Byzantine Fault Tolerance Can Be Fast

Miguel Castro
Microsoft Research Ltd.
1 Guildhall St., Cambridge CB2 3NH, UK
mcastro@microsoft.com

Barbara Liskov
MIT Laboratory for Computer Science
545 Technology Sq., Cambridge, MA 02139, USA
liskov@lcs.mit.edu

Abstract

Byzantine fault tolerance is important because it can be used to implement highly-available systems that tolerate arbitrary behavior from faulty components. This paper presents a detailed performance evaluation of BFT, a state-machine replication algorithm that tolerates Byzantine faults in asynchronous systems. Our results contradict the common belief that Byzantine fault tolerance is too slow to be used in practice — BFT performs well so that it can be used to implement real systems. We implemented a replicated NFS file system using BFT that performs 2% faster to 24% slower than production implementations of the NFS protocol that are not fault-tolerant.

1. Introduction

We are increasingly dependent on services provided by computer systems. We would like these systems to be *highly-available*: they should provide correct service without interruptions. There is an extensive body of research on replication techniques to achieve high availability but most assume that nodes fail by stopping or by omitting some steps. We believe that these assumptions are not likely to hold in the future. For example, malicious attacks are increasingly common and can cause faulty nodes to behave arbitrarily.

Byzantine fault tolerance techniques can be used to implement highly-available systems that tolerate arbitrary behavior from faulty replicas. The problem is that it is widely believed that these techniques are too slow to be used in practice. This paper presents experimental results showing that it is possible to implement highly-available systems that tolerate Byzantine faults and perform well.

We developed a *state machine* replication [14] algorithm, BFT, that tolerates Byzantine faults. BFT has been implemented as a generic program library with a simple interface and we used the library to implement the first Byzantine-fault-tolerant NFS file system, BFS. The algorithm, the library, and BFS were described elsewhere [3, 4, 2]. This paper presents a performance evaluation of BFT and BFS.

BFT performs well mostly because it uses symmetric cryptography to authenticate messages. Public-key cryptography, which was the major bottleneck in previous systems [12, 9], is used only to exchange the symmetric keys. Additionally, BFT incorporates several important optimizations that reduce the size and number of messages used by the protocol.

We present results of several micro-benchmarks that characterize the performance of the BFT library in a service-independent way, and evaluate the impact of each of the performance optimizations. Additionally, we present performance results for BFS on the modified Andrew benchmark [11] and PostMark [8]. These results show that BFS performs 2% faster to 24% slower than production implementations of the NFS protocol that are not replicated.

There is little published work on the performance of Byzantine fault tolerance. There is an evaluation of the Rampart toolkit [12], and performance studies of three services Ω [13], e-Vault [7], and COCA [15]. It is hard to perform a direct comparison between these systems and the BFT library but it is clear that our library is significantly faster. Some performance numbers reported in this paper appeared in [4] but we expand on their analysis.

The rest of the paper is organized as follows. We start by describing the properties provided by the algorithm and its assumptions. Section 3 presents a brief overview of the algorithm and the performance optimizations. The performance evaluation is in Sections 4 and 5: they present micro-benchmark and file system benchmark results, respectively. Section 6 summarizes our results.

2. Properties and Assumptions

BFT can replicate any service that can be modeled as a deterministic state machine, i.e., the different replicas are required to produce the same sequence of results when they execute the same sequence of operations. Note that replicas do not need to run the same code [5].

We make no assumptions about the network that connects replicas and clients except that we assume eventual time bounds on message delays for liveness. We use a Byzantine failure model, i.e., faulty nodes may behave arbitrarily. But we assume that an attacker is computationally bound so that (with very high probability) it is unable to subvert the cryptographic techniques we use in the algorithm.

This research was supported by DARPA under contract F30602-98-1-0237 monitored by the Air Force Research Laboratory. This paper describes work done while the first author was at the MIT Laboratory for Computer Science.

BFT offers strong safety and liveness properties if less than $1/3$ of the replicas are faulty and regardless of the number of faulty clients. BFT provides linearizability [6] without relying on any synchrony assumption, and it guarantees that correct clients receive replies to their requests if delays are bounded eventually. In particular, BFT can guarantee safety in the presence of denial of service attacks whereas previous state-machine replication systems [12, 9] could not.

BFT can recover replicas proactively [4]. This allows BFT to offer safety and liveness even if all replicas fail provided less than $1/3$ of the replicas become faulty within a window of vulnerability. To simplify the presentation, we assume that there are $3f + 1$ replicas to enable the system to tolerate up to f faults.

There is little benefit in any form of replication if the replicas are likely to fail at the same time. We discuss techniques to fight this problem in [5, 2].

3. Algorithm

The algorithm is described in detail in [2]. Here, we provide only a brief overview to help understand the performance evaluation. The basic idea is simple. Clients send requests to execute operations and all non-faulty replicas execute the same operations in the same order. Since replicas are deterministic and start in the same state, all non-faulty replicas send replies with identical results for each operation. The client chooses the result that appears in at least $f + 1$ replies.

The hard problem is ensuring non-faulty replicas execute the same requests in the same order. BFT uses a combination of primary-backup and quorum replication techniques to order requests. Replicas move through a succession of numbered configurations called *views*. In a view, one replica is the *primary* and the others are *backups*. The primary picks the execution order by assigning a sequence number to each request. Since the primary may be faulty, the backups check the sequence numbers and trigger *view changes* to select a new primary when it appears that the current one has failed.

Figure 1 shows the operation of the algorithm in the normal case of no primary faults. In the figure, $< m >_{\mu_{ij}}$ denotes a message m from i to j with a *message authentication code* (MAC), and $< m >_{\alpha_i}$ is a message with a vector of MACs with an entry for each replica other than i. The algorithm also uses a cryptographic hash function D. Currently, we use UMAC32 [1] to compute MACs and MD5 to compute digests.

The figure illustrates an example where a client c sends a request m to execute an operation o with a timestamp t to the primary for the current view v (replica 0). The primary assigns a sequence number n to m and sends a pre-prepare message with the assignment to the backups. Each backup i accepts the assignment if it is in view v, and it has not assigned n to a different request in v. If i accepts the pre-prepare, it multicasts a prepare message to all other replicas signalling that it accepted the sequence number assignment. Then, each replica collects messages until it has a pre-prepare and $2f$ matching prepare messages for sequence number n, view v, and request m. When the replica has these messages, we say that it prepared the request. The protocol guarantees that it is not possible for correct replicas to prepare distinct requests with the same view and sequence number.

The algorithm uses an additional phase to ensure this ordering information is stored in a quorum to survive view changes. Each replica multicasts a commit message saying that it has prepared the request. Then each replica collects messages until it has $2f + 1$ commit messages for v and n from different replicas (including itself). We say that the request is committed when the replica has these messages.

Each replica executes operation o when m is committed and the replica has executed all requests with sequence numbers less than n. Then, the replicas send replies with the operation result r to the client. The reply message includes the current view number so that clients can track the current primary.

3.1. Optimizations

This section describes several optimizations that improve the performance during normal case operation while preserving the safety and liveness properties.

With the *digest replies* optimization, a client request designates a replica to send the result. This replica may be chosen randomly or using some other load balancing scheme. After the designated replica executes the request, it sends back a reply containing the result. The other replicas send back replies containing only the digest of the result. The client uses the digests to check the correctness of the result. If the client does not receive a correct result from the designated replica, it retransmits the request (as usual) requesting all replicas to send replies with the result.

The *tentative execution* optimization reduces the number of message delays for an operation invocation from five to four. Replicas execute requests *tentatively* as soon as: the request is prepared; their state reflects the execution of all requests with lower sequence number; and these requests have committed. After executing the request, the replicas send tentative replies to the client. Since replies are tentative, the client must wait for $2f + 1$ replies with the same result. This ensures that the request is prepared by a quorum and, therefore, it is guaranteed to commit eventually at non-faulty replicas.

It is possible to take advantage of tentative execution to eliminate commit messages without increasing latency: they can be piggybacked in the next pre-prepare or prepare message sent by a replica.

We also have a *read-only* optimization that reduces latency to a single round trip for operations that do not modify the service state. A client multicasts a read-only request to all replicas. The replicas execute the request immediately after checking that it is properly authenticated, and that the request is in fact read-only. A replica sends back a reply only after all requests it executed before the read-only request have committed. The client waits for $2f + 1$ replies with the same result. It may be unable to collect these if there are concurrent writes to data that affect the result. In

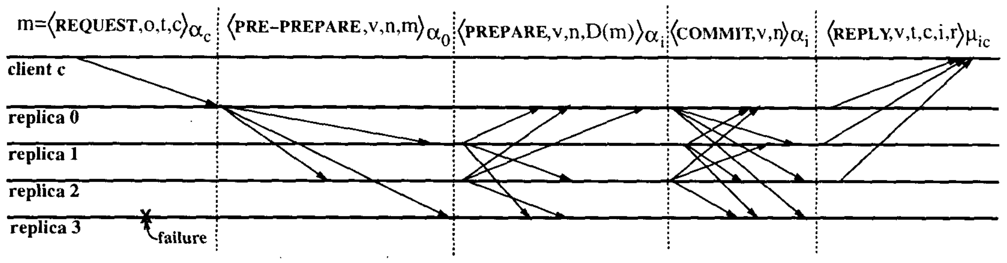

Figure 1. Normal Case Operation.

this case, it retransmits the request as a regular read-write request after its retransmission timer expires. The read-only optimization preserves linearizability provided clients obtain $2f + 1$ matching replies for both read-only and read-write operations.

Request batching reduces protocol overhead under load by starting a single instance of the protocol for a batch of requests. We use a sliding-window mechanism to bound the number of protocol instances that can run in parallel without increasing latency in an unloaded system. Let e be the sequence number of the last batch of requests executed by the primary and let p be the sequence number of the last pre-prepare sent by the primary. When the primary receives a request, it starts the protocol immediately unless $p \geq e+W$, where W is the *window size*. In the latter case, it queues the request. When requests execute, the window slides forward allowing queued requests to be processed. Then, the primary picks the first requests from the queue such that the sum of their sizes is below a constant bound; it assigns them a sequence number; and it sends them in a single pre-prepare message. The protocol proceeds exactly as it did for a single request except that replicas execute the batch of requests (in the order in which they were added to the pre-prepare message) and they send back separate replies for each request.

We modified the algorithm to use *separate request transmission*: requests whose size is greater than a threshold (currently 255 bytes) are not inlined in pre-prepare messages. Instead, the clients multicast these requests to all replicas; replicas authenticate the requests in parallel; and they buffer those that are authentic. The primary selects a batch of requests to include in a pre-prepare message but it only includes their digests in the message.

4. Micro-Benchmarks

This section presents results of micro-benchmarks designed to characterize the performance of the BFT library in a service-independent way, and to evaluate the impact of each performance optimization. The experiments were performed using the setup in Section 4.1. Sections 4.2 and 4.3 measure the latency and throughput of a simple replicated service using all the optimizations. The impact of the different optimizations is studied in Section 4.4. See [2] for a more detailed performance evaluation and description of the experimental setup.

4.1. Experimental Setup

The experiments ran on Dell Precision 410 workstations with a single 600 MHz Pentium III processor, 512 MB of memory, and a Quantum Atlas 10K 18WLS disk. All machines ran Linux 2.2.16-3 compiled without SMP support. The machines were connected by a 100 Mb/s switched Ethernet. The switch was an Extreme Networks Summit48 V4.1. Replicas and clients ran on different machines and all experiments ran on an isolated network.

The experiments compare the performance of two implementations of a simple service: one implementation, BFT, is replicated using the BFT library and the other, NO-REP, is not replicated and uses UDP directly for communication between the clients and the server. The simple service is really the skeleton of a real service: it has no state and the service operations receive arguments from the clients and return (zero-filled) results but they perform no computation. We performed experiments with different argument and result sizes for both read-only (RO) and read-write (RW) operations. It is important to note that this is a worst-case comparison; in real services, computation or I/O at the clients and servers would reduce the slowdown introduced by the BFT library (as shown in Section 5).

The results were obtained by timing a large number of invocations in at least three separate runs. We report the average of the three runs. The standard deviation was always below 10% of the reported value.

4.2. Latency

We measured the latency to invoke an operation with four replicas when the service is accessed by a single client. Figure 2 shows the latency to invoke the replicated service as the size of the operation result increases while keeping the argument size fixed at 8 B. It has one graph with elapsed times and another with the slowdown of BFT relative to NO-REP. We also ran experiments with varying argument sizes (see Figure 3) and obtained very similar results.

The library introduces a significant slowdown relative to NO-REP but the slowdown decreases quickly as the operation argument or result sizes increase. In both cases, the slowdown decreases till an asymptote of 1.26 [2]. The two major sources of overhead are digest computation (of requests and replies) and the additional communication due to the replication protocol. The cost of MAC computation is negligible.

The read-only optimization improves performance by eliminating the time to prepare the requests. This time does

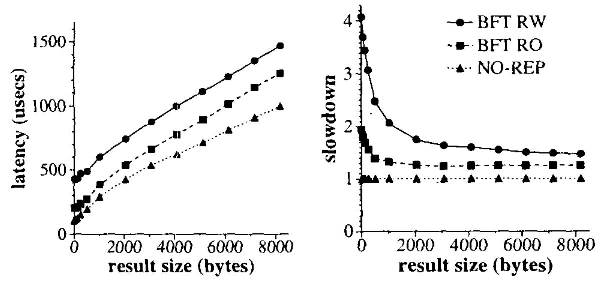

Figure 2. Latency with and without BFT.

not change as the argument or result size increases. Therefore, the speed up afforded by the read-only optimization decreases to zero as the argument or result size increases.

The experiments in Figure 2 ran in a configuration with four replicas, which can tolerate one fault. We believe this level of reliability will be sufficient for most applications but some may require more replicas. Figure 3 compares the latency to invoke the replicated service with four replicas ($f = 1$) and seven replicas ($f = 2$). In both configurations, all the replicas had a 600 MHz Pentium III processor and the client had a 700 MHz Pentium III processor.

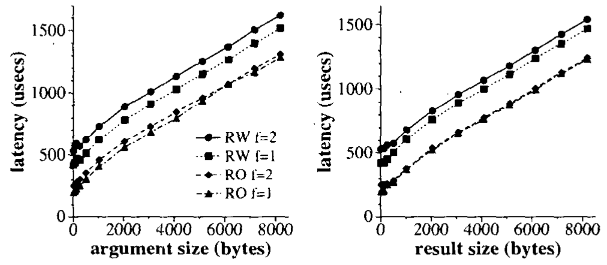

Figure 3. Latency with $f = 2$ and with $f = 1$.

The results show that the slowdown caused by increasing the number of replicas to seven is low. The maximum slowdown is 30% for the read-write operation and 26% for the read-only operation. Furthermore, the slowdown decreases quickly as the argument or result size increases.

4.3. Throughput

This section reports the result of experiments to measure the throughput of BFT and NO-REP as a function of the number of clients accessing the simple service. The client processes were evenly distributed over 5 client machines[1]. There were four replicas. We measured throughput for operations with different argument and result sizes. Each operation type is denoted by *a/b*, where *a* and *b* are the sizes of the argument and result in KB.

Figure 4 shows throughput results for operations 0/0, 0/4, and 4/0. The bottleneck in operation 0/0 is the server's CPU. BFT has lower throughput than NO-REP due to extra messages and cryptographic operations that increase the CPU load. The read-only optimization improves throughput by eliminating the cost of preparing the batch of requests. The throughput of the read-write operation improves as the number of clients increases because the cost of preparing the batch of requests is amortized over the size of the batch. Throughput saturates because the batch size is limited by how many requests can be inlined in a pre-prepare message.

BFT has better throughput than NO-REP for operation 0/4. The bottleneck for NO-REP is the link bandwidth that imposes an upper bound of 3000 operations per second. BFT achieves better throughput because of the digest-replies optimization: clients obtain the replies with the 4 KB result in parallel from different replicas. BFT achieves a maximum throughput of 6625 operations per second (26MB/s) for the read-write operation and 8987 operations per second (35 MB/s) with the read-only optimization. The bottleneck for BFT is the replicas' CPU.

The bottleneck in operation 4/0 for both NO-REP and BFT is the time to get the requests through the network, which imposes a bound of 3000 operations per second. NO-REP achieves a maximum throughput of 2921 operations per second while BFT achieves 11% less for read-write operations and 2% less with the read-only optimization. There are no points with more than 15 clients for NO-REP because of lost request messages; NO-REP uses UDP directly and does not retransmit requests.

4.4. Impact of Optimizations

The experiments in the previous sections show performance with all the optimizations for both read-write and read-only operations. This section analyses the performance impact of the other optimizations.

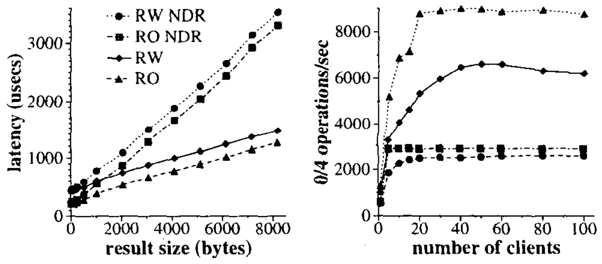

Figure 5. Digest Replies Optimization

Figure 5 compares the performance of BFT with and without the digest replies optimization. We called the version of BFT without the optimization BFT-NDR. The first graph measures latency as the size of the operation result increases with the argument size fixed at 8 B, and the second shows the throughput results for operation 0/4. We chose these experiments because the impact of the digest replies optimization increases with the result size.

The digest replies optimization reduces the latency to invoke operations with large results significantly. Furthermore, this speedup increases linearly with the number of replicas. Additionally, BFT achieves a throughput up to 3 times better than BFT-NDR. The bottleneck for BFT-NDR is the link bandwidth: it is limited to a maximum of at most 3000 operations per-second regardless of the number of replicas. The digest replies optimization enables bandwidth for sending replies to scale linearly with the number of replicas and it also reduces load on replicas' CPUs.

[1]Two client machines had 700 MHz PIIIs but were otherwise identical to the other machines.

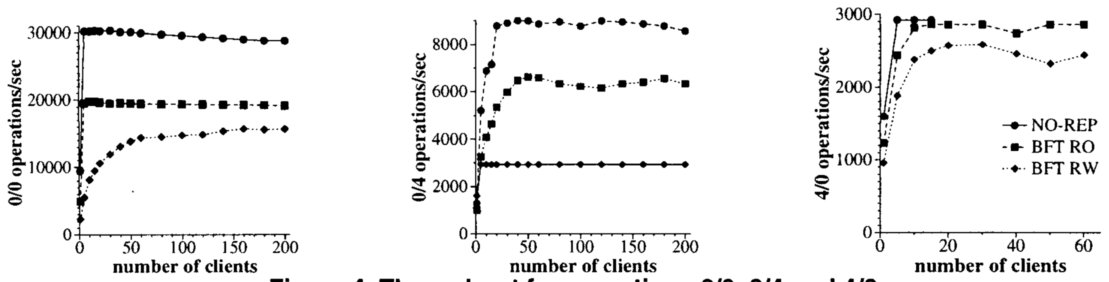

Figure 4. Throughput for operations 0/0, 0/4 and 4/0.

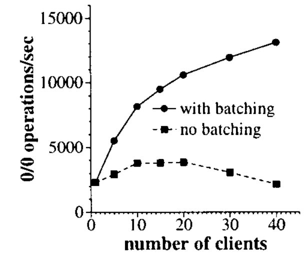

Figure 6. Request Batching Optimization.

Figure 6 compares throughput with and without request batching for read-write operation 0/0. The throughput without batching grows with the number of clients because the algorithm can process many requests in parallel. But the replicas' CPUs saturate for a small number of clients because processing each of these requests requires a full instance of the protocol. Our batching mechanism reduces both CPU and network overhead under load without increasing the latency to process requests in an unloaded system. Previous state machine replication systems that tolerate Byzantine faults [10, 9] have used batching techniques that impact latency significantly.

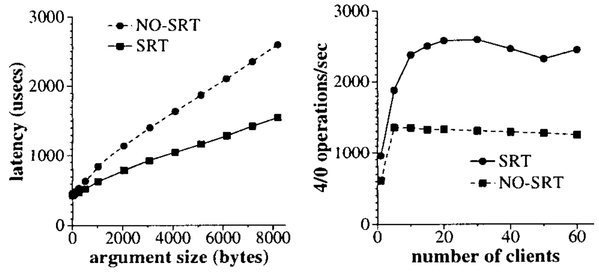

Figure 7. Separate Request Transmission.

Figure 7 compares performance with and without the separate request transmission optimization. The first graph shows latency for varying argument sizes, and the second shows throughput for read-write operation 4/0. We labeled the version of BFT without the optimization BFT-NO-SRT. Separating request transmission reduces latency by up to 40% because the request is sent only once and the primary and the backups compute the request's digest in parallel. The other benefit of separate request transmission is improved throughput for large requests because it enables more requests per batch.

The impact of the tentative execution optimization on throughput is insignificant. The optimization reduces latency by up to 27% with small argument and result sizes but its benefit decreases quickly when sizes increase.

Piggybacking commits has a negligible impact on latency because the commit phase of the protocol is performed in the background (thanks to tentative execution of requests). It also has a small impact on throughput except when the number of concurrent clients accessing the service is small. For example, it improves the throughput of operation 0/0 by 33% with 5 clients but only by 3% with 200 clients. The benefit decreases because batching amortizes the cost of processing the commit messages over the batch size. This optimization is the only one that is not currently part of the BFT library; we only wrote code for the normal case.

5. File System Benchmarks

We compared the performance of BFS with two other implementations of NFS: NO-REP, which is identical to BFS except that it is not replicated, and NFS-STD, which is the NFS V2 implementation in Linux with Ext2fs at the server. The first comparison allows us to evaluate the overhead of the BFT library accurately within an implementation of a real service. The second comparison shows that BFS is practical: it performs similarly to NFS-STD, which is used daily by many users but is not fault-tolerant.

5.1. Experimental Setup

The experiments to evaluate BFS used the setup described in Section 4.1. They ran two well-known file system benchmarks: Andrew [11] and PostMark [8]. There were no view changes or proactive recoveries in these experiments.

The Andrew benchmark emulates a software development workload. We scaled up the benchmark by creating n copies of the source tree in the first two phases and operating on all copies in the remaining phases [4]. We ran a version of Andrew with n equal to 100, Andrew100, and another with n equal to 500, Andrew500. They generate approximately 200 MB and 1 GB of data; Andrew100 fits in memory at both the client and the replicas but Andrew500 does not.

PostMark [8] models the load on Internet Service Providers. We configured PostMark with an initial pool of 10000 files with sizes between 512 bytes and 16 Kbytes. The benchmark ran 100000 transactions.

For all benchmarks and NFS implementations, the actual benchmark code ran at the client workstation using the standard NFS client implementation in the Linux kernel with the same `mount` options. The most relevant of these options: UDP transport, 4 KB buffers, write-back client caching, and attribute caching. BFS and NO-REP do not to maintain the time-last-accessed attribute. We report the mean of at least three runs of each benchmark and the standard deviation was always below 2% of the reported value.

5.2. Experiments

Figure 8 presents results for Andrew100 and Andrew500 in a configuration with four replicas and one client machine. The comparison between BFS and NO-REP shows that the overhead of Byzantine fault tolerance is low for this service — BFS takes only 14% more time to run Andrew100 and 22% more time to run Andrew500. This slowdown is smaller than the one measured with the micro-benchmarks because the client spends a significant fraction of the elapsed time computing between operations, and operations at the server perform some computation. Additionally, there are a significant number of disk writes at the server in Andrew500.

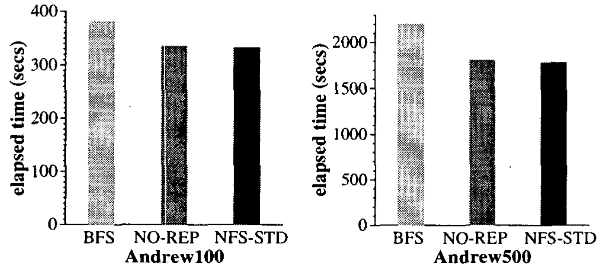

Figure 8. Modified Andrew.

The comparison with NFS-STD shows that BFS can be used in practice — it takes only 15% longer to complete Andrew100 and 24% longer to complete Andrew500. The performance difference would be smaller if Linux implemented NFS correctly. For example, the results in [2] show that BFS is 2% faster than the NFS implementation in Digital Unix, which implements the correct semantics. The implementation of NFS on Linux does not ensure stability of modified data and meta-data before replying to the client (as required by the NFS protocol), whereas BFS ensures stability through replication.

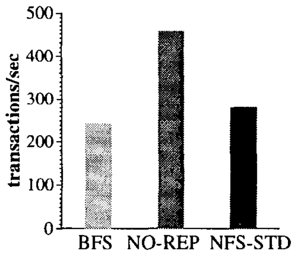

Figure 9. PostMark.

The overhead of Byzantine fault tolerance is higher in PostMark: BFS's throughput is 47% lower than NO-REP's. This is explained by a reduction on the computation time at the client relative to Andrew. What is interesting is that BFS's throughput is only 13% lower than NFS-STD's. The higher overhead is offset by an increase in the number of disk accesses performed by NFS-STD in this workload.

6. Conclusions

Byzantine-fault-tolerant replication can be used to build highly-available systems that can tolerate even malicious behavior from faulty replicas. But previous work on Byzantine fault tolerance has failed to produce solutions that perform well. This paper presented a detailed performance evaluation of the BFT library, a replication toolkit to build systems that tolerate Byzantine faults. Our results show that services implemented with the library perform well even when compared with unreplicated implementations that are not fault-tolerant.

Acknowledgements

We thank the anonymous reviewers and Lorenzo Alvisi for their comments on drafts of this paper.

References

[1] J. Black et al. UMAC: Fast and Secure Message Authentication. In *Advances in Cryptology - CRYPTO*, 1999.

[2] M. Castro. Practical Byzantine Fault Tolerance. Technical Report TR-817, PhD thesis, MIT Lab. for Computer Science, 2001.

[3] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *USENIX Symposium on Operating Systems Design and Implementation*, 1999.

[4] M. Castro and B. Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *USENIX Symposium on Operating Systems Design and Implementation*, 2000.

[5] M. Castro, R. Rodrigues, and B. Liskov. Using Abstraction to Improve Fault Tolerance. Submitted for publication, 2001.

[6] M. Herlihy and J. Wing. Axioms for Concurrent Objects. In *ACM Symposium on Principles of Programming Languages*, 1987.

[7] A. Iyengar et al. Design and Implementation of a Secure Distributed Data Repository. In *IFIP International Information Security Conference*, 1998.

[8] J. Katcher. PostMark: A New File System Benhmark. Technical Report TR-3022, Network Appliance, 1997.

[9] K. Kihlstrom, L. Moser, and P. Melliar-Smith. The SecureRing Protocols for Securing Group Communication. In *Hawaii International Conference on System Sciences*, 1998.

[10] D. Malkhi and M. Reiter. A high-throughput secure reliable multicast protocol. In *Computer Security Foundations Workshop*, 1996.

[11] J. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *USENIX Summer Conference*, 1990.

[12] M. Reiter. The Rampart toolkit for building high-integrity services. *Theory and Practice in Distributed Systems (LNCS 938)*, 1995.

[13] M. Reiter et al. The Ω Key Management Service. In *ACM Conference on Computer and Communications Security*, 1996.

[14] F. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4), 1990.

[15] L. Zhou, F. Schneider, and R. Renesse. COCA: A Secure Distributed On-line Certification Authority. Technical Report TR 2000-1828, C.S. Department, Cornell University, 2000.