BASE: Using Abstraction to Improve Fault Tolerance

Rodrigo Rodrigues[†], Miguel Castro[§], and Barbara Liskov[†]
[†]MIT Laboratory for Computer Science
200 Technology Sq., Cambridge MA 02139, USA
[§]Microsoft Research Ltd.
1 Guildhall St., Cambridge CB2 3NH, UK
rodrigo@lcs.mit.edu, mcastro@microsoft.com, liskov@lcs.mit.edu

ABSTRACT

Software errors are a major cause of outages and they are increasingly exploited in malicious attacks. Byzantine fault tolerance allows replicated systems to mask some software errors but it is expensive to deploy. This paper describes a replication technique, BASE, which uses abstraction to reduce the cost of Byzantine fault tolerance and to improve its ability to mask software errors. BASE reduces cost because it enables reuse of off-the-shelf service implementations. It improves availability because each replica can be repaired periodically using an abstract view of the state stored by correct replicas, and because each replica can run distinct or non-deterministic service implementations, which reduces the probability of common mode failures. We built an NFS service where each replica can run a different off-the-shelf file system implementation, and an object-oriented database where the replicas ran the same, non-deterministic implementation. These examples suggest that our technique can be used in practice — in both cases, the implementation required only a modest amount of new code, and our performance results indicate that the replicated services perform comparably to the implementations that they reuse.

1. INTRODUCTION

There is a growing demand for highly-available systems that provide correct service without interruptions. These systems must tolerate software errors because these are a major cause of outages [13]. Furthermore, there is an increasing number of malicious attacks that exploit software errors to gain control or deny access to systems that provide important services.

This paper proposes a replication technique, BASE, that combines Byzantine fault tolerance [31] with work on data abstraction [20]. Byzantine fault tolerance allows a replicated service to tolerate arbitrary behavior from faulty replicas, e.g., behavior caused by a software bug or an attack. Abstraction hides implementation details to enable the reuse of off-the-shelf implementations of important services (e.g., file systems, databases, or HTTP daemons) and to improve the ability to mask software errors.

We extended the BFT library [7, 8] to implement BASE. (BASE is an acronym for BFT with Abstract Specification Encapsulation.) The original BFT library provides Byzantine fault tolerance with good performance and strong correctness guarantees if no more than 1/3 of the replicas fail within a small window of vulnerability. However, it requires all replicas to run the same service implementation and to update their state in a deterministic way. Therefore, it cannot tolerate deterministic software errors that cause all replicas to fail concurrently and it complicates reuse of existing service implementations because it requires extensive modifications to ensure identical values for the state of each replica.

The BASE library and methodology described in this paper correct these problems — they enable replicas to run different or non-deterministic implementations. The methodology is based on the concepts of *abstract specification* and *abstraction function* from work on data abstraction [20]. We start by defining a common *abstract specification* for the service, which specifies an *abstract state* and describes how each operation manipulates the state. Then we implement a *conformance wrapper* for each distinct implementation to make it behave according to the common specification. The last step is to implement an *abstraction function* (and one of its inverses) to map from the concrete state of each implementation to the common abstract state (and vice versa).

The methodology offers several important advantages.

Reuse of existing code. BASE implements a form of state machine replication [17, 35], which allows replication of services that perform arbitrary computations, but requires determinism: all replicas must produce the same sequence of results when they process the same sequence of operations. Most off-the-shelf implementations of services fail to satisfy this condition. For example, many implementations produce timestamps by reading local clocks, which can cause the states of replicas to diverge. The conformance wrapper and the abstract state conversions enable the reuse of existing implementations without modifications. Furthermore, these implementations can be non-deterministic, which reduces the probability of common mode failures.

Software Rejuvenation through proactive recovery.

This research was partially supported by DARPA under contract F30602-98-1-0237 monitored by the Air Force Research Laboratory. Rodrigo Rodrigues was partially supported by a Praxis XXI fellowship.

It has been observed [16] that there is a correlation between the length of time software runs and the probability that it fails. BASE combines proactive recovery [8] with abstraction to counter this problem. Replicas are recovered periodically even if there is no reason to suspect they are faulty. Recoveries are staggered so that the service remains available during rejuvenation to enable frequent recoveries. When a replica is recovered, it is rebooted and restarted from a clean state. Then it is brought up to date using a correct copy of the abstract state that is obtained from the group of replicas. Abstraction may improve availability by hiding corrupt concrete states, and it enables proactive recovery when replicas do not run the same code or run code that is non-deterministic.

Opportunistic N-version programming. Replication is not useful when there is a strong positive correlation between the failure probabilities of the different replicas, e.g., deterministic software bugs cause all replicas to fail at the same time when they run the same code. N-version programming [9] exploits design diversity to reduce the probability of correlated failures, but it has several problems [13]: it increases development and maintenance costs by a factor of N or more, adds unacceptable time delays to the implementation, and does not provide a mechanism to repair faulty replicas.

BASE enables an opportunistic form of N-version programming by allowing us to take advantage of distinct, off-the-shelf implementations of common services. This approach overcomes the defects mentioned above: it eliminates the high development and maintenance costs of N-version programming, and also the long time-to-market. Additionally, we can repair faulty replicas by transferring an encoding of the common abstract state from correct replicas.

Opportunistic N-version programming is a viable option for many common services, e.g., relational databases, HTTP daemons, file systems, and operating systems. In all these cases, competition has led to four or more distinct implementations that were developed and are maintained separately but have similar (although not identical) functionality. Since each off-the-shelf implementation is sold to a large number of customers, the vendors can amortize the cost of producing a high quality implementation.

Furthermore, the existence of standard protocols that provide identical interfaces to different implementations, e.g., ODBC [11] and NFS [27], simplifies our technique and keeps the cost of writing the conformance wrappers and state conversion functions low. We can also leverage the effort towards standardizing data representations using XML.

The paper explains the methodology by giving two examples, a replicated file service where replicas run different operating systems and file systems, and a replicated object-oriented database, where the replicas run the same implementation but the implementation is non-deterministic. The paper also provides an evaluation of the methodology based on these examples; we evaluate the complexity of the conformance wrapper and state conversion functions and the overhead they introduce.

The remainder of the paper is organized as follows. Section 2 describes our methodology and the BASE library. Section 3 explains how we applied the methodology to build the replicated file system and object-oriented database. We evaluate our technique in Section 4. Section 5 discusses related work and Section 6 presents our conclusions.

2. THE BASE TECHNIQUE

This section provides an overview of our replication technique. It starts by describing the methodology that we use to build a replicated system from existing service implementations. It ends with a description of the BASE library.

2.1 Methodology

The goal is to build a replicated system by reusing a set of off-the-shelf implementations, $I_1, ..., I_n$, of some service. Ideally, we would like n to equal the number of replicas so that each replica can run a different implementation to reduce the probability of simultaneous failures. But the technique is useful even with a single implementation.

Although off-the-shelf implementations of the same service offer roughly the same functionality, they behave differently: they implement different specifications, $S_1, ..., S_n$, using different representations of the service state. Even the behavior of different replicas that run the same implementation may be different when the specification they implement is not strong enough to ensure deterministic behavior. For example, the NFS specification [27] allows implementations to choose the value of file handles arbitrarily.

BASE, like any form of state machine replication, requires determinism: replicas must produce the same sequence of results when they execute the same sequence of operations. We achieve determinism by defining a *common abstract specification*, S, for the service that is strong enough to ensure deterministic behavior. This specification defines the abstract state, an initial state value, and the behavior of each service operation.

The specification is defined without knowledge of the internals of each implementation. It is sufficient to treat them as black boxes, which is important to enable the use of existing implementations. Additionally, the abstract state captures only what is visible to the client rather than mimicking what is common in the concrete states of the different implementations. This simplifies the abstract state and improves the effectiveness of our software rejuvenation technique.

The next step, is to implement *conformance wrappers*, $C_1, ..., C_n$, for each of $I_1, ..., I_n$. The conformance wrappers implement the common specification S. The implementation of each wrapper C_i is a veneer that invokes the operations offered by I_i to implement the operations in S; in implementing these operations this veneer makes use of a *conformance representation* that stores whatever additional information is needed to allow the translation from the concrete behavior of the implementation to the abstract behavior. The conformance wrapper also implements some additional methods that allow a replica to be shutdown and then restarted without loss of information.

The final step is to implement the *abstraction function* and one of its inverses. These functions allow state transfer among the replicas. State transfer is used to repair faulty replicas, and also to bring slow replicas up-to-date when messages they are missing have been garbage collected. For state transfer to work, replicas must agree on the value of the state of the service after executing a sequence of operations; they will not agree on the value of the concrete state but our methodology ensures that they will agree on the value of the abstract state. The abstraction function is used to convert the concrete state stored by a replica into the abstract state, which is transferred to another replica. The receiving replica uses its inverse abstraction function to convert the abstract state into its own concrete state representation.

To enable efficient state transfer between replicas, the abstract state must be defined as an array of objects. The array has a fixed maximum size, but the objects it contains can vary in size. We explain how this representation enables efficient state transfer in Section 2.2.

2.1.1 Applicability

In theory, the methodology can be used to build a replicated service from any set of existing implementations of any service. But sometimes this may not be practical because of the following three problems.

Undocumented behavior. To apply the methodology, we need to understand and model the behavior of each service implementation. We do not need to model low level implementation details but only the behavior that can be observed by the clients of that implementation. We believe that the behavior of most software is well documented at this level, and we can use black box testing to understand small omissions in the documentation and small deviations from documented behavior. Implementations whose behavior we cannot model are unlikely to be of much use. But it may be possible to remove operations whose behavior is not well documented from the abstract specification, or to implement these operations entirely in the conformance wrapper.

Very different behavior. If the implementations used to build the service behave very differently, any common abstract specification will deviate significantly from the behavior of some implementations. Theoretically, it is possible to write arbitrarily complex conformance wrappers and state conversion functions to bridge the gap between the behavior of the different implementations and the common abstract specification. In the worst case, we could implement the entire abstract specification in the wrapper code. But in practice this is undesirable because it is expensive to write complex wrappers, and complex wrappers are more likely to introduce new bugs. Therefore, it is important to use a set of implementations with similar behavior.

Overly narrow interfaces. The external interface of some implementations may not allow the wrapping code to read or write data that has an impact on the behavior observed by the client. There are three options in this case. First, the data can be shadowed in the conformance wrapper. This is practical if it is a small amount of data that is simple to maintain. Second, it may be possible to change the abstract specification such that this data has no impact on the behavior observed by the client. Third, it may be possible to gain access to internal APIs that avoid the problem.

There is an important trend that avoids these problems and, therefore, makes it easier to apply the methodology. Market forces pressure vendors to offer interfaces that are compliant with standard specifications for interoperability, e.g., ODBC [11]. Usually, a standard specification S' cannot be used as the common specification S because it is too weak to ensure deterministic behavior. But it can be used as a basis for S and, because S and S' are similar, it is relatively easy to implement conformance wrappers and state conversion functions, and these implementations can be reused across implementations. This is illustrated by the replicated file system example in Section 3. In this example, we take advantage of the NFS standard by using the same conformance wrapper and state conversion functions to wrap different implementations.

2.2 The BASE Library

The BASE library extends BFT with the features necessary to support the methodology. Figure 1 presents the library's interface.

```
Client call:
int invoke(Byz_req *req, Byz_rep *rep,
           bool read_only);

Execution upcall:
int execute(Byz_req *req, Byz_rep *rep,
            int client, Byz_buffer *non-det);

State conversion upcalls:
int get_obj(int i, char** obj);

void put_objs(int nobjs, char **objs,
              int *indices, int *sizes);

Checkpointing:
void modify(int nobjs, int* objs);

Non-determinism upcalls:
int propose_value(Seqno seqno, Byz_buffer *req,
                  Byz_buffer *non-det);

int check_value(Seqno seqno, Byz_buffer *req,
                Byz_buffer *non-det);

Recovery upcalls:
void shutdown_proc(FILE *out);

void restart_proc(FILE *in);
```

Figure 1: BASE Interface and Upcalls

The `invoke` procedure is called by the client to invoke an operation on the replicated service. This procedure carries out the client side of the replication protocol and returns the result when enough replicas have responded.

When the library needs to execute an operation at a replica, it makes an upcall to an `execute` procedure that is implemented by the conformance wrapper for the service implementation run by the replica.

To perform state transfer in the presence of Byzantine faults, it is necessary to be able to prove that the state being transferred is correct. Otherwise, faulty replicas could corrupt the state of out-of-date but correct replicas. (A detailed discussion of this point can be found in [8].) Consequently, replicas cannot discard a copy of the state produced after executing a request until they know that the state produced by executing later requests can be proven correct. Replicas could keep a copy of the state after executing each request but this would be too expensive. Instead replicas keep just the current version of the concrete state plus copies of the abstract state produced every k-th request (e.g., k=128). These copies are called checkpoints. Replicas inform each other when they produce a checkpoint and the library only transfers checkpoints between replicas.

Creating checkpoints by making full copies of the abstract state would be too expensive. Instead, the library uses copy-on-write such that checkpoints only contain the differences relative to the current abstract state. Similarly, transferring a complete checkpoint to bring a recovering or out-of-date replica up to date would be too expensive. The library employs a hierarchical state partition scheme to transfer state efficiently. When a replica is fetching state, it recurses down a hierarchy of meta-data to determine which partitions are out-of-date. When it reaches the leaves of the hierarchy (which are the abstract objects), it fetches only the objects that are corrupt or out-of-date.

As mentioned earlier, to implement checkpointing and state transfer efficiently, we require that the abstract state be encoded as an array of objects, where the objects can have variable size. This representation allows state transfer to be done on just those objects that are out-of-date or corrupt.

The current implementation of the BASE library requires the array to have a fixed size. This limits flexibility in the definition of encodings for the abstract state but it is not an intrinsic problem. The maximum number of entries in the array can be set to an extremely large value without allocating extra space for the portion of the array that is not used, and without degrading the performance of state transfer and checking.

To implement state transfer, each replica must provide the library with two upcalls, which implement the *abstraction function* and one of its inverses. These upcalls do not convert the entire state each time they are called because this would be too expensive. Instead, they perform conversions at the granularity of an object in the abstract state array. The abstraction function is implemented by `get_obj`. It receives an object index i, allocates a buffer, obtains the value of the abstract object with index i, and places that value in the buffer. It returns the size for that object and a pointer to the buffer.

The inverse abstraction function receives a new abstract state value and updates the concrete state to match this argument. This function should also work incrementally to achieve good performance. But it cannot process just one abstract object per invocation because there may be invariants on the abstract state that create dependencies between objects. For example, suppose that an object in the abstract state of a file system can be either a file or a directory. If a slow replica misses the operations that create a directory, d, and a file, f, in d, it has to fetch the abstract objects corresponding to d and f from the others. Then, it invokes the inverse abstraction function to bring its concrete state up-to-date. If f is the argument to the first invocation and d is the argument to the second, it is impossible for the first invocation to update the concrete state because it has no information on where to create the file. The reverse order does not work either because the first invocation creates a dangling reference in d.

To solve this problem, `put_objs` receives a vector of objects with the corresponding sizes and indices in the abstract state array. The library guarantees that this upcall is invoked with an argument that brings the abstract state of the replica to a consistent value (i.e., the value of a valid checkpoint).

Each time the `execute` upcall is about to modify an object in the abstract state it is required to invoke a `modify` procedure, which is supplied by the library, passing the object index as argument. This is used to implement copy-on-write to create checkpoints incrementally: the library invokes `get_obj` with the appropriate index and keeps the copy of the object until the corresponding checkpoint can be discarded.

BASE implements a form of state machine replication that requires replicas to behave deterministically. Our methodology uses abstraction to hide most of the non-determinism in the implementations it reuses. However, many services involve forms of non-determinism that cannot be hidden by abstraction. For instance, in the case of the NFS service, the time-last-modified for each file is set by reading the server's local clock. If this were done independently at each replica, the states of the replicas would diverge.

Instead, we allow the primary replica to propose values for non-deterministic choices by providing the `propose_value` upcall, which is only invoked at the primary. (Like BFT [7], BASE uses a primary that proposes sequence numbers for requests and backups that check on the primary and trigger view changes if it misbehaves.) The call receives the client request and the sequence number for that request; it selects a non-deterministic value and puts it in `non-det`. This value is going to be supplied as an argument of the `execute` upcall to all replicas.

The protocol implemented by the BASE library prevents a faulty primary from causing replica state to diverge by sending different values to different backups. However, a faulty primary might send the same, incorrect value to all backups, subverting the system's desired behavior. The solution to this problem is to have each replica implement a `check_value` function that validates the choice of non-deterministic values that was made by the primary. If 1/3 or more non-faulty replicas reject a value proposed by a faulty primary, the request will not be executed and the view change mechanism will cause the primary to be replaced soon after.

Proactive recovery periodically restarts each replica from a correct, up-to-date checkpoint of the abstract state that is obtained from the other replicas. Recoveries are triggered by a watchdog timer. When a replica is recovered, it reboots after saving to disk the abstract service state, and the replication protocol state, which includes abstract objects that were copied by the incremental checkpointing mechanism.

The library could invoke `get_obj` repeatedly to save a complete copy of the abstract state to disk but this would be expensive. It is sufficient to ensure that the current concrete state is on disk and to save a small amount of additional information to enable reconstruction of the conformance representation when the replica restarts. Since the library does not have access to this representation, the service state is saved to a file by an additional upcall, `shutdown`, that is implemented by the conformance wrapper. The conformance wrapper also implements a `restart` upcall that is invoked to reconstruct the conformance representation from the file saved by `shutdown` and from the concrete state of the service. This enables the replica to compute the abstract state by calling `get_obj`.

In some cases, the information in the conformance representation is volatile; it is no longer valid when the replica restarts. In this case, it is necessary to augment it with information that is persistent and allows `restart` to reconstruct the conformance representation after a reboot.

After calling `restart`, the library uses the hierarchical state transfer mechanism to compare the value of the abstract state of the replica with the abstract state values stored by the other replicas. It computes cryptographic hashes of the abstract objects and compares them with the hashes in the state partition tree to check if the objects are corrupt. The state partition tree also contains the sequence number of the last checkpoint when each object was modified [8]. The replica uses this information to check which objects are out-of-date without having to compute their hash. These checks are performed in parallel with fetches of objects that have already been determined to be out-of-date or corrupt. This is efficient: the replica fetches only the value of objects that are out-of-date or corrupt. We use a single threaded implementation with event queues representing objects to fetch and objects to check. Checks are performed while waiting for replies to fetch requests. The replica does not execute operations until it completes the recovery.

The object values fetched by the replica could be supplied to `put_objs` to update the concrete state, but the concrete state might still be corrupt. For example, an implementation may have a memory leak and simply calling `put_objs` will not free unreferenced memory. In fact, implementations will not typically offer an interface that can be used to fix all corrupt data structures in their concrete state. Therefore, it is better to restart the implementation from a clean initial concrete state and use the abstract state to bring it up-to-date.

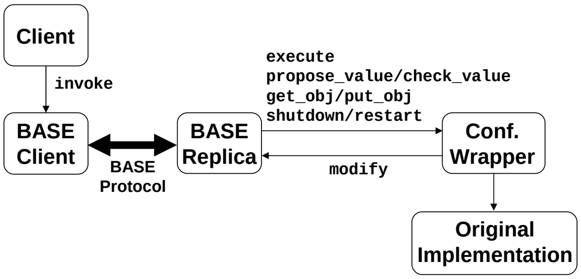


Figure 2: BASE function calls and upcalls

A global view of all BASE functions and upcalls that are invoked is shown in Figure 2.

3. EXAMPLES

This section uses two examples to illustrate the methodology: a replicated file system and an object oriented database.

3.1 File System

The file system is based on the NFS protocol [27]. Its replicas can run different operating systems and file system implementations.

3.1.1 *Abstract Specification*

The common abstract specification is based on the specification of the NFS protocol [27]. The abstract file service state consists of a fixed-size array of pairs containing an object and a generation number. Each object has a unique identifier, *oid*, which is obtained by concatenating its index in the array and its generation number. The generation number is incremented every time the entry is assigned to a new object. There are four types of objects: files, whose data is a byte array; directories, whose data is a sequence of <name, oid> pairs ordered lexicographically by name; symbolic links, whose data is a small character string; and special *null* objects, which indicate that an entry is free. All non-null objects have meta-data, which includes the attributes in the NFS `fattr` structure, and the index (in the array) of its parent directory. Each entry in the array is encoded using XDR [26]. The object with index 0 is a directory object that corresponds to the root of the file system tree that was mounted.

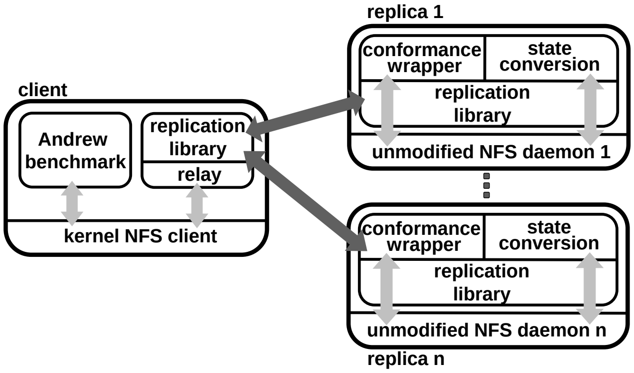


Figure 3: Software architecture.

Keeping a pointer to the parent directory is redundant, since we can derive this information by scanning the rest of the abstract state. But it simplifies the inverse abstraction function and the recovery algorithm, as we will explain later.

The operations in the common specification are those defined by the NFS protocol. There are operations to read and write each type of non-null object. The file handles used by the clients are the *oids* of the corresponding objects. To ensure deterministic behavior, we require that *oid*s be assigned deterministically, and that directory entries returned to a client be ordered lexicographically.

The abstraction hides many details; the allocation of file blocks, the representation of large files and directories, and the persistent storage medium and how it is accessed. This is desirable for simplicity and performance. Additionally, abstracting from implementation details like resource allocation improves resilience to sofware faults due to aging because proactive recovery can fix resource leaks.

3.1.2 *Conformance Wrapper*

The conformance wrapper for the file service processes NFS protocol operations and interacts with an off-the-shelf file system implementation using the NFS protocol as illustrated in Figure 3. A file system exported by the replicated file service is mounted on the client machine like any regular NFS file system. Application processes run unmodified and interact with the mounted file system through the NFS client in the kernel. We rely on user level relay processes to mediate communication between the standard NFS client and the replicas. A relay receives NFS protocol requests, calls the `invoke` procedure of our replication library, and sends the result back to the NFS client. The replication library invokes the `execute` procedure implemented by the conformance wrapper to run each NFS request. This architecture is similar to BFS [7].

The conformance representation consists of an array that corresponds to the one in the abstract state but it does not store copies of the objects; instead each array entry contains the type of object, the generation number, and for non-empty entries it also contains the file handle assigned to the object by the underlying NFS server, the value of the timestamps in the object's abstract meta-data, and the index of the parent directory. The representation also contains a map from file handles to *oids* to aid in processing replies efficiently.

The wrapper processes each NFS request received from a client as follows. It translates the file handles in the request, which encode *oid*s, into the corresponding NFS server file handles. Then it sends the modified request to the underlying NFS server. The server processes the request and returns a reply.

The wrapper parses the reply and updates the conformance representation. If the operation created a new object, the wrapper allocates a new entry in the array in the conformance representation, increments the generation number, and updates the entry to contain the file handle assigned to the object by the NFS server and the index of the parent directory. If any object is deleted, the wrapper marks its entry in the array free. In both cases, the reverse map from file handles to *oids* is updated.

The wrapper must also update the abstract timestamps in the array entries corresponding to objects that were accessed. For this, it uses the value for the current clock chosen by the primary using the `propose_value` upcall in order to prevent the states of the replicas from diverging. However, if a faulty primary chooses an incorrect value the system could have an incorrect behavior. For example, the primary might always propose the same value for the current time; this would cause all replicas to update the modification time to the same value that it previously held and therefore, according to the cache consistency protocol implemented by most NFS clients [3], cause the clients to erroneously not invalidate their cached data, thus leading to inconsistent values at the caches of different clients. The solution to this problem is to have each replica validate the choice for the current timestamp using the `check_value` function. In this case, this function must guarantee that the proposed timestamp is not too far from the replica's own clock value, and that the timestamps produced by the primary are monotonically increasing.

Finally, the wrapper returns a modified reply to the client, using the map to translate file handles to *oids* and replacing the concrete timestamp values by the abstract ones.

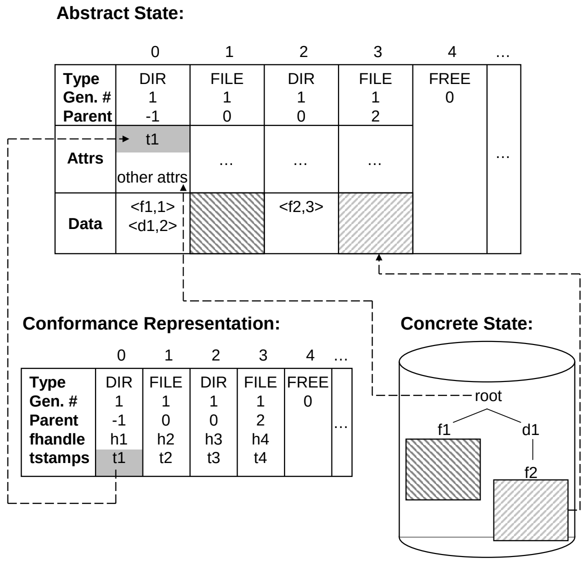
When handling *readdir* calls the wrapper reads the entire directory and sorts it lexicographically to ensure the client receives identical replies from all replicas.

3.1.3 State Conversions

The abstraction function in the file service is implemented as follows. For each file system object, it uses the file handle stored in the conformance representation to invoke the NFS server to obtain the data and meta-data for the object. Then it replaces the concrete timestamp values by the abstract ones, converts the file handles in directory entries to *oids*, and sorts the directories lexicographically.

Figure 4 shows how the concrete state and the conformance representation are combined to form the abstract state for a particular example. Note that the attributes in the concrete state are combined with the timestamps in the conformance representation to form the attributes in the abstract state. Also note that the contents of the files and directories are not stored by the conformance representation, but only in the concrete state.

The pseudocode for the inverse abstraction function in the file service is shown in Figure 5. This function receives an array with the indices of the objects that need to be updated and the new values for those objects. It scans each entry in the array to determine the type of the new object, and acts accordingly.



	0	1	2	3	4	...
Type	DIR	FILE	DIR	FILE	FREE	
Gen. #	1	1	1	1	0	
Parent	-1	0	0	2		
Attrs	t1 other attrs	...	...	...		...
Data	<f1,1> <d1,2>	▨	<f2,3>	▨		

	0	1	2	3	4	...
Type	DIR	FILE	DIR	FILE	FREE	...
Gen. #	1	1	1	1	0	
Parent	-1	0	0	2		
fhandle	h1	h2	h3	h4		
tstamps	t1	t2	t3	t4		

Figure 4: Example of the abstraction function

If the new object is a file or a symbolic link, it starts by calling the `update_directory` function, passing the new object's parent directory index as an argument. This will cause the object's parent directory to be reconstructed if needed, and the corresponding object in the underlying file system will be created if it did not exist already. Then it can update the object's entry in the conformance representation, and issue a `setattr` and a `write` to update the file's meta-data and data in the concrete state. For symbolic links, it is sufficient to update their meta-data.

```
function put_objs(in modified object array)
  for each entry ∈ modified object array do
    if new object's type = file or symbolic link then
      update_directory(new object's parent directory index)
      update object's meta-data in the conformance representation
      set file attributes
      write file's new contents or update link's target
    else if new object's type = directory then
      update_directory(this object's index)
      update object's meta-data in the conformance representation
    else if new object's type = free entry then
      update object's type in the conformance representation

function update_directory(in directory's index)
  if (directory has already been updated or
    directory has not changed) then
    do nothing
  else
    update_directory(new directory's parent directory index)
    read directory's old contents using NFS calls
    for each entry in old directory do
      if entry is not in new directory then
        remove entry using NFS call
      else if entry is in new directory but type is wrong then
        remove entry
    for each entry in new directory do
      if entry is not in old directory then
        create entry using NFS call
```

Figure 5: Inverse abstraction function

When the new object is a directory, it is sufficient to invoke update_directory passing its own index as an argument, and then updating the appropriate entry in the conformance representation.

Finally, if the new object is a free entry it updates the conformance representation to reflect the new object's type and generation number. If the entry was not previously free, it must also remove the mapping from the file handle that was stored in that entry to its *oid*. We do not have to update the parent directory of the old object, since it must have changed and will be processed eventually.

The update_directory function can be summarized as follows. If the directory that is being updated has already been updated or is not in the array of objects that need to be updated then the function performs no action. Otherwise it calls itself recursively passing the index of the parent directory (taken from the new object) as an argument. Then, it looks up the contents of the directory by issuing a readdir call. It scans the entries in the old state to remove the ones that are no longer present in the abstract state (or have a different type) and finally scans the entries in the new abstract state and creates the ones that are not present in the old state. When an entry is created or deleted, the conformance representation is updated to reflect this.

3.1.4 Proactive Recovery

After a recovery, a replica must be able to restore its abstract state. This could be done by saving the entire abstract state to disk before the recovery, but that would be very expensive. Instead we want to save only the metadata (e.g., the oids and the timestamps). But to do this we need a way of relating the oids to the files in the concrete file system state. This cannot be done using file handles since they can change when the NFS server restarts. However, the NFS specification states that each object is uniquely identified by a pair of meta-data attributes: <fsid,fileid>. We solve the problem by adding another component to the conformance representation: a map from <fsid,fileid> pairs to the corresponding *oids*. The shutdown method saves this map (as well as the metadata maintained by the conformance representation for each file) to disk.

After rebooting, the restart method performs the following steps. It reads the map from disk; performs a new mount RPC call, thus obtaining the file handle for the file system root; and places null file handles in all the other entries in the conformance representation that correspond to all the other objects, indicating that we do not know the new file handles for those objects yet. It then initializes the other entries using the metadata that was stored by shutdown.

Then the replication library runs the protocol to bring the abstract state of the replica up to date. As part of this process, it updates the digests in its partition tree using information collected from the other replicas and calls get_obj on each object to check if it has the correct digest. Corrupt or out-of-date objects are fetched from the other replicas.

The call to get_obj determines the new NFS file handle if necessary. In this case, it goes up the directory tree (using the parent index in the conformance representation) until it finds a directory whose new file handle is already known. Then it issues a readdir to learn the names and fileids of the entries in the directory, followed by a lookup call for each one of those entries to obtain their NFS file handles; these handles are then stored in the position that is determined by the <fsid,fileid> to *oid* map. Then it proceeds down the path of the object whose file handle is being reconstructed, computing not only the file handles of the directories in that path, but also those of all their siblings in the file system tree.

When walking up the directory tree using the parent indices, we need to detect loops so that the recovery function will not enter an infinite loop due to erroneous information stored by the replica during shutdown.

Currently, we restart the NFS server in the same file system and update its state with the objects fetched from other replicas. We plan to change the implementation to start an NFS server on a second empty disk and bring it up-to-date incrementally as we obtain the values of the abstract objects. This has the advantage of improving fault tolerance as discussed in Section 2. Additionally, it can improve disk locality by clustering blocks from the same file and files that are in the same directory.

3.2 Object-Oriented Database

We have also applied our methodology to replicate the servers in the Thor object-oriented database [18]. In this example, all the replicas run the same server implementation. The example is interesting because the service is more complex than NFS, and the server implementation is multithreaded and exhibits a significant degree of nondeterminism. The methodology enabled reuse of the existing server code and could enable software rejuvenation through proactive recovery. We begin by giving a brief overview of Thor and then describe how the methodology was applied in this example. A more detailed description can be found in [32].

3.2.1 System Overview

Thor [18] provides a persistent object store that can be shared by applications running concurrently at different locations. It guarantees type-safe sharing by ensuring that all objects are used in accordance with their types. Additionally, it provides atomic transactions [14] to guarantee strong consistency in the presence of concurrent accesses and crashes.

Thor is implemented as a client/server system in which servers provide persistent storage for objects and applications run at clients on cached copies of persistent objects.

Servers store objects in *pages* on disk and also cache these pages in main memory. Each object stored by a server is identified by a 32-bit *oref*, which contains a *pagenum* that identifies the page where the object is stored and an *onum* that identifies the object within the page. Objects are uniquely identified by a pair containing the object oref and the identifier of the server that stores the object.

Each client maintains a cache of objects retrieved from servers in main memory [6]. Applications running at the client invoke methods on these cached objects. When an application requests an object that is not cached, the client fetches the page that contains the object from the corresponding server.

Thor uses an optimistic concurrency control algorithm [1] to *serialize* [14] transactions. Clients run transactions on cached copies of objects assuming that these copies are up-to-date but record orefs of objects read or modified by the transaction. To commit a transaction, the client sends a commit request to the server that stores these objects. (Thor uses a two-phase commit protocol [14] when transactions access objects at multiple servers but we will not describe this case to simplify the presentation.) The commit request includes a transaction timestamp assigned by the client, the orefs it recorded, and the new values of modified objects.

The server attempts to serialize transactions in order of increasing timestamps. To determine if a transaction can commit, the server uses a *validation queue* (VQ) and *invalid sets* (ISs). The VQ contains an entry for every transaction that committed recently. Each entry contains the orefs of objects that were read or modified by the transaction, and the transaction's timestamp. There is an IS for each active client that lists orefs of objects with stale copies in that client's cache. A transaction can commit if none of the objects it accessed is in the corresponding IS, if it did not modify an object that was read by a committed transaction in the VQ with a later timestamp, and if it did not read an object that was modified by a transaction in the VQ with a later timestamp. If the transaction commits, its effects are recorded persistently; otherwise, it has no effect. In either case, the server informs the client of its decision.

The server updates the ISs of clients when a transaction commits. It adds orefs of objects modified by the transaction to the ISs of clients that are caching those objects. It computes this set of clients using a *cached-pages directory* that maps each page in the server to the set of clients that may have cached copies of that page. The server adds clients to the directory when they fetch pages and clients piggyback information about pages that they have discarded in fetch and commit requests that they send to the server. Similarly, the servers piggyback invalidation messages on fetch and commit replies to inform clients of objects in their IS. An object is removed from a client's IS when the server receives an acknowledgement for the invalidation. These acknowledgements are also piggybacked on other messages.

When a transaction commits, clients send new versions of modified objects but not their containing pages. These objects are stored by the server in a *modified object buffer* (MOB) [12] that allows the server to defer installing these objects to their pages on disk. The modifications are installed to disk lazily by a background flusher thread when the MOB is almost full to make room for new modifications.

3.2.2 Abstract Specification

We applied our methodology to replicate Thor servers. The abstract specification models the behavior of these servers as seen by clients. The interface exported by servers has four main operations: *start_session* and *end_session*, which are used by clients to start and end sessions with a server; and *fetch* and *commit*, which were described before. Invalidations are piggybacked on fetch and commit replies, and invalidation acknowledgments and notifications of page evictions from client caches are piggybacked on fetch and commit requests.

The abstract state of the service is defined as follows. The array of abstract objects is partitioned into four fixed-size areas.

Database pages. Each entry in this area corresponds to a page in the database. The value of the entry with index i is equal to the value of the page with pagenum i.

Validation Queue. Entries in this area correspond to entries in the VQ. The value of each entry contains the timestamp that was assigned to the corresponding transaction (or a null timestamp for free entries), the status of the transaction, an array with the orefs of objects read by the transaction, and an array with the orefs of objects written by the transaction. When a transaction commits, it is assigned the free entry with the lowest index. When there are no free entries, the entry of the transaction with the lowest timestamp t is discarded to free an entry for a new transaction and any transaction that attempts to commit with timestamp lower than t is aborted [18].

Note that entries are not ordered by timestamp because this could lead to inefficient checkpoint computation and state transfer. Inserting entries in the middle of an ordered sequence could require shifting a large number of entries. This would increase the cost of our incremental checkpointing technique and could increase the amount of data sent during state transfers.

Invalid sets. Each entry in this area corresponds to the invalid set of an active client. The value of an entry contains the client identifier (or a null identifier for free entries), and an array with the orefs of invalid objects. When a new client invokes *start_session*, it is assigned an *abstract client number* that corresponds to the index of its entry in this area. The entry is discarded when the client invokes *end_session*.

Cached-pages directory. There is one entry in this area per database page. The index of an entry is equal to the pagenum of the corresponding page minus the starting index for the area. The value of an entry is an array with the abstract numbers of clients that cache the page.

The abstraction hides the details of how the page cache and the MOB are managed at the servers. This allows different replicas to cache different pages, or install objects to disk pages at different times without having their abstract states diverge.

3.2.3 Conformance Wrapper

Thor servers illustrate one of the problems that make applying our methodology harder. The external interface they offer is too narrow to implement state conversion functions that are both simple and efficient. For example, the interface between clients and servers does not allow reading or writing the validation queue, the invalid sets, or the cached-pages directory.

We could solve this problem by shadowing this data in the wrapper but this is not practical because it would require reimplementing the concurrency control algorithm. Instead, we implemented the state conversion functions using internal APIs. This was possible because we had access to the server source code. We used these internal APIs as black boxes; we did not add new operations or change existing operations. These internal APIs were used only to implement the state conversion functions. They were not used to define the abstract specification. This is important because we want this specification to abstract as many implementation details as possible.

We also replaced the communication library used between servers and clients by one with the same interface that calls the BASE library. This avoids the need for interposing client and server proxies, which was the technique we used in the file system example.

The conformance wrapper maintains only two data structures: the *VQ array* and the *client array*, which are used in the state conversion functions as we will describe next. Each entry in the VQ array corresponds to the entry with the same index in the VQ area of the abstract state, and it contains the transaction timestamp in that abstract entry. When a transaction commits, the wrapper assigns it an entry in the VQ array (as described in the abstract specification) and stores its timestamp there.

The entries in the client array are used to map abstract client numbers to the per-client data structures maintained by Thor. They are updated by the wrapper when clients start and end sessions with the server.

In Thor, transaction timestamps are assigned by clients. The conformance wrapper rejects timestamps that deviate more than a threshold from the time when the commit request is received. This is important to prevent faulty clients from committing transactions with very large timestamps, which could cause spurious aborts. The conformance wrapper uses the `propose_value` and `check_value` upcalls offered by the BASE library for replicas to agree on the time when the commit request is received. Replicas use the agreed upon value to decide whether to reject or accept the proposed timestamp. This ensures that all correct replicas reach the same decision.

Besides maintaining these two data structures and checking timestamps, the wrapper simply invokes the operations exported by the Thor server after calling `modify` to inform the BASE library of which abstract objects are about to be modified.

3.2.4 State Conversions

The `get_obj` upcall receives the index of an abstract object and returns a pointer to a buffer containing the current value of that abstract object. The implementation of `get_obj` in this example uses the index to determine which area the abstract object belongs to. Then, it computes the value of the abstract object using the procedure that corresponds to the object's area:

Database pages. If the abstract object is a database page, `get_obj` retrieves a copy of the page from disk (or from the page cache) and applies any pending modifications to the page that are in the MOB. This is the current value of the page that is returned.

Validation queue. If the object represents a validation queue entry, `get_obj` retrieves the timestamp that corresponds to this entry from the VQ array in the conformance representation. Then, it uses the timestamp to fetch the entry from the VQ maintained by the server, and copies the sets with orefs of objects read or modified by the transaction to compose the value of the abstract object.

Invalid sets. If the object represents an invalid set for a client with number c, `get_obj` uses the client array in the conformance representation to map c to the client data structure maintained by the server for the corresponding client. Then, it retrieves the client invalid set from this data structure and uses it to compose the abstract object value.

Cached-pages directory. In this case, `get_obj` determines the pagenum of the requested abstract object by computing the offset to the beginning of the area. Then, it uses the pagenum to lookup the information to compose the abstract object value in the cached-pages directory maintained by the server.

The `put_objs` upcall receives an array with new values for abstract objects and updates the concrete state to match these values. It iterates over the abstract object values and uses the object indices to determine which of the procedures below to execute.

Database pages. To update a concrete database page, `put_objs` removes any modifications in the MOB for that page to ensure that the new page value will not be overwritten with old modifications. Then, it places a page matching the new abstract value in the server's cache and marks it as dirty.

Validation queue, invalid sets and cached-pages directory. If the relevant server data structure already contains an entry corresponding to a new abstract object value, the function just updates the entry according to the new value. Otherwise, it must delete the entry from the server data structure if the new abstract object value describes a non-existent entry, or create the entry if it did not previously exist and fill in the values according to the new abstract value. The conformance representation is updated accordingly.

4. EVALUATION

Our replication technique must achieve two goals to be successful: it must have low overhead, and the code of the conformance wrapper and the state conversion functions must be simple. It is important for the code to be simple to reduce the likelihood of introducing more errors and to keep the monetary cost of using our technique low. This section evaluates the extent to which both example applications meet each of these goals.

4.1 File System Overhead

This section presents results of experiments that compare the performance of our replicated file system with the off-the-shelf, unreplicated NFS implementations that it wraps.

4.1.1 Experimental Setup

Our technique has three advantages: reuse of existing code, software rejuvenation using proactive recovery, and opportunistic N-version programming. We present results of experiments to measure the overhead in systems that benefit from different combinations of these advantages. We ran experiments with and without proactive recovery in a *homogeneous* setup, where all replicas ran the same operating system, and in a *heterogenous* setup, where each replica ran a different operating system.

All experiments ran with four replicas and one client. Four replicas can tolerate one Byzantine fault; we expect this reliability level to suffice for most applications. Experiments to evaluate the performance of the replication algorithm with more clients and replicas appear in [5].

In the homogeneous setup, clients and replicas ran on Dell Precision 410 workstations with Linux 2.2.16-3 (uniprocessor). These workstations have a 600 MHz Pentium III processor, 512 MB of memory, and a Quantum Atlas 10K 18WLS disk. All machines were connected by a 100 Mb/s switched Ethernet and had 3Com 3C905B interface cards. The switch was an Extreme Networks Summit48 V4.1. The experiments ran on an isolated network.

The heterogeneous setup used the same hardware setup but some replicas ran different operating systems. The client and one of the replicas ran Linux as in the homogeneous setup. The other replicas ran different operating systems: one ran Solaris 8 1/01; another ran OpenBSD 2.8; and the last one ran FreeBSD 4.0.

All experiments ran the modified Andrew benchmark [15, 30], which emulates a software development workload. It has five phases: (1) creates subdirectories recursively; (2) copies a source tree; (3) examines the status of all the files in the tree without examining their data; (4) examines every byte of data in all the files; and (5) compiles and links the files. They ran the scaled up version of the benchmark described in [8] where phase 1 and 2 create n copies of the source tree, and the other phases operate in all these copies. We ran a version of Andrew with n equal to 100, Andrew100, that creates approximately 200 MB of data and another with n equal to 500, Andrew500, that creates approximately 1 GB of data. Andrew100 fits in memory at both the client and the replicas but Andrew500 does not.

The benchmark ran at the client machine using the standard NFS client implementation in the Linux kernel with the following `mount` options: UDP transport, 4096-byte read and write buffers, allowing write-back client caching, and allowing attribute caching. All the experiments report the average of three runs of the benchmark and the standard deviation was always below 7% of the reported values.

4.1.2 Homogeneous Results

Tables 1 and 2 present the results for Andrew100 and Andrew500 in the homogeneous setup with no proactive recovery. They compare the performance of our replicated file system, BASEFS, with the standard, unreplicated NFS implementation in Linux with Ext2fs at the server, NFS-std. In these experiments, BASEFS is also implemented on top of a Linux NFS server with Ext2fs at each replica.

phase	BASEFS	NFS-std
1	0.9	0.5
2	49.2	27.4
3	45.4	39.2
4	44.7	36.5
5	287.3	234.7
total	427.65	338.3

Table 1: Andrew100: elapsed time in seconds

The results show that the overhead introduced by our replication technique is low: BASEFS takes only 26% longer than NFS-std to run Andrew100 and 28% longer to run Andrew500. The overhead is different for the different phases mostly due to variations in the amount of time the client spends computing between issuing NFS requests.

There are two main sources of overhead: the cost of running the Byzantine-fault-tolerant replication protocol, and the cost of abstraction. The latter includes the time spent running the conformance wrapper and the time spent running the abstraction function to compute checkpoints of the abstract file system state. We estimate that the Byzantine fault tolerance protocol adds approximately 15% of overhead relative to NFS-std in Andrew100 and 20% in Andrew500. This estimate is based on the overhead of BFS relative to NO-REP for Andrew100 and Andrew500 that was reported in [8]. We expect this estimate to be fairly accurate: BFS is very similar to BASEFS except that it does not use abstraction, and NO-REP is identical to BFS except that it is not replicated. The remaining overhead of 11% relative to NFS-std in Andrew100 and 8% in Andrew500 can be attributed to abstraction.

We also ran Andrew100 and Andrew500 with proactive recovery. The results, which are labeled BASEFS-PR, are shown in Table 3. The results for Andrew100 were obtained by recovering replicas round robin with a new recovery starting every 80 seconds, and reboots were simulated by sleeping 30 seconds[1]. We obtained the results for Andrew500 in the same way but in this case a new recovery was started every 250 seconds. This leads to a window of vulnerability of approximately 6 minutes for Andrew100 and 17 minutes for Andrew500; that is, the system will work correctly as long as fewer than 1/3 of the replicas fail in a correlated way within any time window of size 6 (or 17) minutes. The results show that even with these very strong guarantees BASEFS is only 32% slower than NFS-std in Andrew100 and 31% slower in Andrew500.

phase	BASEFS	NFS-std
1	5.0	2.4
2	248.2	137.6
3	231.5	199.2
4	298.5	238.1
5	1545.5	1247.1
total	2328.7	1824.4

Table 2: Andrew500: elapsed time in seconds

system	Andrew100	Andrew500
BASEFS-PR	448.2	2385.1
BASEFS	427.65	2328.7
NFS-std	338.33	1824.4

Table 3: Andrew with proactive recovery: elapsed time to run the benchmark in seconds.

Table 4 presents a breakdown of the time to complete the slowest recovery in Andrew100 and Andrew500. *Shutdown* accounts for the time to write the state of the replication library and the conformance representation to disk, and *restart* is the time to read this information back. *Fetch and check* is the time to rebuild the *oid* to file handle mappings in the conformance wrapper, to convert the state stored by the NFS server to its abstract form and check it, and to fetch out-of-date objects from other replicas. Fetching out-of-date objects is done in parallel with converting and checking the state.

	Andrew100	Andrew500
shutdown	0.07	0.32
reboot	30.05	30.05
restart	0.18	0.97
fetch and check	18.28	141.37
total	48.58	172.71

Table 4: Andrew: maximum time to complete a recovery in seconds.

The recovery time in Andrew100 is dominated by the time to reboot but as the state size increases reading, converting, and checking the state becomes the dominant cost; this accounts for 141 seconds in Andrew500 (82% of the total recovery time). Scaling to larger states is an issue but we

[1] This reboot time is based on the results obtained by the LinuxBIOS project [23]. They claim to be able to reboot Linux in 35 s by replacing the BIOS with Linux.

could use the techniques suggested in [8] that make the cost of checking proportional to the number of objects modified in a time period rather than to the total number of objects in the state.

As mentioned, we would like our implementation of proactive recovery to start an NFS server on a second empty disk with a clean file system to improve the range of faults that can be tolerated. Extending our implementation in this way should not affect the performance of the recovery significantly. We would write each abstract object to the new file system asynchronously right after checking it. Since the value of the abstract object is already in memory at this point and it is written to a different disk, the additional overhead should be minimal.

4.1.3 Heterogeneous Results

Table 5 presents results for Andrew100 with and without proactive recovery in the heterogenous setup. In this experiment, each BASEFS replica runs a different operating system with a different NFS and file system implementation. The table also presents results for the standard NFS implementation in each operating system without replication.

system	elapsed time
BASEFS-PR	1950.6
BASEFS	1662.2
OpenBSD	1599.1
Solaris	1009.2
FreeBSD	848.4
Linux	338.3

Table 5: Andrew100 heterogeneous: elapsed time in seconds

The overhead of BASEFS in this experiment varies from 4% relative to the slowest replica (OpenBSD) to 391% relative to the fastest replica (Linux). The replica running Linux is much faster than all the others because Linux does not ensure stability of modified data and meta-data before replying to the client as required by the NFS protocol. The overhead relative to OpenBSD is low because BASEFS only requires a quorum with 3 replicas, which must include the primary, to complete operations. These results were obtained with the primary replica in the machine running Linux. Therefore, BASEFS does not need to wait for the slowest replica to complete operations. However, this replica slows down the others because it gets out-of-date frequently and initiates state transfers. This explains why the overhead relative to the third fastest replica (Solaris) is higher than in the homogeneous case.

We also ran BASEFS with proactive recoveries in the heterogeneous setup. We recovered a new replica every 425 seconds and reboots were simulated by sleeping 30 seconds. In this case, the overhead varies from 22% relative to the slowest replica to 477% relative to the fastest replica.

The overhead of BASEFS-PR relative to BASEFS without proactive recovery is higher in the heterogeneous setup than in the homogeneous setup. This happens because proactive recovery causes the slowest replica to become the primary periodically. During these periods the system must wait for the slowest replica to complete operations.

4.2 Object-Oriented Database Overhead

This section presents results of experiments to measure the overhead of our replicated implementation of Thor relative to the original implementation of Thor without replication. To provide a conservative measurement, the version of Thor without replication does not ensure stability of information committed by a transaction. A real implementation would save a transaction log to disk or use replication to ensure stability as we do. In either case, the overhead introduced by BASE would be lower.

4.2.1 Experimental Setup

We ran four replicas and one client in the homogeneous setup described in Section 4.1. The experiments ran the OO7 benchmark [4], which is intended to match the characteristics of many different CAD/CAM/CASE applications. The OO7 database contains a tree of assembly objects, with leaves pointing to three composite parts chosen randomly from among 500 such objects. Each composite part contains a graph of atomic parts linked by connection objects; each atomic part has 3 outgoing connections. All our experiments ran on the medium database, which has 200 atomic parts per composite part.

The OO7 benchmark defines several database traversals; these perform a depth-first traversal of the assembly tree and execute an operation on the composite parts referenced by the leaves of this tree. Traversals T1 and T6 are read-only; T1 performs a depth-first traversal of the entire composite part graph, while T6 reads only its root atomic part. Traversals T2a and T2b are identical to T1 except that T2a modifies the root atomic part of the graph, while T2b modifies all the atomic parts. We ran each traversal in a single transaction.

The objects are clustered into 4 KB pages in the database. The database takes up 38 MB in our implementation. Each server replica had a 20 MB cache (of which 16 MB were used for the MOB); the client cache had 16MB. All the results we report are for cold traversals: the client and server caches were empty in the beginning of the traversals.

4.2.2 OO7 Results

The results in Figure 6 are for read-only traversals. We measured elapsed times for T1 and T6 traversals of the database, both in the original implementation, Thor, and the version that is replicated with BASE, BASE-Thor. The figure shows the total time to run the transaction broken into the time to run the traversal and the time to commit the transaction.

BASE-Thor takes 39% more time to complete T1, and 29% more time to complete T6. The commit cost is a small fraction of the total time in these experiments. Therefore, most of the overhead is due to an increase in the cost to fetch pages. The micro-benchmarks in [8] predict an overhead of 60% when fetching 4 KB pages with no computation at the client or the replicas. The overhead here is lower because the pages have to be read from the replicas' disks. Similarly, the relative overhead is lower for traversal T6 because it generates disk accesses with less locality. Thus, the average time to read a disk page from the server disk is higher in T6 than in T1. We expect a similar effect in more realistic settings where the database does not fit in main memory at either the server or the clients. In these settings, BASE will have lower overhead because the cost of disk accesses will

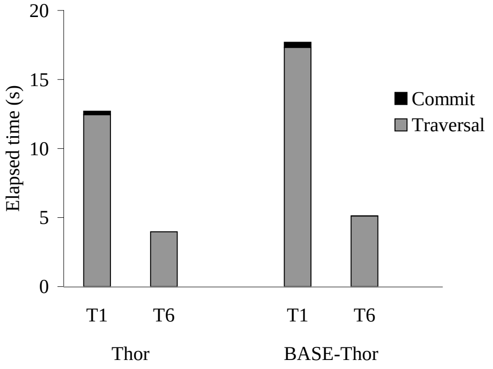


Figure 6: OO7: Elapsed time, cold read-only traversals

dominate performance.

Figure 7 shows elapsed times for read-write traversals. In this case, BASE adds an overhead relative to the original implementation of 38% in T2a and 45% in T2b. The traversal times for T1, T2a, and T2b are almost identical because these traversals are very similar. What is different is the time to commit the transactions. Traversal T2a modifies 500 atomic parts whereas T2b modifies 100000. Therefore, the commit time is a significant fraction of the total time in traversal T2b but not in traversal T2a. BASE increases the commit overhead significantly due to the cost of maintaining checkpoints. The overhead for read-write traversals would be significantly lower relative to a version of Thor that ensured stability of transaction logs.

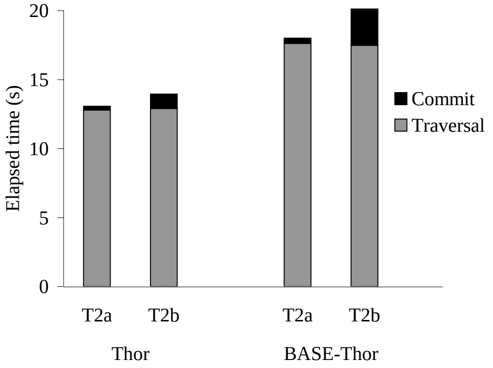


Figure 7: Elapsed time, cold read-write traversals

4.3 Code Complexity

To implement the conformance wrapper and the state conversion functions, it is necessary to write new code. It is important for this code to be simple so that it is easy to write and not likely to introduce new bugs. We measured the number of semicolons in the code we wrote for the replicated file system and for the replicated database to evaluate its complexity. Counting semicolons is better than counting lines because it does not count comment and blank lines.

The code we wrote for the replicated file system has a total of 1105 semicolons with 624 in the conformance wrapper and 481 in the state conversion functions. Of the semicolons in the state conversion functions, only 45 are specific to proactive recovery. The code of the conformance wrapper is trivial. It has 624 semi-colons only because there are many NFS operations. The code of the state conversion functions is slightly more complex because it involves directory tree traversals with several special cases but it is still rather simple. To put these numbers in perspective, the number of semicolons in the code in the Linux 2.2.16 kernel that is directly related with the file system, NFS, and the driver of our SCSI adapter is 17735. Furthermore, this represents only a small fraction of the total size of the operating system; Linux 2.2.6 has 544442 semicolons including drivers and 229095 semicolons without drivers.

The code we wrote for the replicated database has a total of 658 semicolons with 345 in the conformance wrapper and 313 in the state conversion functions. To put these numbers in perspective, the number of semicolons in the original Thor code is 37055.

5. RELATED WORK

Our technique for software rejuvenation [16] is based on the proactive recovery technique implemented in BFT [8]. But the use of abstraction allows us to tolerate software errors due to aging that could not be tolerated in BFT, e.g., resource leaks in the service code. Additionally, it allows us to combine proactive recovery with N-version Programming.

N-Version Programming [9] exploits design diversity to reduce common mode failures. It works as follows: N software development teams produce different implementations of the same service specification for the same customer; the different implementations are then run in parallel; and voting is used to produce a common result. This technique has been criticized for several reasons [13]: it increases development and maintenance costs by a factor of N or more, and it adds unacceptable time delays to the implementation. In general, this is considered to be a powerful technique, but with limited usability since only a small subset of applications can afford its cost.

BASE enables low cost N-version programming by reusing existing implementations from different vendors. Since each implementation is developed for a large number of customers, there are significant economies of scale that keep the development, testing, and maintenance costs per customer low. Additionally, the cost of writing the conformance wrappers and state conversion functions is kept low by taking advantage of existing interoperability standards. The end result is that our technique will cost less and may actually be more effective at reducing common mode failures because competitive pressures will keep implementations from different vendors independent.

Recovery of faulty versions has been addressed in the context of N-Version Programming [33, 36] but these approaches have suffered from two problems. First, they are inefficient and cannot scale to services with large state. Second, they require detailed knowledge of each version, which precludes our opportunistic N-Version programming technique. For example, Romanovsky [33] proposes a technique where each version defines a conversion function from its concrete state to an abstract state. But this abstract state is based on what is common across the implementations of the different versions and the conversion functions have glass-box access to each implementation.

Our technique improves on this by providing a very efficient recovery mechanism and by treating each implementation as a black box — the state conversion functions use only existing interfaces, which is important to allow the reuse of existing implementations. Furthermore, we derive the abstract state from an abstract behavioral specification that captures what is visible to the client succinctly; this leads to better fault tolerance and efficiency.

Several other systems have used wrapping techniques to replicate existing components, e.g., [10, 22, 19, 2, 21, 24, 25]. Many of these systems have relied on standards like NFS or CORBA [28] to simplify wrapping of existing implementations. For example, Eternal [24] is a commercial implementation of the new Fault Tolerant CORBA standard [29]. All these systems assume benign faults except Immune [25].

There are significant differences between these systems and BASE. First, they assume that replicas run identical implementations. They also assume that replicas are deterministic or they resolve the non-determinism at a low level of abstraction. For example, many resolve non-determinism by having a primary run the operations and ship the resulting state or a log with all non-deterministic events to the backups (e.g., [2]). This does not work with Byzantine faults, and replicas are more likely to fail at the same time because they are forced to behave identically at a low level of abstraction. Finally, these systems leave state transfer to the application.

BASE uses abstraction to hide most non-determinism and to enable replicas to run different implementations. It also offers an efficient mechanism for replicas to agree on non-deterministic choices that works with Byzantine faults. This is important when these choices are directly visible by clients, e.g., timestamps. Additionally, we provide support for efficient state transfer and for incremental conversion between abstract and concrete state, which is important because these are harder with Byzantine faults.

The work described in [34] uses wrappers to ensure that an implementation satisfies an abstract specification. These wrappers use the specification to check the correctness of outputs generated by the implementation and contain faults. They are not used to enable replication with different or non-deterministic implementations as in BASE.

6. CONCLUSION

Software errors are a major cause of outages and they are increasingly exploited in malicious attacks to gain control or deny access to important services. Byzantine fault tolerance allows replicated systems to mask some software errors but it has been expensive to deploy. We have described a replication technique, BASE, which uses abstraction to reduce the cost of deploying Byzantine fault tolerance and to improve its ability to withstand attacks and mask software errors.

BASE reduces cost because it enables reuse of off-the-shelf service implementations without modifications, and it improves resilience to software errors by enabling opportunistic N-version programming, and software rejuvenation through proactive recovery.

Opportunistic N-version programming runs distinct, off-the-shelf implementations at each replica to reduce the probability of common mode failures. To apply this technique, it is necessary to define a common abstract behavioral specification for the service and to implement appropriate conversion functions for the state, requests, and replies of each implementation in order to make it behave according to the common specification. These tasks are greatly simplified by basing the common specification on standards for the interoperability of software from different vendors; these standards appear to be common, e.g., ODBC [11], and NFS [27]. Opportunistic N-version programming improves on previous N-version programming techniques by avoiding the high development, testing, and maintenance costs without compromising the quality of individual versions.

Additionally, we provide a mechanism to repair faulty replicas. Proactive recovery allows the system to remain available provided no more than 1/3 of the replicas become faulty and corrupt the abstract state (in a correlated way) within a window of vulnerability. Abstraction may enable service availability even when more than 1/3 of the replicas are faulty because it can hide corrupt items in concrete states of faulty replicas.

The paper described BASEFS — a replicated NFS file system implemented using our technique. The conformance wrapper and the state conversion functions in our prototype are simple, which suggests that they are unlikely to introduce new bugs and that the monetary cost of using our technique would be low.

We ran the Andrew benchmark to compare the performance of our replicated file system and the off-the-shelf implementations that it reuses. Our performance results indicate that the overhead introduced by our technique is low; BASEFS performs within 32% of the standard NFS implementations that it reuses.

We also used the methodology to build a Byzantine fault-tolerant version of the Thor object-oriented database [18] and made similar observations. In this case, the methodology enabled reuse of the existing database code, which is non-deterministic.

As future work, it would be interesting to apply the BASE technique to a relational database service by taking advantage of the ODBC standard. Additionally, a library of mappings between abstract and concrete states for common data structures would further simplify our technique.

7. ACKNOWLEDGMENTS

We would like to thank Chandrasekhar Boyapati, João Garcia, Ant Rowstron, Larry Peterson (our shepherd), and the anonymous referees for their helpful comments on drafts of this paper. We also thank Charles Blake, Benjie Chen, Dorothy Curtis, Frank Dabek, Michael Ernst, Kevin Fu, Frans Kaashoek, David Mazières and Robert Morris for help in providing an infrastructure to run some of the experiments.

8. REFERENCES

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control using Loosely Synchronized Clocks. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 23–34, San Jose, CA, May 1995.

[2] T. Bressoud and F. Schneider. Hypervisor-based Fault Tolerance. In *Proceeding of the 15th ACM Symposium on Operating System Principles*, pages 1–11, Dec. 1995.

[3] B. Callaghan. *NFS Illustrated.* Addison-Wesley, 1999.

[4] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 Benchmark. In *Proceedings of ACM SIGMOD*

International Conference on Management of Data, pages 12–21, Washington D.C., May 1993.

[5] M. Castro. *Practical Byzantine Fault-Tolerance*. PhD thesis, Massachusetts Institute of Technology, 2000.

[6] M. Castro, A. Adya, B. Liskov, and A. Myers. HAC: Hybrid Adaptive Caching for Distributed Storage Systems. In *Proceeding of the 16th ACM Symposium on Operating System Principles*, pages 102–115, St. Malo, France, Oct. 1997.

[7] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, LA, Feb. 1999.

[8] M. Castro and B. Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, CA, Oct. 2000.

[9] L. Chen and A. Avizienis. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. In *Fault Tolerant Computing, FTCS-8*, pages 3–9, 1978.

[10] E. Cooper. Replicated Distributed Programs. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 63–78, Dec. 1985.

[11] K. Geiger. *Inside ODBC*. Microsoft Press, 1995.

[12] S. Ghemawat. *The Modified Object Buffer: a Storage Management Technique for Object-Oriented Databases*. PhD thesis, Massachusetts Institute of Technology, 1995.

[13] J. Gray and D. Siewiorek. High-availability computer systems. *IEEE Computer*, 24(9):39–48, Sept. 1991.

[14] J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1993.

[15] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988.

[16] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, modules and applications. In *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing*, pages 381–390, Pasadena, CA, June 1995.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[18] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99)*, Lisbon, Portugal, June 1999.

[19] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *Proceeding of the 13th ACM Symposium on Operating System Principles*, pages 226–238. ACM Press, 1991.

[20] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[21] S. Maffeis. Adding group communication and fault tolerance to CORBA. In *Proceedings of the 2nd USENIX Conference on Object-Oriented Technologies*, pages 135–146, June 1995.

[22] K. Marzullo and F. Schmuck. Supplying high availability with a standard network file system. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 447–453. IEEE, June 1988.

[23] R. Minnich. The Linux BIOS Home Page. http://www.acl.lanl.gov/linuxbios, 2000.

[24] L. Moser, P. Melliar-Smith, and P. Narasimhan. Consistent object replication in the eternal system. *Theory and Practice of Object Systems*, 4(2):81–92, Jan. 1998.

[25] P. Narasimhan, K. Kihlstrom, L. Moser, and P. Melliar-Smith. Providing Support for Survivable CORBA Applications with the Immune System. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 507–516, May 1999.

[26] Network working group request for comments: 1014. XDR: External data representation standard, June 1987.

[27] Network working group request for comments: 1094. NFS: Network file system protocol specification, March 1989.

[28] Object Management Group. The Common Object Request Broker: Architecture and Specification. OMG techical committee document formal/98-12-01, June 1999.

[29] Object Management Group. Fault Tolerant CORBA. OMG techical committee document orbos/2000-04-04, Mar. 2000.

[30] J. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proceedings of USENIX Summer Conference*, pages 247–256, Anaheim, CA, June 1990.

[31] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, Apr. 1980.

[32] R. Rodrigues. Combining abstraction with Byzantine fault-tolerance. Master's thesis, Massachusetts Institute of Technology, 2001.

[33] A. Romanovsky. Faulty version recovery in object-oriented N-version programming. *IEE Proceedings - Software*, 147(3):81–90, June 2000.

[34] F. Salles, M. Rodriguez, J. Fabre, and J. Arlat. MetaKernels and Fault Containment Wrappers. In *Proceedings the 29th Annual International Symposium on Fault-Tolerant Computing*, pages 22–29, June 1999.

[35] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[36] K. Tso and A. Avizienis. Community error recovery in N-version software: A design study with experimentation. In *Proceedings of the 17th Annual International Symposium on Fault-Tolerant Computing*, pages 127–133, Pittsburgh, PA, July 1987.