Load-Aware Dynamic Replication Management in a Data Grid

Laura Cristiana Voicu and Heiko Schuldt

Department of Computer Science, University of Basel, Switzerland
{laura.voicu,heiko.schuldt}@unibas.ch

Abstract. Data Grids are increasingly popular in novel, demanding and data-intensive eScience applications. In these applications, vast amounts of data, generated by specialized instruments, need to be collaboratively accessed, processed and analyzed by a large number of users spread across several organizations. The nearly unlimited storage capabilities of Data Grids allow these data to be replicated at different sites in order to guarantee a high degree of availability. For updateable data objects, several replicas per object need to be maintained in an eager way. In addition, read-only copies serve users' needs of data with different levels of freshness. The number of updateable replicas has to be dynamically adapted to optimize the trade-off between synchronization overhead and the gain which can be achieved by balancing the load of update transactions. Due to the particular characteristics of the Grid, especially due to the absence of a global coordinator, replication management needs to be provided in a completely distributed way. This includes the synchronization of concurrent updates as well as the dynamic deployment and undeployment of replicas based on actual access characteristics which might change over time. In this paper we present the Re:GRIDiT approach to dynamic replica deployment and undeployment in the Grid. Based on a combination of local load statistics, proximity and data access patterns, Re:GRIDiT dynamically adds new replicas or removes existing ones without impacting global correctness. In addition, we provide a detailed evaluation of the overall performance of the dynamic Re:GRIDiT protocol which shows increased throughput with respect to the replication management protocol with a static number of replicas.

1 Introduction

Novel data-intensive applications are increasingly popular in eScience. In these applications, vast amounts of data are generated by specialized instruments and need to be collaboratively accessed, processed and analyzed by a large number of scientists around the world. Many good examples can be listed such as high energy physics, meteorology, computational genomics, or earth observation. Due to their enormous volumes, such data can no longer be stored in centralized systems but need to be distributed worldwide across data centers and dedicated storage servers. Motivated by the success of computational Grids in which distributed resources (CPU cycles) are globally shared in order to solve complex problems in a distributed way, a second generation of Grids, namely *Data Grids*, has emerged as a solution for distributed data management in data-intensive applications. The size of data required by these applications may be

R. Meersman, T. Dillon, P. Herrero (Eds.): OTM 2009, Part I, LNCS 5870, pp. 201–218, 2009.

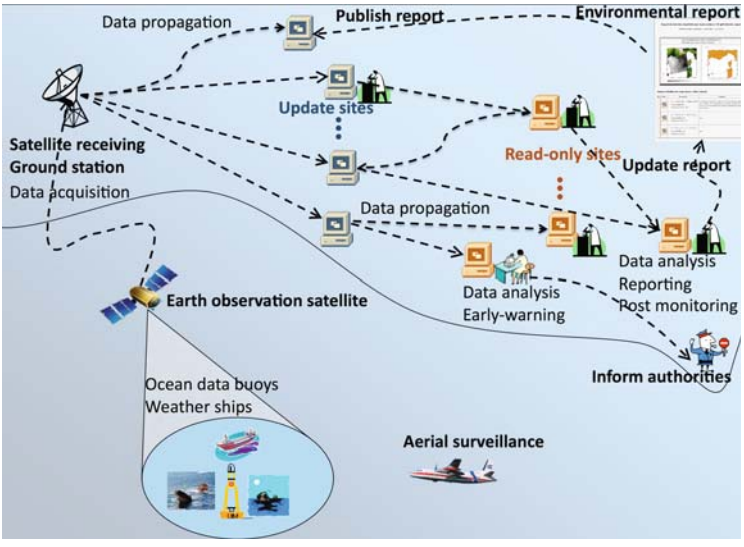

Fig. 1. Example Application Scenario: Data Grids for Earth Observation

upto petabytes. High energy physics data instruments like the large hadron collider at CERN, will produce vast amounts of data during their lifetime: "[...] will be the most data-intensive physics instrument on the planet, producing more than 1500 megabytes of data every second for over a decade" [7]. Similarly, in the earth observation domain, data archives around the world will grow to around 14.000 Tbytes by the year 2014 [14].

Consider the following earth observation scenario (depicted in Figure 1), where data are collected at one or more stations and maintained at multiple sites [26]. In order to improve availability, scalability and avoid single points of failure in the system as well as to best meet the users' requirements, copies of data are maintained at multiple geographically distributed sites. Assume that scientists are closely monitoring oil spills into the sea in a collaborative effort. The potential damage to the area at stake requires the readiness to detect, monitor and clean-up any large spill in a rapid manner. The occurrence of such an incident results in the acquisition and analysis of thousands of images summing up to Tbytes of data and generating multiple concurrent accesses to data. For this type of applications the most up-to-date data, provided in a timely manner is essential, even in the presence of concurrent updates. An efficient data replication protocol has to guarantee the success of such applications, by ensuring, to mention only a few of the advantages: availability of data, efficient access to data, freshness of data, consistency of replicas and reliability in accessing the data.

Replication techniques are able to reduce response times by moving and distributing content closer to the end user, speeding content searches, and reducing communication overhead. An efficient replication management protocol has to consider changes in the access patterns, i.e., should dynamically provide more replicas for data objects which are frequently accessed. However, increasing the number of updateable replicas per data object in an unlimited way may have significant drawbacks on the overall system performance. Therefore, the number of updateable replicas for data objects which are

no longer of importance to a large group of users should be dynamically reduced, in order to reduce the overhead of replica maintenance. A number of existing approaches to replication management in parallel and distributed systems address data placement in regular network topologies such as hypercubes, rings or trees. These networks possess many attractive properties that enable the design of simple and robust placement algorithms. These algorithms, however, cannot be directly applied to Data Grid systems due to the lack of regular network structures and the data access patterns in Data Grid systems which are very dynamic and thus hard to predict.

In this paper, we propose the Re:GRIDiT approach to dynamic replica management in Data Grid systems, taking into account several important aspects described below. First, replicas are placed according to an algorithm that ensures load balancing on replica sites and minimizes response times. Nevertheless, since maintaining multiple updateable copies of data is expensive, the number of updateable replicas needs to be bounded. Clearly, optimizing access cost of data requests and reducing the cost of replication are two conflicting goals and finding a good balance between them is a challenging task. We propose efficient algorithms for selecting optimal locations for placing the replicas so that the load among these replicas is balanced. Given the data usage from each user site and the maximum load of each replica, our algorithm efficiently minimizes the number of replicas required, reducing the number of unnecessary replicas. This approach to dynamic replica management is embedded into the Re:GRIDiT replica update protocol which guarantees consistent interactions in a Data Grid and takes into account concurrent updates in the absence of any global component [25,26].

The remainder of the paper is organized as follows. Section 2 discusses related work. In Section 3, we introduce our system model. Section 4 gives details of the proposed protocol. Section 5 describes the implementation of our protocol and provides a detailed performance evaluation. Finally, Section 6 concludes.

2 Related Work

In contrast to its vital importance, dynamic replication management in Data Grids which takes into account its particular requirements of the Grid has not been the focus of previous work. In general, we can distinguish three related fields: i.) data replication in the Grid which mainly concentrates on read-only data and/or manual data placement; ii.) replication management in database clusters and peer-to-peer networks; iii.) dynamic replication of services in Grid environments.

In current Data Grids, replication is mostly managed manually and based on single files as the replication granularity [17]. In addition, these files are mainly considered as read-only [10,11]. There are also more comprehensive solutions as proposed in [20]. This solution, however, may suffer from consistency problems in case of multiple catalogs and while accessing more than one replica. Moreover, to the best of our knowledge, there is no replication solution for Data Grids which address freshness and versioning issues so far while also taking into account several updateable replicas.

In the context of replication management in database clusters, there are several well-established protocols in the database literature which deal with *eager* and *lazy* [13] replication management. Conventional eager replication protocols have significant drawbacks with respect to performance and scalability [13,15] due to the high communication

overhead among the replicas and the high probability of deadlocks. In order to overcome these drawbacks, group communication primitives have been proposed in [16]. Lazy replication management, on the other hand, maintains the replicas by using decoupled propagation transactions which are invoked after the "original" transaction has committed. Older works on lazy replication focus only on performance and correctness [5,8]. The approach presented in [4] extends this focus by imposing much weaker requirement on data placement and recent works consider freshness issues as well [19,1]. The drawback of these approaches is that they assume a central coordinator which can become a bottleneck or a single point of failure. Dynamic replication in peer-to-peer networks has also known extensive development over the years [2,9,12,18]. Sophisticated solutions, which eliminate the need of a central coordinator, have been proposed, but they either do not take into account data freshness [2], or multiple data types ([18] for example, only considers images as data objects) or neglect consistency issues [12].

An early attempt to evaluate scalability properties of adaptive algorithms for replica placement services in dynamic, distributed systems was reported in [3]. Weissman and Lee [27] proposed a replica management system which dynamically allocates replicated resources on the basis of user demand, where resource requests by users can be transparently rerouted if individual replicas fail. More recently, other replication schemes have been proposed. Through experiments, most have demonstrated limited scalability and ability to operate under conditions of resource heterogeneity and dynamism. Valcarenghi [21] presented a replication approach in which replicas are located in proximity to each other to form service islands in a Grid network. These approaches have been developed at the service level and thus do not take into account data replication constraints or freshness and versioning issues.

3 The Re:GRIDiT System Model

Despite recent advances, Data Grid technologies often propose only very generic services for deploying large scale applications such as users authorization and authentication, fast and reliable data transfer, and transparent access to computing resources. In this context, we are facing important challenges such as the ones coming from the earth observation application presented in Section 1. We built therefore further functionality on top of the underlying Grid middleware, while taking into account particular requirements coming from both the Grid infrastructure and the Grid applications (dynamically distributed management of replicated data, different levels of freshness, the absence of a central coordinator). In this paper we present the Re:GRIDiT (Replication Management in Data GRid Infrastructures using Distributed Transactions) protocol that dynamically synchronizes updates to several replicas in the Grid in a distributed way. Re:GRIDiT was developed as a response to the need of a protocol that meets the challenges of replication management in the Grid, and that re-defines the Grid and re-discovers it (in other words, that "re:grids it"), bringing it to a level where it can satisfy the needs of a large variety of users from different communities.

Figure 2 sketches the Re:GRIDiT architecture. The bottom layer is built by the hardware: computing nodes, storage nodes and fast network connections. At the middleware level, Grid services provide homogeneous and transparent access to the underlying heterogeneous components while Re:GRIDiT services (which build on top of any Grid

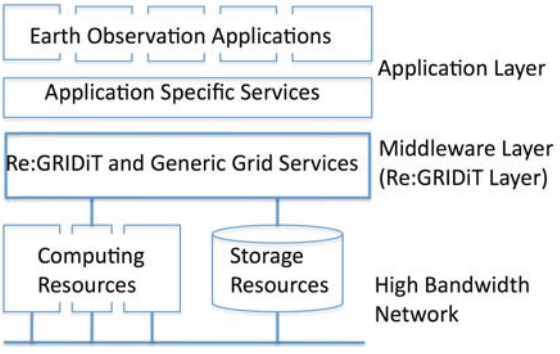

Fig. 2. A Layered Architecture, from Resources to Applications

middleware) provide dynamic and distributed replication of Grid-enabled applications with data sharing and distributed computation capabilities in a way that is transparent to the applications and the users. The application specific layer provides high level and domain specific services taking into account the data semantics, access to heterogeneous database formats, high level services, support for parallel and interactive applications, etc. The components in each layer build on new capabilities and behaviors provided by the lower layer. This model demonstrates flexibility and shows how Grid architectures can be extended and evolved upon.

At the middleware layer, we assume Re:GRIDiT as being part of the Grid middleware and present on each site. The sites hold data objects, replicated in the network, and operations through which the data objects can be accessed. A data object can be represented at the base level as a file stored on the local file system, or inside a database on the local database management system (similar to the model presented in [6]). Each data object is uniquely identified by an object identifier OID.

As illustrated in Figure 3 each site s_j offers a set of operations on data objects $Op^{s_j} = \{op_1, op_2, \ldots, op_n\}$. The operations can be invoked within transactions using the interface of the site. We consider the following update operations (shortly referred to as updates): insert object (denoted $insert(d, t, OID)$ for an insertion of a data object d into a storage type t (file or database), with a specified OID); delete object (denoted $delete(OID)$ for the deletion of a data object with a specified OID); replace object (denoted $replace(d, OID)$ for the replacement of an object with a specified OID). We consider the following read operations: read object (denoted $read(OID)$ for the reading of an object with a specified OID; read versioned object (denoted $read_v(vnr, OID)$ for the reading of a version of an object with a specified OID).

In addition to these operations, which we call *direct operations*, and which result from the translation of user operations, we assume two additional operations which are not available at the user level. These operations are used by the system in order to support replication (*indirect operations*). We consider the following indirect operations: copy object (denoted $copy(OID, vnr, dest)$ for the copying of a version of a data object to a specified destination site); remove object (denoted $remove(OID, vnr, dest)$ for the removal of a version of a data object from a specified destination site).

In our work the notion of conflict is defined based on the commutativity behavior of operation invocations (semantically rich operations [22]). Dependencies between transactions in a transaction's schedule occur when there is at least a pair of operation invocations that is in conflict. We model our system such that each site holds a copy of the conflict matrices, by means of which they can automatically detect conflicts. We assume conflicts to be rather infrequent, but they need to be handled properly in order to guarantee globally serializable executions.

In Re:GRIDiT one data object can reside only on one site, but could be replicated on several sites. We distinguish between two types of replica sites: *update* and *read-only* sites. Read-only sites allow read access to data only. Updates occur on the update sites and are propagated to other update sites and finally to the read-only sites. The nature of a site (i.e., update or read-only) is determined by the operations it provides [26]. We define an *n-to-m* relationship between the two types of sites; an update site can have any number of read-only children (to which updates are propagated). Read-only sites in turn can be shared with other update sites.

Re:GRIDiT assumes the presence of local logs on each site (see Figure 3), where operation invocations are recorded. These logs are used for conflict detection and recovery purposes. Each site uses a set of tools to obtain a (typically partial and sometimes even out-dated) information regarding the state of the system and take replication decisions. We introduce the following components to facilitate the scheduling of read-only transactions in the Grid and the replica management decision. These components are not centrally materialized in the system, and contain global information that is distributed and replicated to all the update sites in the system:

- *replica repository*: used to determine the currently available update replicas in the network. Moreover, each update replica is aware of its own read-only replicas, where it propagates update changes, in order to maintain a certain level of freshness/staleness in the network.
- *freshness repository*: used to collect freshness levels of the data objects periodically or whenever significant changes occur. In this paper, the term freshness level is used to emphasize the divergence of a replica from the up-to-date copy. Consequently update sites will always have the highest freshness level, while the freshness level of the read-only sites will measure the staleness of their data. Since update sites propagate changes to read-only sites, they are aware of the freshness levels of the read-only sites to which they propagate changes.
- *load repository*: used to determine an approximate load information regarding sites. This information can then be used to balance the load while routing read-only transactions to the sites or for the replica selection algorithm. Update sites periodically receive information regarding the load levels of other update sites and their read-ony children. Read-only sites are only aware of their own load levels.[1]

[1] In order to improve efficiency and not to increase the message overhead this information can be exchanged together with replica synchronization request. This information needs to be exchanged more frequently while there is a replica synchronization process in place, while the exchange is not needed during a site's idle time, when the load is unlikely to vary.

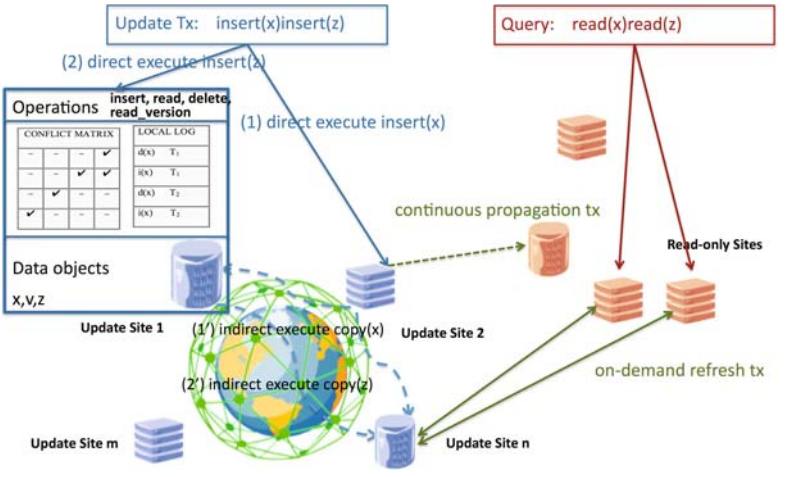

Fig. 3. System Model Architecture at the Middleware Level

Besides these local metadata each site maintains a set of global, pre-defined set of variables: δ_T, collection time interval (defined as the time between individual state information collection) and Δ_{load}. These values strongly depend on the nature of the supported application, but the protocol may allow these values to be self-defined on a per site basis, depending on the physical characteristics of the site.

With respect to transactions submitted by clients, i.e., client transactions, we distinguish between *read-only transactions* and *update transactions* (according to [1]). A read-only transaction only consists of read operations. An update transaction comprises at least one update operation next to arbitrarily many read operations. Decoupled *refresh transactions* propagate updates through the system, on-demand, in order to bring the read-only replicas to the freshness level specified by the read-only transaction. *Propagation transactions* are performed during the idle time of a site in order to propagate the changes present in the local propagation queues to the read-only replicas. Therefore, propagation transactions are continuously scheduled as long as there is no running read or refresh transaction. Re:GRIDiT exploits the sites' idle time by continuously scheduling propagation transactions as update transactions at the update sites commit. Copies at the read-only sites are kept as up-to-date as possible such that the work of refresh transactions (whenever needed) is reduced and the performance of the overall system is increased. Through our protocol a globally serializable schedule is produced (although no central scheduler exists and thus no schedule is materialized in the system) [24]. Moreover, the update transactions' serialization order is their commit order.

Note that a transaction may not always succeed due to several failure reasons. To satisfy the demand for an atomic execution, the transaction must compensate the effects of all the operations invoked prior to the failure [22]. This compensation is performed by invoking semantically inverse operations in reverse order.

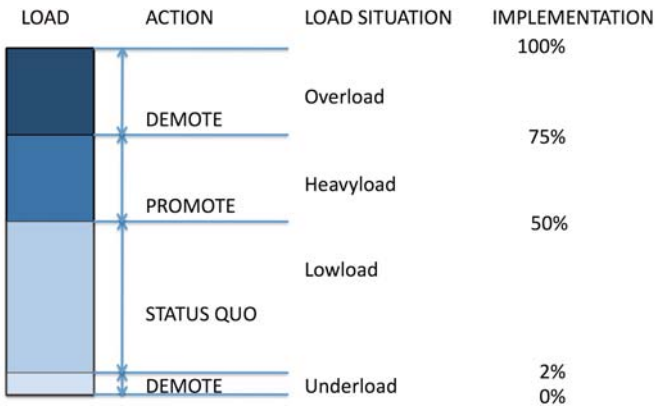

Fig. 4. Possible Load Situations

3.1 Load Metrics

In practical scenarios, the load metrics may reflect multiple components, such as CPU load, storage utilization, disk usage or bandwidth consumption. We abstract from the notion of load by defining a load function $load \in [0, 1]$, where the maximum, in the case of CPU load, for instance, corresponds to 100 % load.

We assume that an individual site can measure its total load. This can be done by keeping track of resource consumption (CPU time, IO operations, access count rate etc.) In order to ensure an accurate measurement, the load collection is averaged over a certain time window and consequently the decisions are made based on load trends rather than punctual values.

We define the mean load as $\overline{load(s)} = \frac{\sum_{i=0}^{n} load(s_i)}{n}$ where n represents the total number of load calculations within the given time window and classify the following load situations (for a site s):

- An *overload* situation can occur if $\overline{load(s)} \in (\alpha_{overload}, 1)$, where $\alpha_{overload} \in [0; 1]$.
- A *heavyload* situation can occur if $\overline{load(s)} \in (\alpha_{heavyload}, \alpha_{overload})$, where $\alpha_{overload} \in [0; 1]$ and $\alpha_{heavyload} \in [0; 1]$.
- A *lowload* situation can occur if $\overline{load(s)} \in (\alpha_{underload}, \alpha_{heavyload})$, where $\alpha_{heavyload} \in [0; 1]$ and $\alpha_{underload} \in [0; 1]$.
- An *underload* situation can occur if $\overline{load(s)} \in (0, \alpha_{underload})$, where $\alpha_{underload} \in [0; 1]$.

The parameters $\alpha_{underload}$, $\alpha_{heavyload}$ and $\alpha_{overload}$ are application dependent and obey the following relation: $0 \leq \alpha_{underload} \leq \alpha_{heavyload} \leq \alpha_{overload} \leq 1$. The possible load levels (together with an example of their implementation in a practical scenario) are presented in Figure 4. We extend the above classification to a system-wide level (i.e., a *system overload* can occur if $\frac{\sum_{j=0}^{N} \overline{load(s_j)}}{N} \geq \alpha_{overload}$, where

$\overline{load(s)}$ is the average load on a site s and N represents the total number of sites in the system) and define two additional situations that can be observed at a system level:

- A *majority overload* situation can occur if a majority of sites (of at least $\frac{N}{2}+1$) is in an overload situation. The remaining sites must be at least in a heavyload situation.
- A *minority overload* situation can occur if a minority of sites (of at most $\frac{N}{2}-1$) is in an overload situation. The remaining sites must be at least in a heavyload situation.

Note that the definition of the load levels in a practical scenario is of major importance since crossing the boundaries of a certain level will trigger a certain action in the dynamic replication protocol.

3.2 Best Replica

The efficiency of the replication system depends on the criteria used for the selection of replica sites. In our protocol, we choose the notion of *host proximity* as one of the criteria for replica selection. The proper definition of this metric impacts both the response time perceived by the user and the overhead involved in measuring the distance. In order to define the "closest" replica, the metrics may include response time, latency, ping round-trip time, network bandwidth, number of hops or geographic proximity. Since most of the above metrics are dynamic, replica selection algorithms such as [23] typically rely on estimating the current value of the metric using samples collected in the past. We abstract from the above notions by defining a *closeness function*, in which the notion of closeness can reflect any of the metrics above, and is defined as $close(s, c) \in [0, 1]$, where 0 is the absolute minimum value for this metric (for example, in the case of geographic proximity, 0 would correspond to the site s that is geographically farthest away from the client c and 1 to the site s that is geographically closest to the client c).

The second criterion for the replica selection is based on the notion of "freshness". We use in our approach a freshness function, $fresh(d, s) \in [0, 1]$, (as defined in [19]) that reflects how much a data object d on a site s has deviated from the up-to-date version. The most recent data has a freshness of one, while a freshness of zero represents infinitely outdated data (data object is not present on that site). For read-only sites, the freshest replica has the most up-to-date data (ideally as close to 1 as possible).

The third criterion is related to the concept of load (see Subsection 3.1) and defines the "least loaded" site as the one with the smallest value of the load within a given set.

The following situation may require a replica selection: a particular system load situation may require the acquisition of a new update replica from the read-only replicas available. For each replica s the replica selection algorithm keeps track of its load, $load(s)$ and the freshness of each data object d, $fresh(d, s)$. For read-only sites, the freshness is smaller or equal to one and represents a measure of the staleness of the data on a site. The algorithm begins by identifying replicas with the smallest load level, replicas with the highest freshness level and replicas that are the closest to the requester and chooses the best replica among them.

Definition 1. ***(Best replica function)*** *Let S be the set of all sites in the system and D the set of all data objects. Then, for a copy of a data object d, required by a given client site c, the function best(c,d) is defined as follows: $best : S \times D \rightarrow S$, such that $best(c,d) = s^* \in S$, where s^* corresponds to the greatest value of the sum: $\alpha close(s^*,c) + \beta fresh(d,s^*) + \gamma(1 - load(s^*))$, for given α, β, γ with $\alpha + \beta + \gamma = 1$, $\forall s^* \in S$.* □

Choosing replicas in the round-robin manner would neglect the proximity factor. On the other hand, always choosing the closest replicas could result in poor load distribution. Our protocol keeps track of load bounds on replicas, freshness levels and proximity factors, which enables the replica management algorithm to make autonomous selection decisions. We allow the decision to take into account the freshness level of a replica. The usage of freshness in this protocol is a trade off of consistency for performance. Furthermore, by using a weighted sum for the notion of best replica, we allow higher level applications to give preference to a certain parameter over the others.

4 The Re:GRIDiT Protocol

4.1 Replica Promote and Demote

In the following we define the conditions and consequences of dynamic replica acquisition or release. When a new updateable replica is created, it is updated with the content of existing update replicas (depending on the data objects that exist at that site). In order to ensure consistency the creation of a new update replica takes place in two phases: (i) *Phase 1*: sites are informed that a new replica will join the network, they finish currently executing transactions and start queueing any direct operations belonging to subsequent transactions; (ii) *Phase 2*: the replica has joined and is up-to-date, the sites start executing the queued direct operations taking into account the new replica when executing indirect operations.

However, a new replica promote may not always be beneficial. The decision whether to acquire or release a replica is made based on a combination of local (accurate) and global (partially accurate) information. We identify the following cases, depending on the local load of a site (see Figure 4):

- local *underload*; An *underload* situation intuitively implies that there are no updates and no queries at this site. In this case the site autonomously decides to self demote.
- local *lowload*; A *lowload* situation is considered to be a normal load situation. The site continues the normal execution.
- local *heavyload*; A *heavyload* situation is dealt with by the acquisition of additional replica(s). A site in *heavyload* will inform the other update sites of the intention to promote a new replica. If there is no other promote taking place at the same time and all the sites have acknowledged the promote, the site will proceed to choose the best replica, as described in Subsection 3.2, and promote it to update site.
- local *overload*; An *overload* situation is a special case in which the global state of the system needs to be taken into account, since a local situation may be the result of several influencing factors that do not necessarily reflect the state of the system. We distinguish two sub-cases:

Algorithm 1. Re:GRIDiT Site Protocol (Extension)

```
1: Main Execution Thread:
2: while true do
3:    Proceed normal Re:GRIDiT execution;
4:    wait for next message m;
5:    if m contains message: inform promote then
6:       finish direct operations of active transactions;
7:       update replica repository information;
8:       return ACK message;
9:       queue all incoming direct and indirect operations;
10:   else if m contains message: end inform promote then
11:      execute queued operations;
12:   else if m contains message: inform demote then
13:      update replica repository information;
14:   end if
15: end while
```

- In the case in which the system is in a majority overload situation the acquisition of a new replica is unlikely to improve the situation (since more replicas imply more synchronization). No action will be taken.
- In the case in which the system is in a minority overload situation, the overload can be assumed to be due to read operations (which are not replicated). The protocol will dictate the site to migrate the read request to other update sites in the system which are not in an overload situation.

Definition 2. ***(Replica promote).*** *The process by which a read-only site is transformed into an update site is called* ***replica promote*** *(replica acquisition).*
(Preconditions). *An update site s will begin the process of the acquisition of the best replica, from its own read-only children, which satisfies the criteria defined in Definition 1, if and only if the following statements hold: load(s) $= heavyload$ and there is no other promote initiated by a different site taking place at the same time and the already existing number of update replicas in the system has not reached a maximum.*
(Postconditions). *A read-only site r has been promoted to an update site and all the other update site are aware of the new replica. The information in the distributed replica repository is updated accordingly.* □

Definition 3. ***(Replica demote)*** *The process by which an update site is transformed into a read-only site is called* ***replica demote*** *(replica release).*
(Preconditions) *An update site s will begin the process of self release for a data object x, if and only if the following statements hold: load(s) $= underload$ or load(s) $= overload$ and no active transactions exist at s and the data are available elsewhere at a minimum number of replicas.*
(Postconditions) *Site s no longer exists as update site and all the other update sites are made aware of the disappearance of the replica site s. The information in the distributed replica repository is updated accordingly.* □

Algorithm 2. Re:GRIDiT Site Protocol (continued)

```
1: Background Thread:
2: while δ(T) do
3:    collect load
4:    if Promote Precondition then
5:       select best read-only replica according to Definition 1;
6:       inform update replicas of new promote;
7:       promote read-only replica;
8:       update replica repository information;
9:    else if Demote Precondition (in underload) then
10:      inform update replicas of self demote;
11:      update replica repository information;
12:   else if Demote Precondition (in overload) then
13:      if majority overload then
14:         inform update replicas of self demote;
15:         update replica repository information;
16:      else if minority overload then
17:         route read requests to sites ∉ overload
18:      end if
19:   end if
20:   if (load - previous load) > Δ_load then
21:      δ(T) - -;
22:      propagate load changes to update replicas
23:   end if
24:   if (previous load - load) > Δ_load then
25:      δ(T) + +;
26:      propagate load changes to update replicas
27:   end if
28: end while
```

Since our protocol is completely distributed the decision on what action to take and when is made mostly based on local information and the partially outdated information about the state of the system. Nevertheless, in order to maintain consistency, the updateable replicas are synchronously informed of any replica promote or demote. The extensions required by the Re:GRIDiT Protocol are illustrated in Algorithms 1 and 2.

4.2 Failure Handling

We identify the following types of failure that could occur in dynamic Data Grid environments. We suppose the sites fail-stop and that failures are detected by means of failure to acknowledge requests within a certain timeout. This timeout mechanism ensures that failure detection prevents processes from long delays but also, from suspecting sites too early which could incur unnecessary communication overhead. If a site crashes during an inform demote or inform promote phase, the failure is detected by means of the timeout mechanism. The site in charge of the inform will update the replica repository and reissue the inform request. If the site to be chosen promoted is failing to answer, the next best replica (according to Definition 1) is chosen to be promoted.

Since the number of copies of the object varies in time the dynamic replication and allocation algorithm is vulnerable to failures that may render the object inaccessible. To address this problem, we impose a reliability constraints of the following form: "The number of copies per data object cannot decrease below a minimum threshold". If such constraint is present, and the local site s fails to meet it, then any of sites informed of the demote p refuses to accept the exit of a data site, if such exit will downsize the replication scheme below the threshold. In other words, p informs s that the request to exit from the replication scheme is denied; subsequently, updates continue to be propagated to s. Site s continues to reissue the request whenever the preconditions for replica demote dictates to do so. The request may be granted later on, if the replication scheme is expanded in the meantime. Failure during other phases of the algorithm are considered in the semi-static replication protocol described in [26].

5 Implementation and Evaluation

The Re:GRIDiT protocol has been implemented using state-of-the-art Web service technologies which allows an easy and seamless deployment in any Grid environment. We have evaluated Re:GRIDiT with support for dynamic replica placement and deployment against Re:GRIDiT using a static replication scheme (presented in [25]), in a Java WS-Core Globus Toolkit container environment where the web services run. The experimental setup consists of multiple hosts equipped with a local Derby database and Java WS-Core. Both have been chosen due to their platform-independence, but any other existing off-the-shelf database can be used for this purpose.

The goal of these evaluations is to verify the potential of the protocol, in terms of scalability and performance, compared to a protocol allowing only a semi-static

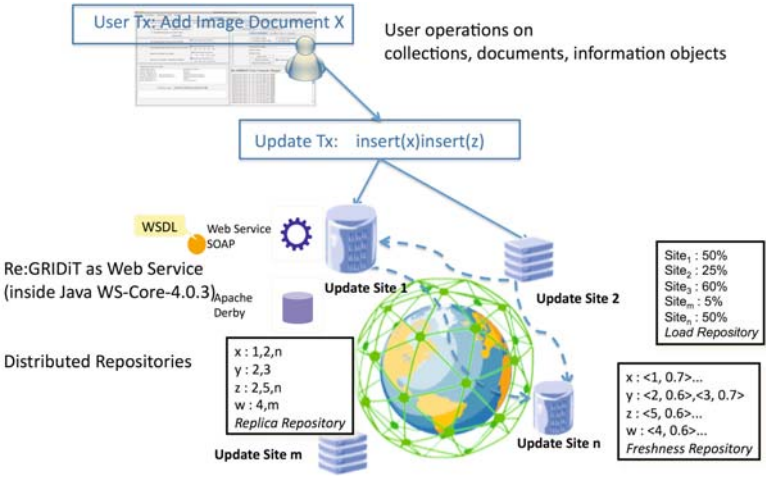

Fig. 5. System Model Implementation

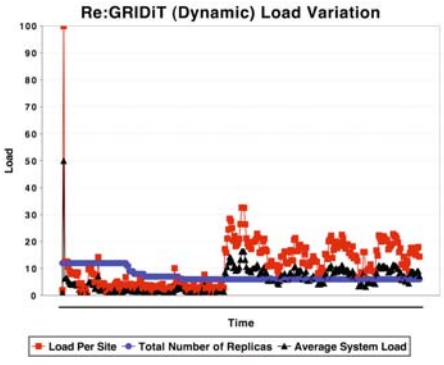

Fig. 6. Load Variation in Time. Dynamic Setup (12 Initial Update Sites).

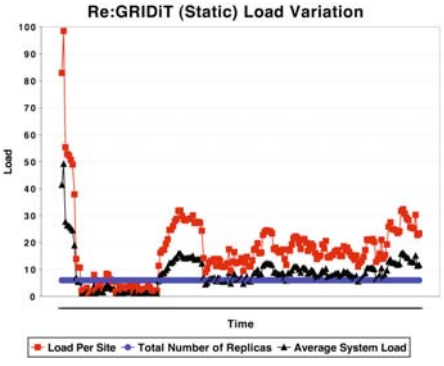

Fig. 7. Load Variation in Time. Static Setup (6 Initial Update Sites).

replication scheme. For the purpose of our evaluation we used up to 48 sites[2], equipped with a Dual Intel®CPU 3.20 GHz processor, 5 GB RAM and running Ubuntu Linux 8.0.4 as operating system. They are all running Java WS-Core version 4.0.3. The evaluation setup is schematically presented in Figure 5. For simplicity, we have used the site's CPU percentage as load measurement. The actual values used for the different load situations are depicted in Figure 4. These value have been chosen such that the intervals for a site promote or demote are comparable. Unless otherwise stated the measurements have consisted on runs of 100 transactions each. The conflict rate was set to 0.01 (since conflicts are assumed to be infrequent). Each transaction consists of 5 direct operations. The transactions were started sequentially, one after the other, with an update interval of milliseconds, such that as many transactions as possible are active at the same time.

5.1 Load Variation in the Presence of Active Transactions

In order to evaluate the performance of our protocol we have conducted several experiments that evaluate the load variation in the presence of active transactions at the sites. We have recorded the mean local load variations, the mean system loads and the evolution of the number of replicas in time (in a dynamic setting). As observed from Figure 6 for an initial number of 12 update sites, the average number of replicas in the system tends to remain stable at an minimum of 6. We have compared these results with the static replication protocol for which we used 6 update sites. The average system load shows more fluctuations in the dynamic case (due to the dynamic replica management overhead), and on average, the local site loads show a less than 5% increase with respect to the static case. In this case, the static setting of 6 update replicas has been chosen to match the optimal minimum which is reached by the dynamic setting; therefore an unfortunate choice of less than optimal number of replicas in the static setting produces non-negligible load differences with respect to the dynamic setting.

[2] This number refers to the maximum number of update sites in the system. In addition there may be hundreds of read-only sites present in the system.

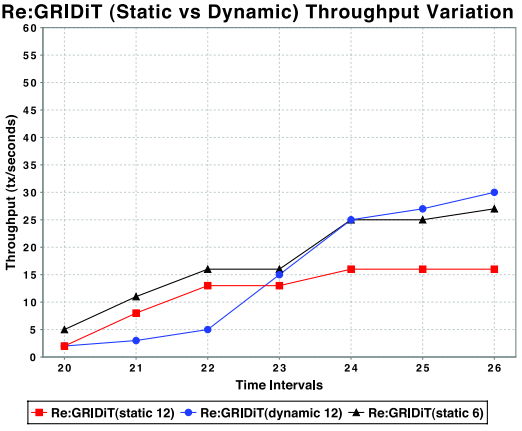

Fig. 8. Throughput Variation. 1 Run.

5.2 Evolution of Throughput

Another means of evaluating the performance of the dynamic and static Re:GRIDiT protocols is by comparing their throughput, calculated as the number of transactions committed within a time interval (in this example 20 seconds). In this measurement, both protocols with an initial setup of 12 sites and the static protocol with an initial setup of 6 sites have been compared. As in the previous cases, the dynamic replication protocol has stabilized the number of update sites at a minimum of 6. As it can be seen from Figure 8, the throughput in the dynamic setting is higher than in both static settings. The reason for this behavior is that transactions begin the commit much sooner in time in the dynamic case than in the static one. The rationale behind it is that the static Re:GRIDiT requires more time to synchronize the update to a higher (and constant) number of update sites in the case of 12 static sites, therefore the transaction duration is higher. It can also be observed that initially the throughput in the dynamic setting is smaller than in the case of the static setting with 6 update sites, due to the extra load imposed by the demote of the unnecessary update sites. Nevertheless, the throughput of the dynamic setting is increasing and outperforms the throughput of the static setting with 6 update sites, due to the selection of the best replica sites in the dynamic case.

5.3 Load Variation in the Presence of Active Transactions and Additional Load

As it can be seen from Figures 6 and 7, the distributed concurrency control of active running transactions and the replica management hardly drive the CPU load within the $heavyload$ level and never in the $overload$ level. In order to observe the system's behavior under heavier load stress we have tested the protocols in a setting using active transaction and artificial load variations (that mimic the behavior of additional read operations). These load variations introduce dynamic changes in the system which lead to promote and demote situations. In each of the cases the load has been maintained stable for several minutes in order to give the system enough time to react to the changes.

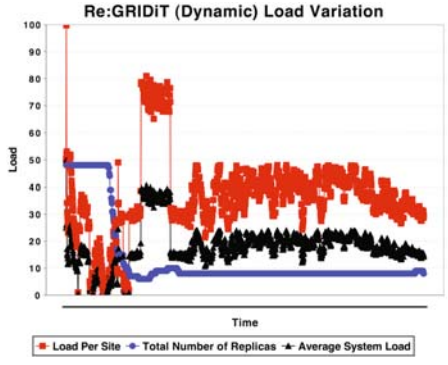

Fig. 9. Load Variation in Time. Dynamic Setup (48 Initial Update Sites).

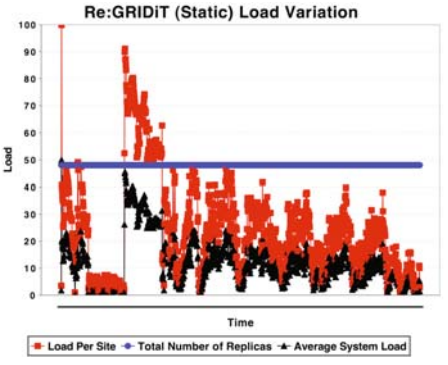

Fig. 10. Load Variation in Time. Static Setup (48 Initial Update Sites).

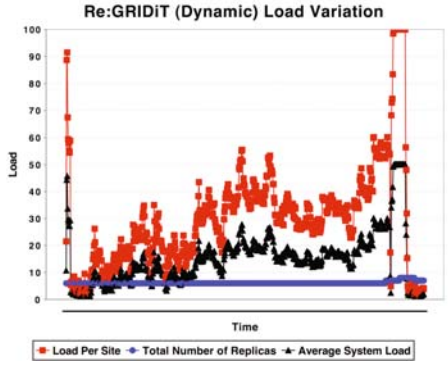

Fig. 11. Load Variation in Time. Dynamic Setup (6 Initial Update Sites).

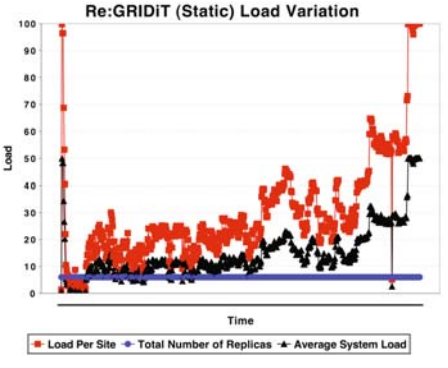

Fig. 12. Load Variation in Time. Static Setup (6 Initial Update Sites).

The results can be seen in Figures 9 and 10 where an initial setup of 48 update sites is compared and Figures 11 and 12 where an initial setup of 6 update sites is compared. It is clear that while both protocols are subject to the same load levels the dynamic Re:GRIDiT protocol is able to better cope with the varying load situations and consequently promote and demote sites as needed.

5.4 Dynamic Re:GRIDiT Protocol: Summary of Evaluation Results

Summarizing, the main achieved goals of our dynamic Re:GRIDiT protocol are:

– Dynamic: replicas can be created and deleted dynamically when the need arises. As it can be seen from Figures 9 and 11 dynamic changes in the load determines when new replicas need to be acquired or released.

- Efficient: replicas should be created in a timely manner and with a reasonable amount of resources. Figure 6 reflects the overhead in replica acquisition with respect to the static protocol presented in Figure 7 which is shown to be at approximately 5%.
- Flexible: replicas should be able to join and leave the Grid when needed. Our dynamic replication protocol allows replicas to join and leave the replication scheme as long as a minimum number of replica is present in the system. This limit is application dependent since different application scenarios might have different needs.
- Replica Consistency: in an environment where updates to a replica are needed, different degrees of consistency and update frequencies should be provided. In our measurements the Re:GRIDiT distributed and optimistic concurrency control protocol (presented in [25]) is enabled.
- Scalable: the replication system should be able to handle a large number of replicas and simultaneous replica creation. We have tested our protocol on a setup consisting of up to 48 update sites. In addition there may be hundreds of read-only sites.

6 Conclusions and Outlook

Data Grids are providing a cutting-edge technology of which scientists and engineers are trying to take advantage by pooling their resources in order to solve complex problems and have known intensive developments over the past years. Basic middlewares are available and it is now time for the developers to turn toward specific application problems. In this paper we propose an algorithm that aims at solving key problems of replication management in Data Grids. Re:GRIDiT hides the presence of replicas to the applications, provides replica consistency without any global component and allows an efficient, flexible and dynamic replication scheme where replicas can be created and deleted as the need arises. Furthermore we provide an evaluation of our dynamic replication protocol and compare it with a protocol which allows a static replication scheme, clearly proving the advantages of Re:GRIDiT.

Some open question remain for future research and certainly require future investigation: (i) how to respond in an efficient manner to read-only requests with different freshness levels or (ii) how to monitor access patterns (to improve access of heavy/frequent users to data).

References

1. Akal, F., Türker, C., Schek, H.-J., Breitbart, Y., Grabs, T., Veen, L.: Fine-Grained Replication and Scheduling with Freshness and Correctness Guarantees. In: VLDB, pp. 565–576 (2005)
2. Akbarinia, R., Pacitti, E., Valduriez, P.: Data currency in replicated DHTs. In: Proceedings of the ACM SIGMOD international conference on Management of data, pp. 211–222 (2007)
3. Andrzejak, A., Graupner, S., Kotov, V., Trinks, H.: Algorithms for Self-Organization and Adaptive Service Placement in Dynamic Distributed Systems. Technical report, HP (2002)
4. Breitbart, Y., Komondoor, R., Rastogi, R., Seshadri, S., Silberschatz, A.: Update propagation protocols for replicated databates. In: EDBT, pp. 97–108 (1999)
5. Breitbart, Y., Korth, H.F.: Replication and consistency: being lazy helps sometimes. In: PODS, pp. 173–184 (1997)

6. Candela, L., Akal, F., Avancini, H., Castelli, D., Fusco, L., Guidetti, V., Langguth, C., Manzi, A., Pagano, P., Schuldt, H., Simi, M., Springmann, M., Voicu, L.: DILIGENT: integrating digital library and Grid technologies for a new Earth observation research infrastructure. Int. J. Digit. Libr. 7(1), 59–80 (2007)
7. CERN. LHC Computing Centres Join Forces for Global Grid Challenge. CERN Press Release (2005), http://press.web.cern.ch/press/PressReleases/Releases2005/PR06.05E.html
8. Chundi, P., Rosenkrantz, D.J., Ravi, S.S.: Deferred Updates and Data Placement in Distributed Databases. In: ICDE, pp. 469–476 (1996)
9. Cohen, E., Shenker, S.: Replication strategies in unstructured peer-to-peer networks. In: SIGCOMM Comput. Commun. Rev, pp. 177–190 (2002)
10. EDG: The European DataGrid Project, http://eu-datagrid.web.cern.ch/eu-datagrid/
11. EGEE: The Enabling Grids for E-sciencE Project, http://www.eu-egee.org/
12. Gopalakrishnan, V., Silaghi, B., Bhattacharjee, B., Keleher, P.: Adaptive Replication in Peer-to-Peer Systems. In: ICDCS, pp. 360–369 (2003)
13. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The Dangers of Replication and a Solution. In: International Conference on Management of Data, pp. 173–182 (1996)
14. Harris, R., Olby, N.: Archives for Earth observation data. Space Policy 16, 223–227 (2007)
15. Jiménez-Peris, R., Patiño-Martínez, M., Kemme, B.: Are Quorums an Alternative For Data Replication. ACM Transactions on Database Systems 28, 2003 (2003)
16. Kemme, B., Alonso, G.: A new approach to developing and implementing eager database replication protocols. ACM Transactions on Database Systems 25, 2000 (2000)
17. The Laser Interferometer Gravitational Wave Observatory, http://www.ligo.caltech.edu/
18. Rathore, K.A., Madria, S.K., Hara, T.: Adaptive searching and replication of images in mobile hierarchical peer-to-peer networks. Data Knowl. Eng. 63(3), 894–918 (2007)
19. Röhm, U., Böhm, K., Schek, H.-J., Schuldt, H.: FAS: a freshness-sensitive coordination middleware for a cluster of OLAP components. In: VLDB 2002, pp. 754–765 (2002)
20. SRB: The Storage Resource Broker, http://www.sdsc.edu/srb/
21. Valcarenghi, L., Castoldi, P.: QoS-Aware Connection Resilience for Network-Aware Grid Computing Fault Tolerance. In: Intl. Conf. on Transparent Optical Networks, vol. 1, pp. 417–422 (2005)
22. Vingralek, R., Hasse-Ye, H., Breitbart, Y., Schek, H.-J.: Unifying concurrency control and recovery of transactions with semantically rich operations. Theoretical Computer Science 190(2) (1998)
23. Vingralek, R., Sayal, M., Scheuermann, P., Breitbart, Y.: Web++: A system for fast and reliable web service. In: USENIX Annual Technical Conference, pp. 6–11 (1999)
24. Voicu, L., Schuldt, H.: The Re:GRIDiT Protocol: Correctness of Distributed Concurrency Control in the Data Grid in the Presence of Replication. Technical report, University of Basel, Department of Computer Science (2008)
25. Voicu, L.C., Schuldt, H., Akal, F., Breitbart, Y., Schek, H.-J.: Re:GRIDiT – Coordinating Distributed Update Transactions on Replicated Data in the Grid. In: Proceedings of the 10th IEEE/ACM Intl. Conference on Grid Computing (Grid 2009), Banff, Canada (October 2009)
26. Voicu, L.C., Schuldt, H., Breitbart, Y., Schek, H.-J.: Replicated Data Management in the Grid: the Re:GRIDiT Approach. In: ACM Workshop on Data Grids for eScience (May 2009)
27. Weissman, J., Lee, B.: The Virtual Service Grid: an Architecture for Delivering High-End Network Services. Concurrency And Computation 14, 287–319 (2002)