Replication Heuristics and Polling Algorithms for Object Replication and a Replicating File Transfer Protocol

Silvano Maffeis*, Clemens H. Cap
{maffeis,cap}@ifi.unizh.ch

Institut für Informatik der Universität Zürich (IFI)
Winterthurerstr. 190, CH–8057 Zürich, Switzerland

IFI TR 92.06

July 30, 1992

ABSTRACT

This paper deals with the dynamic regulation of the level of replication in distributed systems, and leads to the introduction of *replication-heuristics* therefore. The question of *which* replica of an object should be referenced by operations when several copies are provided at different remote sites in the net is also addressed. *Polling-algorithms* performing this choice will also be presented. Replication-heuristics and polling-algorithms are an important research field in distributed, replicating systems but not adequately treated yet. The purpose of this work is to present possible solutions for these two problems and to suggest how the area should further be studied. As an important application of replication-heuristics and polling-algorithms, a replicating file transfer protocol (RFTP) is proposed.

KEYWORDS

Replication, replication-heuristics, polling-algorithms, file transfer, MUTS

1 INTRODUCTION

Object replication is an effective way for increasing availability and decreasing access times of data-objects distributed over a computer network. Such a network could either consist of several processors interconnected by highspeed-busses, or of many workstations in the same LAN, or even of a WAN made up by several LANs and associated long haul links. Questions arising when small data-objects ($<$ 100 kb per object) should be replicated over systems communicating via fast ($>$ 5 Mbits/sec) communication-paths have been investigated and practical results can be found in the literature (in [2] for example). These experiments are frequently conducted with a total replication of all objects over all nodes to obtain minimal access-times and maximal availability of the objects. The purpose of these experiments is often to compare different distributed programming-paradigms and measure speedup parameters in distributed test-applications, like matrix-multiplication or alpha-beta-search [1]. Other problems relating to replication, like consistency inbetween copies of an object, are often simplified or not treated at all.

The purpose of our work is to study heuristics adequate for *partially* replicating large objects in networks providing relatively slow communication facilities and long roundtrip delays. These assumptions hold for the WANs of today, like the `Internet`. In this paper we survey work concerning object replication in such large-scale systems underway in our research group.

Largely due to the inherent complexity of large-scale systems and the performance penalty paid for an inadequate level of replication, *replication-heuristics* regulating the level of replication in a distributed system are needed. This regulation takes place at runtime by adding or dropping replicas, when system-configuration changes are detected. Replication-heuristics are addressed in section 2.

A data object $\mathcal{O}$ may be provided in several copies $\mathcal{O}'$ residing on different sites of a network. These copies are supposed to be held consistent by some consistency-granting mechanism. Now, supposing a site $\mathcal{S}$ wants to access object $\mathcal{O}$, the question arises *which* specific copy $\mathcal{O}'$ should be addressed to ensure a minimal latency time or a high availability of the object or both. What we call *polling-algorithms* is needed to solve the problem, and will be introduced in section 3.

Section 4 gives a short introduction to the research-methods, mainly simulation, employed in our work. In section 5 a *replicating file transfer protocol* is proposed to demonstrate and analyze replication-heuristics and polling-algorithms.

2 REPLICATION-HEURISTICS

2.1 OVERVIEW

An object *replication-heuristic* comprises a set of rules, defining *which* objects are replicated *when* (conditions), and *where* (locations) the new replicas will be placed.

Replication-heuristics act as controlling unit in a replicating system, by dynamically adapting the level of replication to changes in a distributed system. Changes happen due to events like node– or gateway failures, bursts of re-

*Supported by *Siemens AG, ZFE*, Germany and *Schweizer Bundesamt für Konjunkturfragen*, Grant No. 2255.1

quests to certain objects, new objects being created and put into the system and so forth.

The design of a distributed service should define the level of performance and availability the system must offer under normal conditions. Replication-heuristics form the basic mechanism for assuring that parameters like *performance* or *availability* are held constant as long as possible, but are suitably adapted in case of unpredictable changes happening to a distributed system.

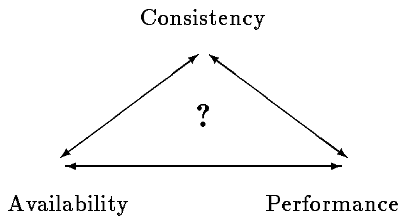

Figure 1: Dependencies in replicating systems

Most distributed applications require some form of *consistency* between copies of an object. At a certain level of replication, the goals *availability* and *performance* (figure 1) may begin to concur, since replication induces both performance gains, in form of faster access to a local copy, but also performance losses, since copying and migrating objects involves some overhead. Preserving consistency between replicas also has an impact on performance and availability. Depending on the many parameters of a distributed system, all sorts of relationships are possible between the three goals, and the relationships cannot be predicted a-priori. We can speak of some kind of 'Replication-Paradoxon'. For solving this problem, replication-heuristics dynamically adapt the level of replication according to the goals set for a replicating service. These goals consist of demands concerning performance, availability, consistency and so forth, or a combination thereof.

The need for replication-heuristics is also expressed by K.P. Birman in [3, 4]: *" rather than asking how much such a [Brokerage-] application must pay for fault-tolerance, more appropriate questions concern the level of replication at which the overhead begins to outweigh the benefits of concurrency. ...fault-tolerance is something of a side-effect of the replication approach ... "*. Or concerning group membership lists in ISIS: *" it is by no means clear a-priori whether membership lists should be replicated at all group members or cached at some smaller set of sites ... "*.

The opposite of a replication-heuristic is the fixed or manual replication scheme. With fixed replication, a system-administrator or the users themselves manually specify the number of replicas for every object, and on which nodes of the network the replicas will be placed. Similar mechanisms are provided by replicating filesystems like RNFS, ECHO, LOCUS, FICUS or CODA [14].

A system which incorporates what we call a replication-heuristic is the DECEIT-filesystem [15]. DECEIT provides a simple heuristic which we call the NAIVE heuristic (see 2.3): Whenever a client requests to access file $\mathcal{F}$ at fileserver $\mathcal{S}$, but $\mathcal{S}$ does not dispose of a local replica, then the request is forwarded to a fileserver containing a replica of $\mathcal{F}$, and a replica of the file is added to the workingset of $\mathcal{S}$ to speed up future accesses.

2.2 SOME EFFECTS OF OBJECT REPLICATION

Varying the level of replication can lead to the following two problems:

TOO-FEW-REPLICAS EFFECT: When a replication level gets too low, references to an object must frequently be rerouted to a remote holder of a replica, and long delays may occur for data-traffic intensive tasks. In this case, system-performance and availability can be increased at the same time by providing the system with further replicas.

TOO-MANY-REPLICAS EFFECT: Assuming an object which is frequently read but most rarely updated, the administrator of a distributed service may take advantage of this fact by creating and disseminating many replicas of the object. In the extreme case, one replica for every node in the network. Now, suppose the arrival rate of write-operations to the object drastically increases. Assuming a consistency protocol like *read-one-write-all* or *voting*, the system would spend an inappropriate amount of time in transmitting updates for holding the replicas consistent. We call this state, where a distributed system spends most of its resources for such housekeeping functions *distributed thrashing*.

Theoretically, the optimal replication level can be found by comparing cost and benefit of creating additional replicas of an object. Nevertheless, under real conditions, such an analysis cannot be performed in many large-scale systems, since many parameters influencing cost and benefit are not traceable or cannot be quantified at runtime. We therefore provide an *upper* and *lower replication-boundary*, which should not be surpassed by the system. If a replication level is surpassed, one of the described negative effects may occur. The computation of these boundaries is related to the replication-heuristic and depends mainly on parameters like the read/write ratio of the objects, consistency constraints in between replicas, object-sizes, link and node failure probabilities, host performances and the capacity of the communication-links. The main task of the system component implementing the replication-heuristic is to dynamically adapt these boundaries along defined rules (the heuristic) and to vary the level of replication according to the boundaries.

2.3 ELEMENTARY HEURISTICS

The development and study of replication-heuristics shall be motivated by providing some elementary heuristics, which can be enhanced and refined in accordance with the applications they are needed for:

NAIVE: Whenever a site $\mathcal{S}$ accesses an object $\mathcal{O}$ for the *first* time, a replica is generated for $\mathcal{S}$, and put into its workingset of local replicas. The upper replication-boundary for NAIVE is limited only by the (storage) capacity of the distributed system. Distributed thrashing (see 2.2) may thus occur in systems relying only on NAIVE. We provide this heuristic mainly as example for analyzing other heuristics. A replication-heuristic similar to NAIVE is provided in DECEIT [15].

LIMIT: This heuristic works like NAIVE with two additional constraints limiting the number of replicas for object $\mathcal{O}$. Therefore, we introduce the upper replication-boundary $\mathcal{O}_U$ and the lower replication-boundary $\mathcal{O}_L$. LIMIT avoids *distributed thrashing* and *too-few-replicas*, provided that both boundaries adapt to changes in the system. Different $\mathcal{O}_U$ and $\mathcal{O}_L$ values may be defined for every Object $\mathcal{O}$.

COUNT: Only after having referenced an object $\mathcal{O}$ for $C_{\mathcal{O}}$ times, $\mathcal{S}$ obtains its own replica. This strategy is suitable for systems, where a large part of objects is accessed only few times by the sites in the net. Where LIMIT or NAIVE would induce excessive copying and migrating of objects, COUNT only replicates objects which get referenced for a certain number of times. Upper and lower replication-boundaries are defined also for COUNT. Counters must be reset to their initial value from time to time.

INTENSITY: INTENSITY enhances COUNT by allowing a replication only when $\mathcal{O}$ has been accessed for more than $C_{\mathcal{O}}$ times during the time interval τ. Furthermore, INTENSITY has the advantage of handling bursts of references to an object by replicating it when the intensity-level $C_{\mathcal{O}}/\tau$ is surpassed. Thus, locality of reference to objects can be exploited.

There is no universal replication-heuristic suitable for every system in any situation. Certainly situations exist, where simple heuristics allow controlled object-replication and avoid abnormal events like distributed thrashing. A suited replication-heuristic must be chosen and parametrized conforming to the environment it will be used for.

It is not clear yet how replication boundaries and intensity-levels may be parametrized in large networks, since the relevant parameters are system-dependent, and many influencing factors exist. Part of our work is concerned with finding out which influencing parameters are important in specific situations. Research methods used therefore will be described in section 4.

The management of each site's workingset also leads to many interesting algorithms, which are comparable to algorithms used for virtual memory management, for example LRU, LFU or the *page-fault-frequency* (PFF) algorithm of Chu and Opderbeck [5]. Concerning the implementation of replication-heuristics, a process group approach [4] seems to be appropriate, since replicas of a certain object can be managed by a group of dedicated processes, and information needed by the replication-heuristic can be exchanged by means of group-multicast.

2.4 BENEFITS OF REPLICATION-HEURISTICS

In this section we summarize advantages related to the use of replication-heuristics. This in contrast to the case where replication is relinquished to the user or 'hard wired' into the application, in form of a fixed replication level:

- By using replication-heuristics, a certain level of availability and performance can be ensured despite communication link failures and other defects. Manually configuring a replicating system by an experienced administrator may yield to better performance or availability in some situations. But using a replication-heuristic, the replication level can be enlarged or reduced dynamically in case of detectable configuration changes which unpredictably happen to a distributed system. *Graceful degradation* of performance and availability can thus be ensured.
- Replication-heuristics may be combined with manual administration. For example, DECEIT allows users to manually create or delete replicas or to specify some minimum replication level. This is important for flexible filesystems providing some form of disconnected operation (e.g. Coda [9]). Here, the experience of the user or system-administrator is needed to specify which objects (files) are essential for the disconnected (stand-alone) operation of a client workstation.
- Large, replicating systems managing thousands of objects cannot be organized manually any more. Only replication-heuristics can handle object replication under such conditions in an efficient way.
- Situations like *distributed thrashing* and *too-few-replicas* can be avoided by providing self-adapting heuristics.
- The proposed heuristics are scalable in the sense that the number of network messages needed grows slowly with the size of the network.
- In large distributed systems replication comes down to engineering considerations a typical user of a system might not be able to handle. Owing to replication-heuristics, users don't have to worry under which conditions an object should or should not be replicated.

3 POLLING-ALGORITHMS

3.1 OVERVIEW

Suppose we have n network-sites and several data objects stored on some of these sites. If a request to access a data object arises at a certain site, there are two possibilities: Either the object is stored locally on this site and then it usually is the fastest solution to access this local copy, or the object is not stored locally, and then an access to a remote site is tried. For this purpose, the issuing site *chooses a set of remote locations to poll for an access.* A request packet is sent out to every element of this set. If a remote site receives such a request, it performs a local access and returns the answer. An optimization for large objects is described in 3.3.

Algorithms deciding in this manner *which* remote site's replica should be accessed will be called *polling-algorithms* in the following. Their principal structure is as follows: The algorithm starts in an initial state. Depending on its actual state, the algorithm chooses a set of remote sites, which will be polled for an access. As soon as the first remote reply arrives, the algorithm temporarily records the identification-number of the remote site and the time it took from sending the request to receiving this reply. All other replies, which might arrive later, are ignored or used for updating the algorithm's data base storing information on the state of the remote nodes. We do not wait arbitrarily long for the first remote reply. Depending on the actual state a timeout is set, after which the algorithm proceeds either with a retry or by informing the calling layer accordingly.

We can distinguish two kinds of polling-algorithms: *Single polling-algorithms* only poll a single site, whereas *multiple polling-algorithms* might poll a number of sites concurrently. A special kind of multiple polling-algorithms may be designed, which polls only a single site whenever the system works reliably, but polls a number of sites, when the failure rate increases. The latter can be estimated based on the frequency of detected timeouts.

3.2 ELEMENTARY POLLING-ALGORITHMS

FRA (Fixed Remote Access): This simplest polling-algorithm always polls a single *fixed* remote node or a fixed set of nodes. A similar algorithm is implemented in SUN's *Network Information Service (YP)*, where the clients always choose a fixed YP-master. We provide FRA mainly for comparison reasons.

LFR (Last Fastest Response): The single LFR polling-algorithm maintains a state space for recording the duration of the last successful access of every remote node in the poll-set. For polling, the site with *minimal last response time* is chosen. The multiple LFR algorithm polls a set of nodes, which are within a given threshold of the minimal recorded time.

BAR (Best Average Response): The single BAR algorithm does not concentrate simply on the last response time but on the *average response time* encountered since startup of the site. The multiple BAR algorithm polls a set of sites, which are within a given threshold of the minimal average time.

DAR (Decayed Average Response): System load varies rapidly over time. Therefore, instead of overall average values a gradually decaying average should be used. This is done in the DAR algorithm. Here, the overall average from the BAR algorithm is replaced by a decaying average concentrating only on the last m values and using a set of weights.

3.3 ADAPTING THE DAR ALGORITHM

For intuitive reasons which will be strenghtened by theory, simulation and experiment, we believe the DAR algortihm is most suitable for a large number of situations. There is a number of possible modifications to DAR which may also be made to some of the other algorithms:

The first modification allows *adaptive timeouts.* Up to now we only considered constant timeout functions, where the predefined timeout constant might prove to be too small or too large for the actual situation. We can construct an easy form of timeout adaption by specifying an increment ε_t, two boundaries $0 \leq T_L < T_U \leq 1$ and a number $N_t \in \mathbb{N}$. Whenever the portion of timeouts in the last N_t requests is outside the interval $[T_L, T_U]$, the value of the timeout function is raised or lowered by the increment ε_t. T_U provides for a maximal percentage of timeouts that we will accept before raising the timeout value. T_L provides for a minimal percentage of timeouts, which is necessary, because otherwise the timeout value would always become large. N_t specifies how far back in the past the algorithm should look and hence specifies some kind of decay for this mechanism. ε_t influences the speed of adaption.

The second modification provides *adaptive thresholds* for *multiple* polling-algorithms. For multiple polling-algorithms, we start with a predefined response-time value, and poll all those sites, whose response-time was within a threshold distance δ from the respective predefined value. We now present an adaption mechanism for this threshold δ. We again supply an increment ε_δ, two boundaries $0 \leq D_L < D_U \leq 1$ and a number $N_\delta \in \mathbb{N}$. Whenever, over the last N_δ requests, the percentage of sites chosen for a remote access is outside $[D_L, D_U]$, the value of the threshold is raised or lowered by the increment ε_δ. D_U provides for a maximal percentage of sites that we would concurrently poll, D_L provides for a minimal percentage. N_δ specifies how far back in the past the algorithm should look and hence specifies some kind of decay for this mechanism. ε_δ influences again the speed of the adaption. The parameters of both adaptive techniques must be chosen carefully in order to prevent the thrashing phenomena mentioned in section 2.2.

The third modification concerns the used *protocol.* If the replies to accesses contain only small amounts of data, it is a good idea to immediately answer such queries. However, if *larger amounts of data* must be sent, the remote sites first should send only an acknowledgement packet, informing the polling site of their availiability, including possibly also information on the approximate time needed to compute and send this answer. Afterwards, the polling site may decide which remote site should in fact perform the data– and computation-intensive access.

When updates and consistency properties must also be considered, the polling algorithms may immediately be connected with optimistic or pessimistic consistency algorithms. *Coteries* or *voting* techniques for example turn out to be an extension to polling. Approaches to consistency like atomic broadcast may also be combined with polling.

To some extent, polling-algorithms are similar to *paging algorithms* in a virtual memory system. However, the main difference is, that paging comprises not only accesses but also the full transfer of the accessed object (the memory-page) to a lower level storage at every access. In our situation this is not feasible, since the individual object may be very large and the accesses may comprise complex queries on this object, not furnishing information on the whole object. Assumptions of paging algorithms like locality of accesses or constant access times to the storage, are no more valid in the framework of replicated objects. Furthermore, polling has to deal with several replicas of an object simultaneously, paging only with single objects (the memory-pages).

4 STUDYING REPLICATION-HEURISTICS AND POLLING-ALGORITHMS

Having realized the importance of adequate replication-heuristics and polling-algorithms, the question arises how such heuristics can be parametrized and studied. Developers and administrators of distributed, replicating services need support in engineering heuristics and testing them under realistic conditions, without having to experiment on their real systems and with the patience of the users.

A solution to this problem consists in a configurable simulator, which can be fed with data concerning the connectivity of the simulated network, host parameters, network parameters, objects and queries as well as with a specification of the replication-heuristic or polling-algorithm to be simulated. The results provide information concerning access times to the various objects, maximal throughput (transactions/sec) achievable with specific heuristics, availability, update traffic and so forth.

Our group is developing the replication-simulator RESIM, which is able to simulate replication-heuristics and polling-algorithms in `Internet`-like networks. RESIM can be parametrized for different network-settings and access-modes, and we intend to use the simulator for the project described in section 5. Details concerning the simulator will be published in a separate paper.

Other investigation methods employed include *queueing models* and methods used for studying resource allocation problems.

5 APPLYING REPLICATION-HEURISTICS AND POLLING-ALGORITHMS

Applications suited for replication-heuristics and polling-algorithms comprise several kinds of remote-access applications. Enhancing X.500 services with object replication has already been proposed [8]. Motivated by this fact and by studies conduced by M. F. Schwartz [13], we envision a *Replicating File Transfer Protocol* RFTP, due to several improvements which could be incorporated into the wide-spread `ftp` [12] utility.

5.1 ENHANCING THE FTP UTILITY

DUPLICATE FILE TRANSFERS: A recent study [7] reports that more than 50% of the `ftp` data-traffic relates to files, which previously have been transferred to the same requesting site. By using file-replication and providing an adequate browsing or directory-lookup facility, this useless waste of network resources could be decreased.

THROUGHPUT: A replicating file transfer protocol would increase throughput and decrease access latencies to frequently accessed files.

NAMING: A user has no possibility to organize the name space in a transparent and user friendly way. Naming problems may occur when files with different contents are given the same name. In such cases, users frequently retrieve many files with the same name from several archives, to afterwards decide locally, which one contains the needed data. Such wastes of network resources can be reduced by providing appropriate naming-mechanisms.

One possible solution consists in enhancing the file transfer protocol to make use of a virtual file system like PROSPERO [11]. In PROSPERO, a user can construct own views of the files that are accessible. Furthermore, PROSPERO offers other useful customization features like user-defined filters and union links.

LOCATING: Because networks rapidly expand, resource location becomes a serious problem [13]. Without the ability to discover files of interest, users perceive only a limited fraction of the full potential of sharing resources.

Utilities like `archie` [6], providing the user with capabilities for easily browsing remote archives by searching for specific filenames, yield only a limited solution. These tools usually lack the possibility to inform the user about the nearest location of a file or about availability of sites providing a copy of the file. In this context, the term 'nearest' is not used in its geographical sense but from the viewpoint of fast file access and transfer.

VERSION-MANAGEMENT: Copies of files are often scattered over many archives in a network and may coexist in several versions, many of them being out of date. Having found a set of sites providing a copy of a certain file (e.g. via `archie`), it becomes hard or even impossible to predict *which* file is the newest version, by only relying on the available information, namely *location-site*, *size* and *modification time*. This problem could be alleviated if version-numbers would be part of filenames. For example, refer to the naming scheme applied by the GNU-staff: Users willing to obtain the newest version of, say, the GNU `gcc` distribution will find it easy to distinguish a new `gcc` package from older ones, since filenames have informative formats like `gcc-1.40.tar.Z` or `gcc-2.1.tar.Z`.

Two problems still remain: First, only few distributions provide such informative filenames. Second, being `gcc-2.1.tar.Z` the newest version stored at site `x.y.edu` does not mean that no newer version exists anywhere. This relates to the locating problem described above. It would be useful if information concerning new versions could be 'posted' to all RFTP archive sites. Browsing the GNU archive, a RFTP-user at domain `x.y.edu` would obtain following information:

```
% rftp
...
rftp> dir
PORT command successful.
Opening data connection for VIEW=gnu NAME=gcc

gcc-1.40.tar.Z (local)
gcc-2.1.tar.Z (local)
gcc-2.2.tar.Z (aeneas.mit.edu)

rftp>
```

Assumed that enough RFTP-users at `x.y.edu` retrieve `gcc-2.2.tar.Z`, the replication-heuristic would decide to replicate the stated file from `aeneas.mit.edu` to `x.y.edus` local workingset. Afterwards, the RFTP users at `x.y.edu` would see

```
gcc-1.40.tar.Z (local)
gcc-2.1.tar.Z (local)
gcc-2.2.tar.Z (local)
```

Further references would be satisfied using the new replica. Note that replication can also be performed by managing one central workingset for many network nodes, in our example for the domain `y.edu`. This would be especially useful if file-accesses are performed beyond the boundary of a local network or domain.

AVAILABILITY: Replication also augments availability of files, by maintaining copies at different locations.

CONSISTENCY: Strong consistency in form of *one-copy-serializability* would lead to problems when operating RFTP over WAN settings. Rigid consistency-mechanism would induce a considerable network traffic, and complicated consistency-protocols would be needed in face of the autonomous structure of the `Internet` and unforeseeable unavailabilities of network resources.

We feel that strong consistency is not needed by most users, since it surely would be sufficient to inform RFTP sites about changes to files by using the incorporated RFTP 'news-posting' mechanism, which will be introduced in section 5.3. Each site itself would then decide upon obtaining new versions or patches of a changed file. The patching approach has the advantage that a history of changes can be maintained, yielding the possibility of reconstructing older versions of files. Network traffic would also be reduced by transferring patches instead of complete files. This way, a weaker form of consistency can be achieved at low cost.

5.2 CONCEPTIONING THE RFTP SERVICE

The RFTP application layer is based on a service access point to procedures for maintaining process groups, group communication, synchronization, distributed execution and so forth. The package providing for the described functionality should easily be obtainable for `Internet` sites interested in using the RFTP service, and also be compatible with several operating systems and architectures. Compactness and ease-to-use should also be ensured.

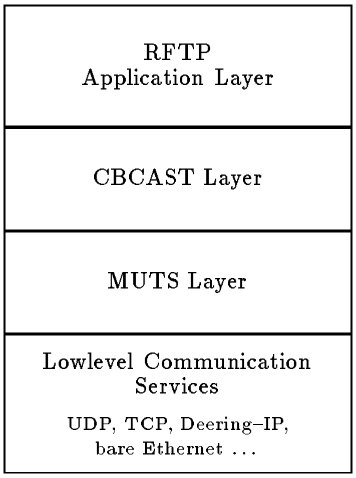

Figure 2: RFTP layers

We therefore envision a structure similar to the one planned for the new Isis implementation [17], consisting mainly in the employment of MUTS [16], as demonstrated in figure 2. MUTS provides most services needed for realizing RFTP.

In our project, RFTP is used for studying and solving problems related to the replication of data objects in heterogeneous, distributed large-scale systems. We primarily aim at achieving goals like short access times to common files, availability and dynamic replication. This will be done by applying the results we obtain researching replication-heuristics and polling-algorithms.

5.3 MAINTAINING REPLICATED FILE-FINGERPRINTS

We believe that an important naming and locating problem can be solved in RFTP by means of object-replication. Suppose a user knows about a text-editor named `ted`. He expects `ted` to be stored in a file `ted.tar.Z` on some archive in the net. The common approach to find `ted.tar.Z` would be to have `archie` search for the specified `tar`-file. We did this recently and found eight files matching the name `ted.tar.Z` on eight different `Internet`-archives. Six of these eight files had different sizes and modification-times. How can one decide which file contains the desired text-editor? In the worst case, the user would have to retrieve all eight files, to realize the first seven did not contain `ted`. This frequently occurring naming problem leads to an unacceptable waste of network-resources and working time.

DESCRIPTION FILES

One simple solution to this problem would be to have each site-administrator compose a special configuration file containing a short description of every file in his archive. An enhanced file-transfer program could read this file and provide users with short descriptions of the files stored at the browsed remote archive. This simple solution suffers from some drawbacks: Site-administrators don't know about every file's meaning and content, since many users will have some of their own files stored in the archive for public access, or mirroring [10] of remote ftp-sites is used. Further, many administrators lack the needed time or motivation for accurately maintaining such description files. A further difficulty of this approach is the maintenance of consistent descriptions over several archives.

REPLICATED FILE-FINGERPRINTS

A better solution which adequately can be implemented using object replication, is to maintain a data-object at RFTP sites containing what we call *file-fingerprints.* A file-fingerprint has two parts: The first is a checksum which is computed for the file, serving as identification-key for the file. The computation of such a checksum can be performed by taking each l/n–th byte of the file, with l being the length in bytes of the file and n an appropriate value. The probability of using the same values when the checksum of two different files is computed would thus be less than or equal 256^{-n} for files with $l \geq n$. The second part of the file-fingerprint contains structured information concerning the name of the file, version, distribution, name of the author and so forth. This way, RFTP users can attach information to files they created. These fingerprints would then be 'posted' to other RFTP sites by using procedures similar to those involved in USENET and which may be realized easily using MUTS-primitives (see the `newstool` contained in the MUTS distribution for example). Each site would then maintain a partially replicated dataobject containing fingerprints and descriptions relating to local files. The locating of files is performed by composing RFTP queries containing what the user knows about the related fingerprint. For example:

```
% rftp
rftp> VIEW=gnu NAME=gcc DOMAIN=local,mit.edu
rftp> dir
PORT command successful.
Opening data connection for VIEW=gnu NAME=gcc
...
```

Clearly, only a part of all files stored at the RFTP archives would be documented by the replicated fingerprint-facility. The facility should therefore be enhanced, for example by the `magic` number mechanism applied in UNIX. As proposed in [7], RFTP could automatically switch to binary-transfermode whenever the `magic` of the file to retrieve recommends it, diminishing the probability of bandwidth wastes due to the retrieval of garbled data, which is frequently caused by an inappropriate setting of the `binary`-filetype parameter in `ftp`.

This also means that whenever a fingerprint is not defined for a certain file, some minimal information could be offered using the `magic` number of the file (for example 'PostScript', 'ASCII' or 'Compressed tar').

Users willing to share their files with the RFTP community will create and post the specific file-fingerprints. RFTP sites only need to provide fingerprints related to files stored locally. Upon receiving a fingerprint, the site checks its fingerprint database for an empty entry relating to a local file. If such an empty entry is found, the fingerprint is added, else the fingerprint is discarded.

5.4 RELATED WORK

We know about similar `ftp`-projects being elaborated by Michel F. Schwartz [13], by the *Internet Anonymous FTP Archive Working Group* (IAFA-WG), by Martijn Lindgreen (EUnet Distributed Archive) and by Vincent Cate who develops the ALEX filesystem. Our contribution is in form of a dynamic RFTP service using some of our replication-heuristics and polling-algorithms. We also envision to gain some insights into MUTS and to realize an application based on this service running over LANs and also over WAN links.

Further ideas to our service come from useful tools like `archie` [6] and `mirror` [10].

6 CONCLUSIONS AND FUTURE ACTIVITY

Replication-heuristics and polling-algorithms enclose several interesting and yet not adequately investigated problems. We believe that our efforts in this field will yield to new insights which may be applied in many distributed, replicating services. The methods we employ in investigating the matter involve simulation, analytical approaches like queueing theory and methods used for studying resource allocation problems. We also develop a *replicating file transfer protocol* RFTP to gain insight into problems related to data-object replication in large-scale systems, and to test replication-heuristics and polling-algorithms under real world conditions.

The work in progress consists of determining suitable working-parameters of replication-heuristics and polling-algorithms for typical situations, and studying the resulting behavior using our replication simulator RESIM. In a further step, we will apply the results by parametrizing the proposed heuristics and algorithms to other replicating services, like a distributed database of multimedia information or a replicating domain name service.

REFERENCES

[1] BAL, H. E. A Comparative Study of Five Programming Languages. Tech. Rep. IR-250, Vrije University, Amsterdam, June 1991.

[2] BAL, H. E., AND KAASHOEK, M. F. Data Distribution in Orca Through Compiler Optimization. Tech. Rep. IR-254, Vrije University, Amsterdam, Sept. 1991.

[3] BIRMAN, K. P. Maintaining Consistency in Distributed Systems. Tech. Rep. 91-1240, Cornell University, Nov. 1991.

[4] BIRMAN, K. P., COOPER, R., AND GLEESON, B. Design Alternatives for Process Group Membership and Multicast. Tech. Rep. 91-1257, Cornell University, Dec. 1991.

[5] CHU, W. W., AND OPDERBECK, H. The Page Fault Frequency Replacement Algorithm. In *AFIPS Conference Proceedings, Fall Joint Computer Conference* (1972), vol. 41, pp. 597–609.

[6] EMTAGE, A., AND DEUTSCH, P. archie – An Electronic Directory Service for the Internet. In *Conference Proceedings Usenix* (San Francisco, CA, Jan. 1992).

[7] EWING, D. J., HALL, R. S., AND SCHWARTZ, M. F. A Measurement Study of Internet File Transfer Traffic. Tech. Rep. CU-CS-571-92, University of Colorado, Jan. 1992.

[8] HARDCASTLE-KILLE, S. E. Replication Requirements to Provide an Internet Directory using X.500. RFC 1275, University College London, Nov. 1991.

[9] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected Operation in the CODA File System. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992).

[10] MCLOUGHLIN, L., ET AL. *Mirror – Mirror Packages on Remote Sites.* (UNIX man–pages).

[11] NEUMAN, B. C. The Prospero File System. In *Proceedings of the USENIX File System Workshop* (Ann Arbor, Michigan, May 1992), pp. 13–28.

[12] POSTEL, J., AND REYNOLDS, J. File Transfer Protocol (FTP). RFC 959, USC Information Science Institute, Oct. 1985.

[13] SCHWARTZ, M. F. Resource Discovery in the Global Internet. Tech. Rep. CU-CS-555-91, University of Colorado, Nov. 1991.

[14] SIEGEL, A. *Performance in Flexible Distributed File Systems.* PhD thesis, Faculty of the Graduate School, Cornell University, May 1992.

[15] SIEGEL, A., BIRMAN, K. P., AND MAZULLO, K. Deceit: A Flexible Distributed File System. Tech. Rep. 89–1042, Cornell University, Nov. 1989.

[16] THE ISIS GROUP. *MUTS Documentation.* Cornell University, Aug. 1991.

[17] VAN RENESSE, R., BIRMAN, K. P., COOPER, R., GLADE, B., AND STEPHENSON, P. Reliable Multicast between Microkernels. Cornell University, 1992.