements

This research work is carried out partially under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] M. Bhide, P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive Push-Pull: Disseminating Dynamic Web Data. *IEEE Trans. Comput.*, 51(6):652–668, 2002.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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, *Engineering the Grid: Status and Perspectives*. American Scientific Publishers, January 2006.

[12] P. Dinda and D. Lu. Fast Compositional Queries in a Relational Grid Information Service. *Journal of Grid Computing*, 3:131–150, June 2005.

[13] 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.

[14] 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.

[15] H.N.L.C. Keung, J.R.D. Dyson, S.A. Jarvis, and G.R. Nudd. Performance Evaluation of a Grid Resource Monitoring and Discovery Service. *IEE Proceedings-Software*, 150(4):243–251, August 2003.

[16] H.N.L.C. Keung, J.R.D. Dyson, S.A. Jarvis, and G.R. Nudd. Predicting the Performance of Globus Monitoring and Discovery Service (MDS-2) Queries. In *Fourth International Workshop on Grid Computing*, pages 176–183. IEEE, 2003.

[17] M. L. Massie, B. N. Chun, and D. E. Culler. Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7), July 2004.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] S. Zanikolas. *Importance-Aware Monitoring for Large-Scale Grid Information Services*. PhD thesis, School of Computer Science, University of Manchester, 2007.

[27] 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.

[28] 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.

[29] X. Zhang, J. L. Freschl, and J. M. Schopf. Scalability Analysis of Three Monitoring and Information Systems: MDS2, R-GMA, and Hawkeye. *Journal of Parallel and Distributed Computing*, 67:883–902, August 2007.