Transactional Replication for Typed Objects: Preliminary Results of Project BURDS*

Abdelsalam Heddaya†
Prakash S. Ramamurthy‡
Himanshu Sinha

Boston University

September 1,1990
Revised January 21, 1991

Abstract

Project BURDS (Boston University Replicated Data System) aims to experiment with event-based replication for typed data, based on transactions. The first phase involves measuring the performance of Gifford's weighted voting replication protocol for files, augmented by caching and reconfiguration protocols. This paper reports the preliminary results of these measurements and sets forth the rationale behind the architecture of BURDS.

The next phase of project BURDS will focus on implementing Herlihy's general quorum consensus, together with the two phase gossip [7] and view-based reconfiguration protocols [8]. This phase will be concerned primarily with typed objects replicated transactionally.

Our primary implementation tool has been the Argus programming language and system, a distributed extension to CLU. We report on some of the advantages and shortcomings of Argus in supporting server parallelism and concurrency control of cache data.

To appear in the Proc. of the Workshop on Transactions and Objects, to be held in Ottawa, Canada on October 21, 1990.

*This research was supported by NSF grants number IRI-8910195 and CDA-8920936.

†Address: Computer Science Dept., Boston University, 111 Cummington Street, Boston, MA 02215; Internet: *heddaya@cs.bu.edu*.

‡Current address: Mentor Graphics Corp., Silicon Design Division, 2045 Hamilton Ave., San Jose, CA 95125.

1 Introduction

An object[1] system must *store* data reliably, provide *data access* with high availability, and invoke *operations* on objects efficiently. Project BURDS (Boston University Replicated Data System) constitutes a vehicle for the study of the last two aspects of an object system. We examine different methods for increasing data availability by replication, concentrating on voting and reconfiguration protocols. We enhance the client/server model of distributed computing using freely replicated shared caches capable of executing object operations, and measure the effects of different cache management protocols.

In this paper, we advocate a specific architecture (see figure 1) and report on preliminary performance data obtained for a particular suite of protocols. The primary features of our architecture include:

- Data replication among *back-end* processes that use stable storage, with replication controlled by voting to provide high availability in the presence of undetected site and communication failures.
- Dynamic reconfiguration of the replicated objects to improve availability under detectable repairable failures and support dynamic object migration.
- *Front-end* processes encapsulate the code implementing type operations as well as volatile cached copies of object states and configurations. When the cache hits, our scheme incurs no extra communication or I/O cost for cache validation, and reduces the number of message rounds needed for read and update operations by one. Since front-ends can be shared among many clients, we can simultaneously increase the cache hit ratio and reduce the overhead of cache maintenance per client.

  Front-ends can be replicated freely because the data they store is treated only as hints (*i.e.*, the data is always validated before it is used). This can be used to ensure that the front-ends will be at least as available as the back-ends.

One instance of our proposed architecture has been implemented using the Argus distributed programming language and system [10]. In this instance, we use Gifford's weighted voting algorithm [5], to replicate an untyped *file* at different back-ends, each of which stores a copy of a version of the file, together with its version number. To execute an operation on such a replicated file, the front-end must obtain the consensus of a quorum of the back-ends. We have also implemented a reconfiguration protocol that was originally proposed by Gifford [5], and detailed by Goldman [6].

Section 2.1 below briefly describes both weighted voting and reconfiguration protocols. Our caching protocol maintains a cache of both the file configuration and the file contents; the essential features of this protocol are given in figure 2 and section 2.2. A statement and discussion of our preliminary results follows in section 3. We conclude the paper with a short review of related work and a general discussion.

The rest of this section outlines the features of Argus most essential for the clarity of our exposition.

[1] By *object* we mean an instance of an Abstract Data Type, associating a set of operations with a hidden persistent state.

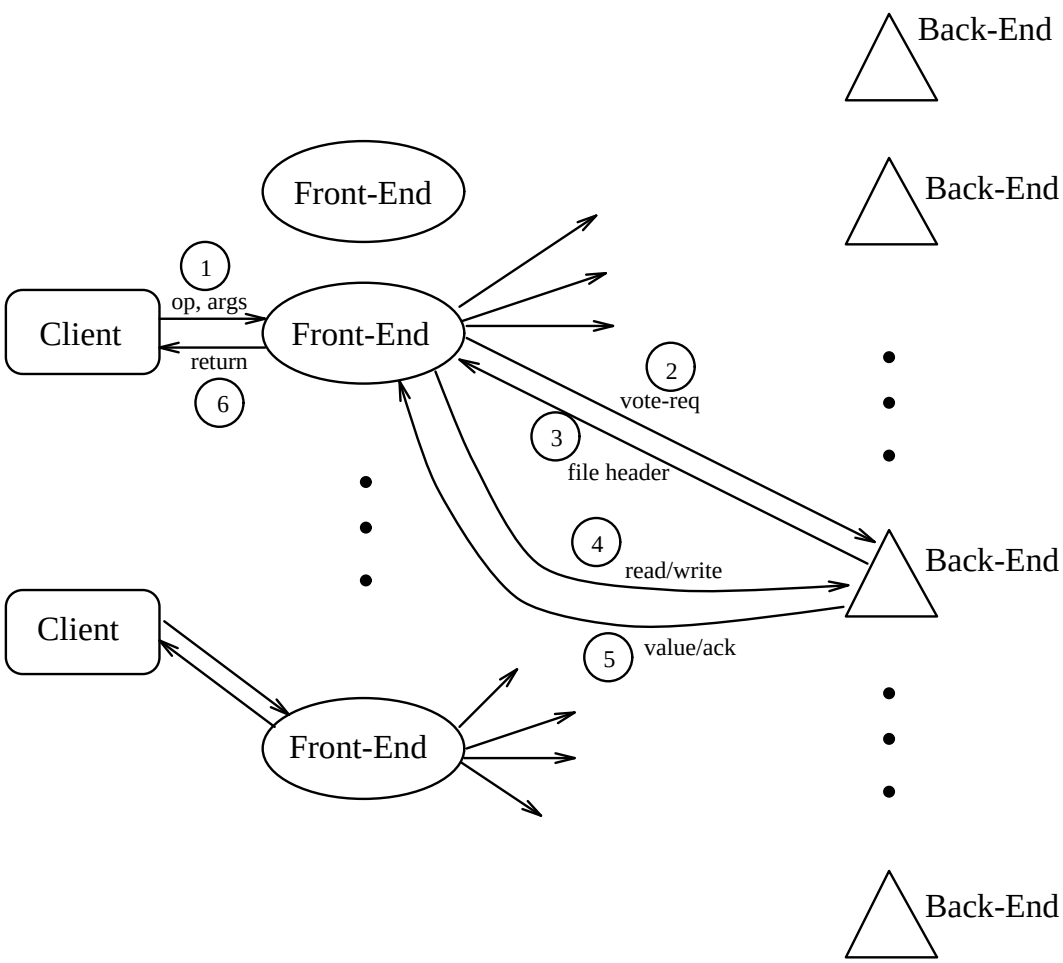

Figure 1: Organization of a reconfigurable composite server that supports replication and caching.

Each back-end keeps, on stable storage, a copy of some version of each of the files it handles, together with the associated version number and configuration. Front-ends encapsulate the replication protocol, and execute file read and update operations. Clients interact only with front-ends, which cache file configurations, versions, and contents in their volatile memory.

1.1 Argus

Argus, an integrated programming language and system, supports the implementation and execution of distributed programs [10]. The key constructs are *guardians* and *actions* (commonly called transactions in the literature). A *guardian* is a multi-threaded process that encapsulates a resource such as an object and controls access to it. Access to the guardian is provided through *handlers*, which are remote procedure calls implemented using subactions. A guardian's state consists of two parts: a *stable* state that survives failures like process crashes and machine crashes, and a *volatile* state that does not. After a crash, the volatile storage can be reconstructed from the stable storage by executing some application dependent code. A guardian executes at a single site but more than one guardian can run at a site. Front-ends and back-ends are implemented in BURDS as guardians.

Actions are indivisible and recoverable and hence, help to mask the effects of concurrency and failures for atomically typed objects [14]. Actions in Argus can be nested, *i.e.*, an action may have *subactions* and each of these subactions may do some computation and commit or abort without altering the parent action. However, an abort of a parent action undoes all the modifications to the state made by any of its subactions – even those that are committed. The stable state of a guardian is modified only if the top-level action (called *topaction*) commits. Concurrency is controlled by strict two phase locking. Failure atomicity of actions is provided by creating a new version of an object when it is written and replacing the old version by the new version when the topaction commits.

2 Replication, Reconfiguration and Caching

2.1 Weighted Voting with Reconfiguration

A file is replicated at several back-ends and a front-end serves as the interface between a client and the back-ends, hiding the replication from the client. Each back-end is assigned a certain number of votes to be collected by the front-end when processing a *read*() or *update*() operation (see figure 2). An operation can proceed only if the number of votes collected meets or exceeds a preset quorum[2] for that operation. The number of back-ends, the quorum assignment, and the location of the back-ends for a file define its *configuration*. The quorum assignment should satisfy the condition that the sum of the read quorum and the write quorum should be greater than the sum of all the votes assigned to the back-ends. Reconfiguration is achieved by writing the new configuration to a write quorum of the old configuration and also to a write quorum of the new configuration [6].

2.2 Cache Maintenance

For each file that it deals with, a front-end maintains a cache of both the file header and its contents (see figure 2). If found to be obsolete, the cache is updated during the *read*() and *update*() handler calls. The validation of the configuration cache is shown in figure 2 and the validation of the data cache is similar.

[2]The word "quorum" is used to denote both the set of back-ends whose votes suffice to complete an operation, and the *threshold* that the sum of the votes must exceed in order for the voting back-ends to be considered a quorum. Here we use it in the latter sense.

Header						Data
back_end list	read quorum	write quorum	votes	configuration number	version number	contents of file

```
front_end = guardian is ...
                handles read, update, reconfigure, ...
header_type = atomic_record[rep_list: array[back_end_server], r_quorum, w_quorum,
                votes, config_num, version:int]
file_type = atomic_record[header: header_type, contents: atomic_array[char]]
cache: file_type            % initialized when front_end guardian is created.
% read, write, etc. handler declarations ommitted.
% Validate_cache collects votes, and iteratively updates header cache if conifugration is obsolete (after that,
% the file contents cache has to be validated separately).
validate_cache = proc() returns array[header_type] signals(no_quorum)
    count: mutex[int] := mutex[int]$create(0)
    % Remember the version number to check later for currency of cached file contents:
    tmp: int := cache.header.version
    best_header := dummy header with -1 configuration number
        while cache.header.config_num > best_header.config_num do
            count.value := 0 % refers to the value of the mutex
            coenter action foreach i: int in int$from_to(1, num_back_ends)
                    header_list[i] := cache.header.back_end_list[i].read_header()
                        except when failure, unavailable(*): leave % leave causes this arm
                                % of the coenter to terminate without affecting the other arms
                        end % except
                    seize count do
                        count.value := count.value + header_list[i].votes
                        if (count.value ≥ cache.header.r_quorum)
                                then exit done % Exit coenter, terminating siblings
                        end % if
                    end % seize
            end % coenter
                except when done:
                        best_header := header_list[*] with largest configuration number
                        continue % Go back to while statement
                end % except
            signal no_quorum % Exits from procedure validate_cache
        end % while
end validate_cache
```

Figure 2: File (and cache) structure and an excerpt from the front-end in Argus pseudo-code.

3 Performance Results

Our preliminary performance data reflect a serial load using two MicroVax IIs and one MicroVax III running UNIX 4.3 BSD and Argus. The response time for transactions, consisting of 5 updates and 5 reads each, was measured. We repeated every measurement at least three times, then selected the minimum of these in order to reduce the effects of unrelated asynchronous UNIX and Argus computations. In other words, the numbers we report below represent the peak rather than steady state performance of our system. We varied the degree of replication (number of back-ends) from one to three, the quorum assignment, the process placements, the file size (1–1000 characters), and the client request size[3] (1–1000 characters). All of the numbers we have reflect a cache hit ratio of 1. We arranged the process placements so as to mimize the use of the MicroVax III, in order not to bias our comparisons by this machine's higher speed.

As background, one handler call to read (update) one character from a one character replicated file takes 50–120msec (100–250msec), depending primarily on the number of copies (back-ends)—as expected. The corresponding numbers for a transaction (topaction) with a single handler call are 100–200msec for reads and 150–330msec for updates.

Interestingly enough, varying the quorum assignment for the case of three back-ends increases the response time by a factor of no more than 1.10–1.25 regardless of the file size and request size. To shed further light on this observation, we plan to vary the operation mix and widen our choices of quorum assignments by increasing the number of back-ends beyond the current maximum of three, which was determined by the accessibility of machines running Argus.

By varying the placement of the front-end, back-end and client processes relative to each other, we could observe the effect of moving the front-end away from the client's site, thus introducing an extra round trip delay in the transaction. Instead of the anticipated increase in response time, we record a slight *decrease* in response time in almost all of the cases involving small request sizes, *e.g.*, one character. A possible explanation is that round trip message delay does not always dominate the cost of IPC (inter-process communication). Other significant factors include the cost of context switching (on lightly loaded machines, there may be less context switching if the two processes are on different machines), and the CPU overhead to prepare messages for transmission. Further evidence in support of this explanation is provided by comparing two placements under a degree of replication of two. In the first placement, the client and front-end reside on the same site, while in the other, the client and one back-end are together, but the front-end moves to a separate site. The difference between these two placements does not affect the transaction response time even for the largest file and request sizes tested (1000 characters). This result also is unexpected; why should replacing the front-end by a back-end on the same machine as the client mitigate the effect of the extra round trip message delay incurred?

We recorded the effect of message sizes on response time. As we change the request size from 1–1000 characters (for a file size of 1000 characters), the response time increases by a factor of 2.0–2.5, and similarly, by a factor of 1.4–1.8 for an increase in the file size from 1–1000 characters (for a request size of 1 character). We conclude that message sizes under Argus must be considered a significant contributor to overall performance, in addition to the number of messages and the number of rounds. This holds even for relatively small messages containing hundreds of characters.

Going from one back-end to two and then three back-ends resulted in a slow down by a factor

[3]The size of the substring that the client requests be read (written) from (to) the file.

of 1.5 and 1.9 respectively, for transactions with ten reads on a file whose read quorum is one. This can be accounted for only by the extra vote request handler calls (*i.e.*, read_header()) made by the validate_cache() procedure, and by the cost of aborting the ones that do not participate in the quorum. We suspect that almost all of the extra response time for higher degrees of replication for the same operation can be attributed to the same cause, because the differences we observed were all in the same range as the above case. Hence, we conjecture that under similar circumstances as ours, voting protocols will outperform other types of protocols in update response time. Our reason for this is that, unlike other replication protocols, such as primary copy [1] and available copies [2], voting protocols do not require that all operational copies of an object learn of an update. Hence, the number of handler calls (RPC's) needed for an update can be made smaller.

4 Evaluation of Argus

A subaction whose parent wishes to commit regardless of its fate, must nevertheless be aborted under Argus before the parent can proceed. We use concurrent subactions to collect votes from, and to write to, multiple back-ends. When a quorum of these subactions commits, the front-end is forced to abort the remaining subactions unnecessarily before it can continue. However, these subactions should be detached from their parent and allowed to commit if they can, especially in the case of updates when all back-ends are up. This maximizes the number of up-to-date back-ends. However, such an enhancement to Argus can result in an increase in the topaction commit overhead and complexity because these detached subactions must become optional participants in the two phase commit.

Our programs would benefit if Argus possessed a multicast facility for handler calls that incurs a constant overhead for issuing multiple identical handler calls to different recipients.

One advantage of Argus over the Camelot distributed transaction facility [13] is that data types can be declared *atomic* with respect to concurrency control, but not recovery. This gives us both concurrency control and proper abort semantics for the shared cache without incurring the cost of stable storage. Mutual exclusion does not suffice, because it does not undo the effects of an aborted action that wrote to the cache, whether the action is aborted while it is holding the mutex lock or after it releases it.

5 Related Work

Satyanarayanan and Agrawal [12] have implemented weighted voting without reconfiguration for another purpose[4]. Using remote procedure calls on Sun 3 machines running SunOS, they obtained a response time of 100msec or more for transactions that read or update 3 to 6 small objects. Note that these figures do not reflect the effect of disk I/O[5], since they stored data in main memory only. By comparison, the response times we observed were about three times larger. We attribute these differences to the higher functionality provided by our system and environment, such as stable storage.

The Coda replicated file system developed at Carnegie Mellon [11] provides for *non-transactional*

[4] To compare weighted voting with general quorum consensus [9] with and without two phase gossip [7].

[5] Except for virtual memory swapping.

file replication using weighted voting. However, the voting occurs only on file open and the updating of the back ends happens on file close. The performance data reported in [11] lead to a response time of about 370msec to read a file of about 3000 characters that is replicated three times. Our preliminary results indicate that we can read a 3000 characters file in about 700msec.

Other projects that implemented and measured weighted voting include the Seth project at Purdue University [3], and Bloch's replicated sparse memories [4].

6 Discussion

In section 3, we reported our measurements of response time at peak performance. However, system throughput (inverse of steady state response time) is an equally important measure of performance. In the absence of failures, replicating an object n times may cause at least a factor of n loss in stable storage write throughput, because every write must be performed n times. One advantage of voting algorithms is that the loss factor for write throughput is governed by the size of the write quorum, and not by the number of back-ends.

Our results also indicate that message sizes contribute significantly to overall transaction response times. Consequently, we believe that a data representation method, such as event-based data representation [7], that helps minimize message sizes for large objects may have strong performance advantages. Such a representation method requires more complex (and permits more flexible) replication, caching, and reconfiguration protocols.

For all of the above reasons, the members of project BURDS are presently working on implementing a voting protocol that operates on an event-based representation of replicated typed data. We expect to obtain initial results towards the end of this Fall.

Acknowledgements: We thank Barbara Liskov and the Argus group (especially Dorothy Curtis) at MIT for their help and for kindly making available to us some of their machines running Argus.

References

[1] P. Alsberg, G. Belford, J. Day, and E. Grapa. Mutli-copy resiliency techniques. CAC Document 202, Center for Advanced Computation, University of Illinois, Urbana, Illinois, May 1976.

[2] P.A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. on Database Systems*, 9(4), Dec. 1984.

[3] B. Bhargava, A. Helal, and J. Srinivasan. SETH: A quorum-based replicated database system for experimentation with failures. Technical Report CSD-TR-784, Purdue University, Computer Sciences Department, June 1988.

[4] Joshua J. Bloch. *A practical approach to replication of abstract data objects.* PhD thesis, Carnegie Mellon University, School of Computer Science, May 1990.

[5] D.K. Gifford. Weighted voting for replicated data. In *Proc. 7th ACM Symp. on Operating System Principles*, Dec. 1979.

[6] K.J. Goldman. Data replication in nested transaction systems. Master's thesis, M.I.T., Lab. for Computer Science, May 1987.

[7] A.A. Heddaya, M. Hsu, and W.E. Weihl. Two phase gossip: Managing distributed event histories. *Information Sciences*, 49(1,2,3):35–57, Oct./Nov./Dec. 1989. Special issue on databases.

[8] Abdelsalam Heddaya. View-based reconfiguration of replicated data. Technical Report BU-CS-90-001, Boston University, Computer Science Dept., 1990.

[9] M.P. Herlihy. A quorum-consensus replication method. *ACM Trans. on Computer Systems*, 4(1):32–53, Feb. 1986.

[10] B. Liskov. Distributed programming in Argus. *Comm. ACM*, 31(3):300–313, Mar. 1988.

[11] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere. Coda: a highly available file system for a distributed workstation environment. Technical Report CMU-CS-89-165, Carnegie Mellon University, School of Computer Science, Nov. 1989. Revised version.

[12] O.T. Satyanarayanan and Divyakant Agrawal. Prototype system for improving availability and efficiency of distributed database queries. Technical Report TRC89-33, University of California, Santa Barbara, Dec. 1989.

[13] A.Z. Spector and K.R. Swedlow (eds.). Guide to the Camelot distributed transaction facility. Technical report, Computer Science Dept., Carnegie Mellon University, May 1988. New edition to be published as a book.

[14] W. Weihl and B. Liskov. Implementation of resilient atomic data types. *ACM Trans. on Programming Languages and Systems*, 7(2):244–269, Apr. 1985.