The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment

Mary Baker, Mark Sullivan – University of California, Berkeley

ABSTRACT

As organizations with high system availability requirements move to UNIX, the elimination of down-time in the UNIX environment becomes a more important issue. Designing for fast recovery, rather than crash prevention, can provide low-cost highly-available systems without sacrificing performance or simplicity. In Sprite, a UNIX-like distributed operating system, we accomplish this fast recovery in part through the use of a *recovery box*: a stable area of memory in which the system stores carefully selected pieces of system state, and from which the system can be regenerated quickly. Error detection using checksums allows the system to revert to its traditional reboot sequence if the recovery box data is corrupted during system failure. Recent statistics about the types and frequencies of operating system failures indicate that fast recovery using the recovery box will be possible most of the time. Using our recovery box implementation, a Sprite file server recovers in 26 seconds and a database manager with ten remote client processes recovers in six seconds – fast enough that many users and applications will not care that the system crashed.

Introduction

Increasing workstation performance is making UNIX and related operating systems more attractive in environments that also value high system availability. Unfortunately, measurements from Internet sites [12] indicate that UNIX machines fail on average once every two weeks. To maintain high availability, these systems must either reduce the failure rate substantially or recover very quickly after errors.

Traditional fault tolerant systems strive for high availability by eliminating or masking failures, so that the system never goes down. In doing so, they sacrifice some combination of the traditional strengths of the UNIX environment: low hardware cost, high performance during normal execution, and simplicity. For example, Tandem [5] and Stratus [23] provide non-stop processing through hardware redundancy and sometimes software redundancy. Auragen [7] applies redundant hardware and a modified process pair scheme to the UNIX environment. The hardware support is costly, and redundant software techniques either reduce normal performance or increase implementation complexity. Harp's [11] application of replicated hardware and software to NFS file servers suffers no performance degradation, but it increases system complexity. HA-NFS [6] improves the availability of NFS file servers through specialized, redundant hardware.

Other fault-tolerant systems provide high availability through recovery schemes that are significantly more complex than the usual UNIX recovery path. Integrity-S2 [10] detects system errors as they occur and attempts to correct affected internal data structures while the system is running. MVS [3] uses a multi-level recovery scheme in which different portions of the system can fail and recover independently. In contrast, UNIX system designers have generally chosen a simple recovery paradigm that is easy to understand and test: when UNIX detects a serious error, the system just shuts down and reboots from scratch (a ''hard reboot'').

To provide relatively high availability while retaining UNIX strengths, we have built a system that allows failures but recovers quickly using a simple, two-tiered recovery mechanism. After a failure, the system first tries to recover quickly from backup data that it stored in main memory during regular execution. If this fast recovery fails, the system just reverts to the traditional disk-based hard reboot. Existing failure statistics [9, 21, 22] show that most common failures will not corrupt the main memory backup data, so the fast recovery path should be successful most of the time. As long as corrupted data is detected, the traditional recovery path will allow us to retain current reliability.

In order to preserve system state across failures, we have designed and implemented a *recovery box* in Sprite [15], a distributed UNIX-compatible operating system. The recovery box is an area of memory used to store recovery information. It can be implemented using non-volatile RAM for protection from power failures. During execution, the operating system stores backup copies of system data in the recovery box. For example, the Sprite file server stores data about files open or cached on its client workstations. After a failure, the system retrieves these items from the box. If any errors are encountered during recovery, the recovery box is cleared and the system reboots from scratch in the traditional fashion. If no errors are detected, the retrieved items are used to regenerate the system state quickly. Computing and storing checksums on items inserted into the recovery box helps detect memory corruption caused by software.

We have provided a data insertion and retrieval interface that allows the recovery box to be used by both the operating system and application programs. The recovery box has been used for fast recovery in the Sprite distributed file system and in an experimental version of POSTGRES [19], a database management system (DBMS) with clients running on different machines. Using the recovery box and other techniques, we have reduced the combined recovery time for a Sprite file server and the POSTGRES DBMS from many minutes to a total of 32 seconds. Other applications could use the recovery box to reduce down time if they have state that is: (1) slow to regenerate after failures, (2) small enough to fit in main memory, (3) updated frequently enough that it should not be stored on disk, and (4) unlikely to propagate the error that caused the system to fail.

This paper describes the motivation for the recovery box, its implementation, and some preliminary performance results. The first section gives statistics on the types of failures that occur in operating systems. It is these statistics on system failure types that lead us to believe the recovery box memory will be undamaged after the majority of software failures. If the recovery box memory has not been corrupted, then the system need not resort to a hard reboot. The section on implementation shows how Sprite and POSTGRES use the recovery box and explains the implementation and its memory layout. Finally, an evaluation section presents measurements of the recovery box's impact on recovery speed and regular execution performance.

Failure Statistics

New statistics on the frequencies of different types of system outages indicate that the recovery box will be intact and able to provide fast recovery from most failures. These statistics suggest that most failures are due to software errors, and that the most common types of software errors will leave the recovery box data undamaged.

Published data about the frequency of different kinds of outages is scarce, but a study of Tandem systems shows that faulty software is responsible for most failures [9]. Over time, Tandem systems have experienced fewer outages caused by hardware failures, environment failures, and operator errors. Software failures, on the other hand, have remained constant. Table 1 shows the percentages of each source of outage. In 1990, software errors accounted for 62% of Tandem system failures, while only 7% were caused by hardware. The trend towards faster increases in hardware reliability than in software reliability holds true in other environments as well.

Outage Sources	Percent
Software failures	62
Operator errors	15
Hardware failures	7
Environment failures	6
Scheduled maintenance	5
Unknown	5
Total	100

Table 1: Distribution of outage types. The table shows the distribution of types of outages occurring in Tandem systems between 1985-1990. Environment failures are caused by floods, fires, and long power outages.

Two studies that categorize the types of operating system software errors indicate that most such errors will not corrupt the recovery box [21, 22]. These two studies, summarized in Table 2, focus on the frequency of addressing errors, which are the errors that cause programs to corrupt memory. The BSD UNIX study divides errors into synchronization (47%), exception-handling (12%), addressing (12%), and miscellaneous (29%) errors. Exception-handling errors are those that occur in code for handling other errors, including transient hardware errors. The MVS study classifies errors in terms of low-level programming errors, of which 41% were ''control'' problems, 30% were addressing errors, 21% were miscellaneous, and 8% were data miscalculations. Control errors include such problems as deadlock, in which the program stops without corrupting anything but transient state. Data miscalculations include errors in which the wrong variable is used or a function returns the wrong value. Most of the errors classified as miscellaneous are related to performance or denial of service. Addressing problems do not cause the majority of software errors in either UNIX or MVS.

In addition, the MVS study shows that most memory corruption due to addressing errors is local to the data structure being manipulated. As shown in Table 3, at least 57% of addressing errors either corrupt the data structure the operating system intends to modify, or else corrupt memory immediately following the data structure. Only 19% of the MVS addressing errors covered in the study damaged parts of the system unrelated to the one where the error occurred. For example, a common type of addressing error in MVS is *copy overrun* in which a copy transfers too many bytes from one buffer into another, overwriting the data structure that follows the intended destination. A second common addressing error, called an *allocation management* error, occurs when the operating system continues to use a structure after deallocating it.

Error Classes	BSD UNIX	MVS
Addressing-related errors	12	30
Control-related errors	NA	41
Data miscalculation errors	NA	8
Synchronization-related errors	47	NA
Exception-related errors	12	NA
Miscellaneous errors	29	21
Total	100	100

Table 2: Software error type distributions. The table shows the distribution of software errors analyzed in studies of error reports from the IBM MVS and 4.1/4.2 BSD UNIX operating systems. The results columns give the percentage of errors that fall into each category. The two studies used different classification schemes, but both list addressing errors – the errors most likely to corrupt recovery box memory. The other error classes are described in the text.

With these statistics in mind, we have designed an interface to the recovery box that reduces the possibility of corruption from software failures. For example, corruption is unlikely if we use the recovery box to store back up copies of critical data structures, rather than allow clients to access recovery box memory directly. Also, an interface that does strict length checking when items are copied into recovery box memory will prevent copy overruns.

Unfortunately, the failure statistics do not give us enough information that we can protect against complex faults involving *error propagation*. If the system fails due to a logical error in one of its data structures, then storing this data in the recovery box could cause us to suffer the same failure after recovery. The only way we can protect the system from it is to choose data structures for insertion in the recovery box that have proven themselves over time to be relatively error-free. In commercial systems using a recovery box (and eventually in Sprite), data gathered from error reports would help indicate which data structures are prone to these propagated errors.

Implementation

To test the effectiveness of the recovery box, we have implemented it in Sprite and have used it for failure recovery of a Sprite file server and a POSTGRES database manager. POSTGRES runs as an application program on the file server and responds to requests from client programs running on other Sprite machines. Both Sprite and POSTGRES have features that allow fast recovery of disk data, so the recovery box experiments focus on recovery of distributed state. We first describe how Sprite and POSTGRES make use of the recovery box. Then we describe the interface through which the kernel and applications access the recovery box. Finally, we show the organization of the recovery box in memory.

Location of Damaged Area	Percent of MVS Errors
Near intended data	57
Anywhere in storage	19
Not evident from error report	24
Total	100

Table 3: Data corrupted by addressing errors. The table shows the relationship between the location of data corrupted by an addressing error in MVS and the location of the intended modification. Usually data corruption occurs near the data owned by the faulty code. In 24% of MVS error reports studied, this information was not evident.

How Sprite Uses the Recovery Box

Although the Sprite distributed operating system is UNIX-compatible at the system call interface, Sprite differs in some important ways from an NFS-style distributed UNIX system [17]. Unlike NFS file servers, Sprite file servers are not stateless. To achieve higher file system performance than NFS, Sprite caches file data on both client workstations and file servers. To maintain cache consistency, Sprite file servers keep information (in *handles*) for files that are open or cached on each client. In addition, the server keeps handles for objects other than files that are accessed through the Sprite file system, such as remote devices and pseudo-devices [24]. After a file server failure, this information must be regenerated before clients can continue opening and closing files. When a file server without a recovery box reboots, clients send the server a description of each of the handles for files and other objects that they have open or cached. The server regenerates its state information from this data sent to it by clients.

Handling this recovery information from the clients can overwhelm the server's processing capabilities, resulting in a *recovery storm* [4]. For a single Sun 4/280 file server with 40 clients, it currently takes over two minutes for all the clients to recover their distributed state.

Using the new recovery box eliminates this recovery-related client/server communication. During normal operation the server maintains structures containing data about each file or other object currently open or cached on each client. On average, the server maintains 10,000 to 15,000 such handles, or about 300 handles per client. From each of these structures the server preserves 52 bytes of essential information in the recovery box, making the server's recovery box space requirements less than a megabyte. Recovery box items must be updated every time a client opens or closes a file. In Sprite, all opens and closes are processed on the file server anyway, in order to maintain cache consistency, so finding the right place to call the recovery box functions was straightforward. We do not expect increased problems with error propagation, because the data stored in the recovery box for each handle is a subset of data that the server gathers from its clients during a hard reboot. When the server reboots using the recovery box, it rebuilds its tables of cache and file information with the data it retrieves from the recovery box. Without the recovery box, these tables would be regenerated by communicating with the client workstations.

An alternative to storing recovery information in main memory is to store it on the file server's disk. For instance, changes to the file handle information could be written to disk. However, it would be necessary to make these updates synchronously, since the file handle changes affect Sprite's distributed cache consistency protocol. Since this information changes on every file open or close, all open and close operations would incur the increased latency of a disk write. Furthermore, measurements of Sprite file system activity have shown bursts of file open and close requests as high as 100 per second – a rate too high to handle using disk storage without a group commit mechanism. Although a group commit would amortize the cost of the disk write across several opens and closes, it would still greatly increase the latency of some of the opens and closes. Another approach, used in Spritely-NFS [14], is to store information about active clients on the server's disk. After a failure, the server only needs to contact these active clients to regenerate the distributed file system state. The state of active clients changes slowly enough that there is no problem maintaining this information on disk, however, the recovery communication with clients could still be significant. The recovery box allows fast recovery without any conversations between client and server, without the complexity of a group commit, and without generating disk traffic on opens and closes.

The recovery box requires the addition of a small amount of new bookkeeping code in the Sprite file system. This code understands the contents of file handles and copies the relevant portions into items to store in the recovery box. It currently uses a hash table to map between file handles and their items. Since files can be open more than once simultaneously on the same client machine, the file system code also maintains reference counts in the hash table entries and in the recovery box items corresponding to file handles. This extra bookkeeping could be largely eliminated if we had the freedom to restructure Sprite's file handles to include the reference count and the ID that refers to the recovery box item.

How POSTGRES Uses the Recovery Box

Recovery performance in the POSTGRES database management system is dominated by the cost of reinitializing the DBMS server's connections with clients. In a conventional database management system, recovery includes the cost of write-ahead log processing (recovering disk state) in addition to client connection reestablishment. POSTGRES has an unconventional storage manager that maintains consistency of data on disk without requiring write-ahead log processing [20], so it does not need the recovery box to avoid this costly recovery step.

Without the recovery box, connection reestablishment is driven by the clients. When the database manager fails, all transactions executing on behalf of clients are aborted and all connection state held at the server is lost. Connection state includes, e.g., authentication information (secret keys or authentication tokens), client addresses, packet sequence numbers, and any packets queued at the DBMS.

When the server recovers, it must wait for the clients to detect that it has failed. After a connection times out, the client must reopen a connection with the DBMS and reauthenticate itself. POSTGRES can establish and authenticate (application-level) connections with only three messages, because it uses a sequenced packet protocol built on top of a datagram protocol (UDP), rather than a protocol based on streams (TCP/IP). After establishing the connection, each client must query the database to find out if its last transaction committed. Then, it must resubmit the transaction or take some other recovery action.

In a system with a recovery box, the DBMS stores authentication information and the client address associated with each connection in a recovery box item. The DBMS also stores the transaction ID of the last transaction it was executing on behalf of each client. Every time the DBMS begins a new transaction, it updates the recovery box item with the new transaction ID. Storing this transaction

ID on disk would be a bad idea, since POSTGRES is already disk bound for workloads with high transaction rates.

After a failure, the DBMS reinitializes its connection data structures from the backup data stored in the recovery box. Once the connections are reinitialized, the DBMS sends a message to each client indicating that (a) recovery has occurred (a new DBMS server ID is sent), and (b) a new sequence number has been chosen by the DBMS. The restart message also indicates the status of the last transaction that the DBMS executed on behalf of the client. If the message initiating the transaction was lost in the failure or if the transaction was aborted, the client must either resubmit the transaction or take some other recovery action. Authentication of the client is reverified when the DBMS receives the next message from the client.

POSTGRES does not use the recovery box to store any of the state associated with its storage system. Storage system performance optimizations requiring non-volatile RAM are discussed in [20]; for example, to reduce commit latency, committed data can be stored in non-volatile RAM instead of on disk. But this technique requires the operating system to guarantee that data stored in non-volatile memory be permanent. The recovery box does not make this guarantee, because if the system detects any errors during the fast recovery path, it will revert to the traditional disk-based recovery path and will discard the contents of the recovery box.

Interface

The recovery box interface is designed to help Sprite and its application programs manage backup data without exposure to some of the common software errors that corrupt main memory. For this reason we chose a structured and relatively inflexible interface; clients of the recovery box must explicitly insert, delete, and update recovery box items. In the structured interface, each item belongs to a type, and all items of the same type have common characteristics, such as size and checksum calculation routine. When a client creates a new type or a new item, the recovery box manager generates a unique ID (*typeID* and *itemID*, respectively). An itemID consists of the typeID and an *itemNumber*. Clients refer to types and items using these IDs.

We chose a structured interface over a more flexible one in which clients directly allocate and manage data structures in a reserved area of memory, because the structured interface provides more opportunity to avoid and detect recovery box corruption. Maintaining item size in the recovery box helps prevent the copy overruns that often caused storage corruption in MVS. Also, the recovery box can detect allocation management errors, because clients explicitly delete items when done with them. The structured interface also makes it easy for the recovery box to provide atomicity of operations. Atomic updates ensure that a system failure that interrupts an update will not cause checksum failure (and a hard reboot).

When an application program begins fast recovery, it must be able to find its recovery box items and begin regenerating state from them. To find these items, the application must be able to remember, across reboots, the type and item IDs assigned to its items by the recovery box. So that the application does not have to store the IDs on disk, the recovery box allows the application to choose its own well-known IDs and map from them to the system-assigned IDs. The application specifies its IDs when it initializes a type or inserts a new item. On recovery, the program maps from its application IDs to the system IDs once, at initialization, and not every time it accesses recovery box items. Allowing applications to specify their own IDs also facilitates sharing of recovery box items between cooperating UNIX processes.

The Sprite kernel and its applications use the same interface to the recovery box, except that applications call the recovery box routines via a system call. Table 4 lists the available functions. In order to initialize a new type of item, the system or application must call `InitItemType`. As parameters to this function, the caller specifies the maximum number of items that can be valid simultaneously, the item size, an optional applicationItemID, and a flag that signals whether checksums should be calculated and stored for items of this type. Applications can free an item type with the `DeleteItemType` function, but we do not expect applications to delete types often, except perhaps when a new application is being tested.

If creating and deleting types occurs frequently, these operations will have poor performance. Freeing a type from the middle of other allocated types can result in a fragmentation problem. The space consumed by the freed type may not be enough for any type allocated subsequently. If this occurs, all the item information and storage arrays can be shifted down (copied) in memory to leave enough space for the new type. The addresses in the per-type information must be updated to point to the new location of the item arrays. This shift operation does not affect clients of the recovery box, because the clients have no direct pointers to internal recovery box data. Shifting the item data may be costly, but it will only occur on type initialization. Another problem is that the shift is not an atomic operation. It is necessary to put a code in the current operation field of the header to signal whether a crash occurred in the middle of a shift. If so, the recovery box is unusable and the system will resort to a hard reboot. If clients of the recovery box need to delete types frequently, a different recovery box design will be necessary.

Operations on types	InitType DeleteType
Operations on single items	InsertItem DeleteItem UpdateItem ReturnItem
Operations on multiple items	InsertItemArray DeleteItemArray UpdateItemArray ReturnItemArray
ID mapping operations	GetTypeIDMapping GetItemIDMapping

Table 4: Recovery box operations. This table lists the operations available through the recovery box interface. The first set of functions applies to item types; the second set applies to individual items, and the third to multiple items. The last set of functions returns the recovery box typeID or itemID when given an application's typeID or itemID.

Interface functions for operating on individual recovery box items include: `InsertItem`, `DeleteItem`, `UpdateItem`, and `ReturnItem`. To insert an item in the recovery box, the caller provides the item's typeID, a pointer to the data for the item, and an optional application itemNumber. DeleteItem frees the space in the recovery box allocated for an item. To update the contents of an item in the recovery box, the caller must provide the itemID and a pointer to the new data for the item. ReturnItem returns a copy of the specified item in a buffer provided by the caller.

Additional interface functions, such as `InsertItemArray` and `ReturnItemArray`, allow applications to operate on multiple items, avoiding the system call overhead that would be incurred by multiple operations on individual items. To insert multiple items the caller must provide the typeID, the number of items it wishes to insert, an optional array giving the application's itemIDs for each item, and an array of items to insert. The function returns an array giving the new system-assigned itemIDs for the inserted items. ReturnItemArray returns copies of all the items for an item type and an array of itemIDs. If there is insufficient space in the buffers given to it as parameters, the function returns an error and the amount of room required for each of the buffers.

Recovery Box Structure

The recovery box is organized in memory to provide fast insert and delete operations and fast access to items and their type information. Figure 1 shows the layout in memory. There are three sections of the recovery box: a header, followed by an array of per-type information, followed by the item area. In the item area there are two arrays for each item type: an array of per-item information and an array storing the items themselves. The type information, item information and item storage are all implemented as arrays for fast access given an array index; the recovery box typeIDs and itemNumbers are actually indices into the per-type and per-item information arrays, respectively. As described below, a portion of the per-item information array implements a free list of unallocated items for each type, providing constant-time item insertion and deletion.

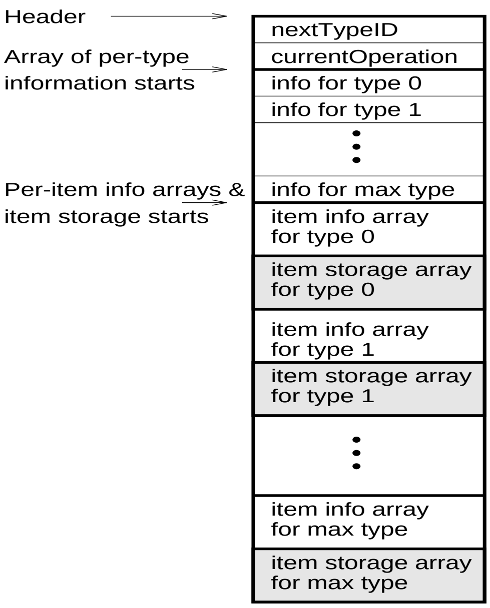

Figure 1: Layout of recovery box in memory. The recovery box layout in memory starts with a header that gives the next typeID to allocate, and provides a code for the current operation. The current operation code is used to ensure atomicity of recovery box insertions, deletions, and updates. The header is followed by an array that gives information about each type of item stored in the table. The per-type information is followed by the per-item information array for the first item type. Following the per-item information for each item type is the array of the items themselves, shown as a shaded entry in the figure.

The header at the very beginning of the of the recovery box contains information that must be saved across fast reboots. The first field in the header, *nextTypeID*, gives the index of the next typeID to be allocated. The second field in the header specifies the current operation on the recovery box in order to ensure that insert, delete, and update item operations are atomic. (Other functions, except those operating on multiple items, are already assured of being atomic.) At the beginning of an insert, delete or update operation, this field is set with a code for the current operation. In

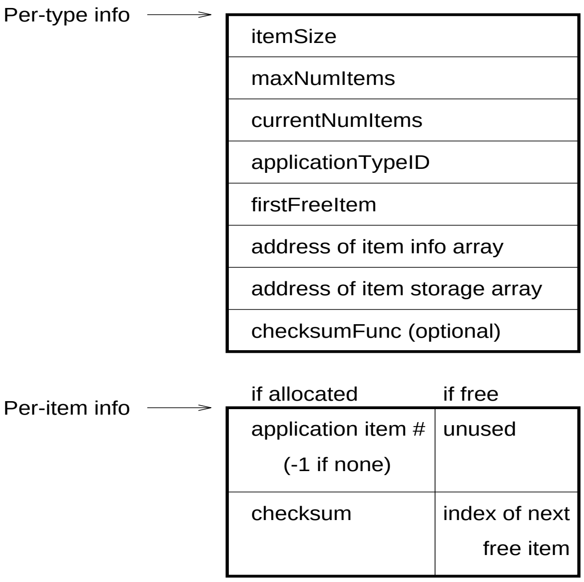

Figure 2: Contents of type and item information arrays. The top half of the figure shows the contents of an entry in the per-type array. This array lists, for each type, the size of the items, the maximum number of items that can be allocated, the current number of allocated items, the itemNumber of the first free item, the memory address for the per-item information array, the memory address for the item storage array, and a possible checksum routine. The lower half of the figure shows the contents of an entry in the per-item information array. This array lists, for each item, the application itemNumber, if one exists, and a possible checksum value for the item. If the item is not allocated, the checksum field instead gives the itemNumber of the next unallocated item.

the case of a delete or update operation, it also includes the target itemID. The field is not cleared until the operation completes, making it possible to detect and back out of incomplete operations. At the beginning of an update, the original value of the item is copied to an extra item space at the end of the item storage array. If the machine crashes during the update, the original value of the item can be retrieved.

The recovery box header is followed by an array, accessed by typeID, that gives information about each item type. The top half of figure 2 shows the information stored in an entry in this array. Each entry lists the item size, the maximum possible number of items, the current number of items, the application typeID, if any, the index in the item storage array of the first free item, the memory address of the per-item information array, the memory address of the item storage array, and a pointer to a checksum routine. If a type's checksum routine field is zero, then no checksums are calculated for that type's items. The generic checksum routine must take as a parameter the size in bytes of the item to be checksummed. However, the checksum routine for the type containing the Sprite kernel's file handle items uses a checksum routine unrolled to optimize checksumming the 52-byte file handle items.

The per-item information arrays are used for fast item allocation and for storing checksums and application itemNumbers. The lower half of figure 2 shows the contents of an entry in such an array. Each entry in the array consists two fields. The first is only valid if the item has been allocated. It contains the application itemNumber, if the application provided one while inserting the item, or a −1 if there is no application itemNumber. The second field has a different meaning depending on whether the item has been allocated or not. If the item has been allocated with a checksum performed on it, the field contains the checksum result. If the item has not been allocated, the field contains the index of the next unallocated item in the array, thus implementing a free list of items. For the last item on the free list, this field is −1. The index of the first free item in the list is stored in the per-type information. Upon deleting an item, its itemNumber is added to the beginning of the list. The free list makes finding free spaces in which to insert new items fast.

Storing the application itemNumbers in the per-item array provides quick mapping from the recovery box system's itemIDs to the application's itemIDs; however, mapping from application IDs to system IDs is not particularly fast. This is why none of the functions in the recovery box interface take the application itemID, except for the function that returns the system itemID given the application itemID. Applications can maintain their own tables to do this mapping quickly, but we leave this functionality outside of the recovery box for the sake of simplicity.

Besides considering the internal memory structure of the recovery box, we must also be able to find the recovery box's memory location after a fast reboot. Our recovery box is always allocated at the same virtual address, so it is easy to locate. Also, the addresses inside the recovery box (specifying the per-item information arrays and storage areas) remain valid without modification across fast reboots. The virtual address of the recovery box must also map to the same physical address upon every fast reboot so that it points to the physical memory containing the recovery box data. The operating system must avoid clearing and initializing this area of memory on a fast reboot. The memory for the recovery box could be battery-backed RAM, but in the absence of non-volatile memory it can be any reserved area of system memory.

We chose to locate the recovery box in a special area of the kernel text segment in order to minimize damage due to memory corruption. The memory pages for the recovery box are marked writable as well as readable, in contrast to the rest of the text segment. We chose the text segment because it reduces the sources of possible system address errors that could overwrite the recovery box. Except for the recovery box, there are no writable data structures in the text segment and no sources of pointer manipulation using text segment addresses. Since error statistics show that most addressing errors are localized around the intended data structures, this should eliminate most addressing errors except for any caused by manipulation of the recovery box itself.

The three main shortcomings in our current recovery box implementation are its lack of access protection, its static memory allocation, and its lack of atomicity for operations on multiple items. At present, there is no security provided by the recovery box, except that only applications with root privilege can access items allocated by the kernel. The static memory allocation imposes a limit on the number of item types that can be initialized and a limit on the overall size of the recovery box. This is why the maximum number of items that will be valid simultaneously for a type must be specified when that type is initialized. The type initialization function returns an error if there is insufficient space for the desired number of items. Currently, the size of the recovery box is compiled into the operating system. If the recovery box resides in non-volatile memory, such as battery-backed RAM, it is likely that the amount of non-volatile memory will already impose a size restriction. Even without such a restriction, placing the recovery box in a static area of system memory, such as the text segment, makes it difficult to expand the recovery box in physical and virtual memory while guaranteeing the same virtual/physical address mapping across reboots. Finally, operations on multiple items are currently not atomic. It would be easy to change our implementation to provide atomicity of multiple item inserts, but providing atomicity of multiple updates and deletes would significantly increase the complexity. While neither the operating system nor the applications we considered would benefit from atomicity of multiple updates, this could be a worthwhile problem to tackle in other environments.

Evaluation

In this section we evaluate the recovery box implementation based on its effect on reboot times and on regular execution performance. To improve reboot times, we use the recovery box to rebuild the operating system and DBMS distributed state, but we have also used a variety of techniques to improve other steps in the reboot sequence. With the recovery box, combined recovery time of Sprite and POSTGRES is about 32 seconds. If the recovery box has been corrupted by a software error or power failure, recovery time will still take many minutes. The effect of the recovery box on regular execution is not large, about 5% overhead on Sprite open/close file operations and less than 1% on POSTGRES debit/credit transactions. The Sprite overhead could be reduced to 2% with optimizations that are not possible in our current development environment. Unless otherwise specified, all measurements were done on a SPARCstation 2 file server (40 Megahertz, 20 integer SPECmarks).

Sprite/POSTGRES Recovery Speed

Although the most lengthy step in rebooting Sprite is distributed recovery, to which we can apply the recovery box, the server must also load the system text and initialized heap, initialize the kernel modules, check its disks, and initialize the daemon processes. To provide fast reboot and recovery, we have reduced the time consumed by all of these steps. Below, we list the steps in the reboot and recovery sequence, along with our improvements and the reduction in time consumed by each step.

(1) Retrieve operating system text and data – The first step in the reboot sequence ensures that an undamaged copy of the operating system text and initialized data is in system memory. Reading the Sprite operating system text and initialized heap from disk into memory requires approximately 20 seconds on a Sun 4/280. This is because the program that loads the kernel image from the disk only manages to read about 50 kilobytes per second. Other systems' disk boot programs might be more efficient. We reduce this cost to much less than 1 second, on both the Sun 4/280 and the SPARCstation 2, by using a technique similar to the recovery box. The operating system text lies in a write-protected area of memory, namely the text segment. Upon fast reboot, we simply reuse the text segment already in memory. Because the initialized data (heap segment) is not write-protected, we must do some extra work to preserve a non-writable copy of it. The hard reboot startup code makes a copy of the kernel initialized data in an area of the text segment that it then write-protects. The fast reboot startup code then copies this initialized heap data from its write-protected storage to the correct address in its heap segment. The system initiates a fast reboot sequence by jumping to a special text address and starting execution at that point rather than downloading a new kernel image and starting execution at the very beginning of the text segment. Because the text segment and copy of the initialized heap data are write-protected, no checksum is performed on this data. Preserving the kernel text and initialized heap data in memory also speeds recovery of machines that ordinarily download the kernel image over the network.

(2) Initialize kernel modules – In addition to the initialized heap data stored in a special area of the text segment, we also store information that would otherwise need to be recomputed when kernel modules are initialized, for example, the machine's internet address. Storing these items shortens our module initialization from about 7 seconds to less than 2.

(3) Check disks – We have converted our disks over to use the Log-Structured File System (LFS) [16]. Without LFS, a hard reboot of a file server with 5 gigabytes of storage using a traditional UNIX file system such as the 4.2 BSD Fast File System [13] can take up to 40 minutes to restore the consistency of file system metadata. LFS does not require a file system check (`fsck`) during system initialization, because it leaves its disk-resident file system metadata consistent even after a crash. LFS recovers from failures by rolling forward from a log on disk which is checkpointed at least every 30 seconds. The cost of rolling forward depends on the number and type of I/O operations written to the log since the last checkpoint; two seconds is a conservative estimate of LFS recovery time given current workloads. While LFS does not need a file system check to make its metadata consistent after a failure, it could use one to check that its directory structure has not been damaged by errors. A version of LFS to be included in a future BSD UNIX release [18] uses `fsck` to check the consistency of essential directories on reboot, such as those on the root file system.

(4) Recover distributed state – The Sprite file servers must regenerate the distributed file system state they were using prior to a crash. For 40 clients accessing a single Sun 4/280 file server, this recovery takes more than two minutes depending on how heavily-used the system is at the time of the crash. Using the recovery box on a SPARCstation 2 file server with 20,000 pieces of distributed state, this portion of the recovery time is ten seconds. We chose 20,000 pieces of distributed state for our tests, because this amount is greater, by one-third to one-half, than the number of file handles needed by Sprite's main file server with 40 clients. The ten-second recovery time includes the cost of some disk accesses to retrieve descriptor information such as file permissions and last access times. If we added the necessary descriptor information to the recovery box file handle items, or if we recovered the information in a lazy fashion as needed, then we could eliminate the disk accesses and reduce this portion of our recovery times further. A direct comparison of our measurements for distributed state recovery times with and without the recovery box is not fair, because the Sun 4/280 file server is a slower machine than the SPARCstation 2. We are currently unable to substitute a SPARCstation 2 for our Sun 4/280 file server. However, the recovery box reduces distributed state recovery by at least an order of magnitude, even on the SPARCstation 2.

(5) Kernel and daemon processes – The start-up of internal kernel processes and various system daemons, such as sendmail, inetd and lpd, currently requires about 40 seconds. Although we could potentially store some of the state of necessary processes in the recovery box, we currently just start up the most crucial processes (such as POSTGRES and login) first, consider the system to be "up" at that point, and then start up the other daemons in a lazy fashion. This has reduced the process start-up time for necessary processes to less than 10 seconds.

The total time required for a fast reboot after our improvements is about 26 seconds on a SPARCstation 2. This includes 10 seconds for distributed state recovery, 2 seconds for disk initialization and 14 seconds for the other steps. This compares with the several minutes required for a hard reboot.

The most lengthy step for POSTGRES recovery is reinitializing the server's connections with client applications. Without the recovery box, POSTGRES clients discover failures using timeouts. After the timeout, each client must query the database to find out whether its last transaction committed. The timeout alone requires several seconds since it must allow for worst-case queueing delays before assuming that the DBMS is down.

To measure POSTGRES recovery times, we ran a debit/credit benchmark based on TP1 [2], but to expedite the measurements we used a much smaller database than TP1 requires. A single POSTGRES DBMS managed the database from a Sprite file server. Ten POSTGRES client processes running on a single Sprite client machine generated the transactions. For our experiments, we used a version of the DBMS that was single-threaded. While the DBMS executes a transaction for one client, transactions sent by the others are queued in the DBMS address space. Single-threaded execution means that adding POSTGRES clients increases recovery time (due to client connections) without increasing the overall transaction rate of the DBMS.

Breaking down DBMS recovery into its component parts, we get the following recovery times:

(1) Demand page DBMS code and load system catalogs from disk – We have made no optimizations here so far, although both the DBMS text segment and some parts of the catalogs could be cached in the recovery box. The cost of this initialization is about one second.

(2) Initialize internal memory data structures – POSTGRES must initialize many hash tables and linked lists. Together, all of this initialization only takes 0.4 seconds, so we have made no optimizations here.

(3) Restore consistency of database – Using the POSTGRES storage system, POSTGRES ensures that disk data is always consistent; updates caused by committed transactions are reflected in the database and updates caused by uncommitted transactions are not. As a result, it does not have a step analogous to Sprite's disk check. A conventional DBMS does have such a step and can spend many minutes restoring the consistency of its disk data from a write-ahead log, depending on the length of time between checkpoints and the DBMS transaction rate.

(4) Recover client connections – With a fast reconnect protocol that relies on the recovery box, POSTGRES clients are notified immediately of a DBMS failure and can resubmit lost transactions with a single message exchange. The time to recover 10 client connections using this protocol is less than a second. Other systems have implemented client recovery with add-hoc mechanisms involving several message exchanges, queries of the database, and sometimes human intervention (including the original POSTGRES implementation). The times required by these mechanisms vary widely but all will be longer than a single message exchange per client.

Using the recovery box, total recovery time for POSTGRES and 10 clients (ignoring operating system recovery time) is about six seconds. Most of this cost comes from reloading the database into main memory.

Regular Execution Performance

Using the recovery box has not significantly reduced the regular execution performance of Sprite or POSTGRES. Table 5 shows the breakdown of time required for the Sprite file system recovery box operations. The table gives measurements for the recovery box code itself, for the file system layer that calls the recovery box code, and for the file open and close times seen by a SPARCstation 2 client of the file server, with and without the recovery box. The file open/close measurements include the time for the kernel-to-kernel remote procedure calls between the client and file server. We report measurements in terms of pairs of operations – recovery box item insert/delete operations and file open/close operations.

For a SPARCstation 2 Sprite file server, the time to insert, checksum and delete a file handle item in the recovery box is 28 microseconds on average. The checksum calculations require 4 microseconds per file handle. The open/close test performs two insert operations during file open and performs two delete operations during file close for reasons explained below.

The file system bookkeeping code that calls the recovery box requires more time for an insert/delete operation: 72.3 microseconds on average. In part, this is due to the time to set up the items to insert in the recovery box, but it is also due to a problem in our current implementation. Inserts and deletes of file handle items currently require some extra bookkeeping and a hash table lookup. The hash table maps from file handles to itemNumbers and recovery box item reference counts. Much of this extra code would be unnecessary if we could avoid the mapping by changing the format of the file handle structure in Sprite to include the itemNumber and recovery box reference count. Unfortunately this modification would require recompiling and rebooting the entire Sprite cluster. The resulting outage would affect dozens of irritable graduate students and several aggressive faculty members. We have postponed making these changes until the recovery box prototype has proven itself to be stable.

Operation	Avg time in microseconds	Standard deviation
Recovery box insert/delete with checksum	28	± 0.0
Recovery box insert/delete with no checksum	24	± 0.0
FS insert/delete with checksum	73	± 1.9
FS insert/delete with no checksum	72	± 2.3
Open/close with with recovery box	3616	± 55.0
Open/close without recovery box	3450	± 86.6

Table 5: Sprite recovery box performance. The first two entries give the time to insert and delete a file handle item in the recovery box, with and without a checksum. The second two entries give the time to insert and delete a file handle, including the extra overhead in the file system (''FS'') bookkeeping code. The last two entries give the time to execute a file open/close operation from a client workstation, with and without a recovery box running on the file server. This time includes the kernel-to-kernel remote procedure calls between the client and file server. The results columns give the average and standard deviation of 10 sets of measurements. Each measurement executed 100 iterations of 50 open/close or 50 insert/delete operation pairs. The recovery box already contained 1000 file handle items before the measurements started.

The latency experienced by a client opening and closing a file on a server with a recovery box includes twice the cost of the file system bookkeeping code, because each file open/close pair requires inserting and deleting two file handle structures in the recovery box. One file handle contains data about the file on disk; this is similar to a UNIX inode but also includes distributed cache consistency information. The other handle is a reference to the open file and contains information similar to a UNIX open file table entry. This second handle is called a *stream handle*. The ability of processes in Sprite to migrate to idle machines [8] means that two processes on different machines may share the same reference to an open file. The server maintains the shared file offset in the stream handle and must be able to regenerate this information after a reboot.

Operation	Average time in microseconds	Standard deviation
Recovery box update	55	± 2.3
System call overhead	19	± 0.3
Checksum 92 bytes	10	± 0.0
Copy 92 bytes	8	± 0.1

Table 6: POSTGRES recovery box performance. The first entry shows the time to update a POSTGRES client connection in the recovery box. The last three entries give the major components of this cost – system call overhead, checksum calculation, and copying costs. The results columns give the average and standard deviation of 10 sets of measurements. Each measurement is the average cost of 100,000 operations.

While the recovery box adds a 5% latency to the open/close times seen by a client workstation, the effect on file server throughput is not as large. On average, the main Sprite file server for 40 clients receives two file open/close pairs per second. The current recovery box implementation thus adds, on average, an extra 332 microseconds of server processing per second, or less than a 0.1% increase in processing demand at the server. However, file system activity is bursty, and the server sometimes sees as many as 50 file open/close pairs in a second. During peak activity, the recovery box would thus add 8.3 milliseconds of server processing per second, for a potential 0.8% reduction in server throughput. If we were able to eliminate much of the overhead in the file system bookkeeping code on the file server, the overhead, including increased latency seen by the clients, could be cut in half. We believe this means the recovery box overhead would be acceptable, even in a high performance system.

POSTGRES updates a 92-byte item in the recovery box on each transaction; the entire connection structure is updated even though only the transaction ID changes. Table 6 shows the breakdown of times for POSTGRES recovery box operations. Individual POSTGRES recovery box operations are slower than the ones made by Sprite. POSTGRES must pay the overhead of system calls. Because it is inserting larger objects into the recovery box, the cost of copying data and computing checksums also increases. The operating system is able to use an optimized checksum routine for file handles (with an unrolled loop) while application programs must use a generalized one. POSTGRES could, of course, compute and check its own checksums if the difference in performance were critical. In fact, the measured cost of the recovery box operations is much smaller than the standard deviation between POSTGRES transaction execution times, so there is no reason to further optimize checksum calculation.

Availability and Reliability Impact

Our testbed has shown that the recovery box has acceptable performance, but its impact on system availability and reliability will determine the viability of the technique. We have just finished the implementation and initial test runs of the recovery box. We hope to have it in general use in the next couple of months. After that, collecting data on its effectiveness should take six to twelve months, given our current system failure rate of about once a week. If the recovery box provides fast recovery from our most common failures without reducing the reliability of the system, then we will consider it to be a successful tool.

Conclusion

We can provide high availability in the UNIX environment without great expense or performance degradation by using a recovery box to store state that would otherwise be regenerated from scratch by a system reboot. Failure statistics indicate that memory used for the recovery box is unlikely to suffer corruption due to system failure. The system can fall back to the traditional, slower, recovery path if it detects any problems with the backup recovery data during reboot. This reduces down-time while avoiding the complexity of more sophisticated fault-tolerant techniques. The recovery box concept can be applied to any system in which it is possible to identify and isolate the data structures that (1) are expensive to regenerate from scratch, (2) are small, (3) are updated too frequently to store on disk, and 4) do not have a history of causing system failures. As long as memory requirements do not become outrageous, the recovery box should scale well to large distributed systems, since it substitutes local memory accesses for extra network communication or disk accesses during restart.

Currently, we have only used the recovery box for Sprite and POSTGRES, but we believe that it would be useful in many applications with state that is expensive to regenerate. The internet name servers and programs with TCP connections are examples of applications with distributed state that could be stored in the recovery box. The recovery box might also be an inexpensive way to limit the number of editing changes lost in a vi session due to a system failure. Especially when implemented with non-volatile RAM, the recovery box could help eliminate file system consistency checks in non-LFS file systems. It should also be possible to apply it to user-level servers, such as those running on top of the Mach microkernel [1], and to other distributed application programs.

Acknowledgements

We would like to thank John Ousterhout, Mendel Rosenblum, Margo Seltzer, Jim Mott-Smith, Joe Hellerstein, Mike Olson, Jeff Mogul, and Adam Glass for their very helpful comments on drafts of this paper. Mark Weiser, Keith Bostic, John Wilkes, John Hartman and Wendell Baker all made useful suggestions about the recovery box design. Mike Kupfer quelled periodic panics. The work described here was supported in part by the National Science Foundation under grants CCR-8900029 and MIP-8715235, and the National Aeronautics and Space Administration and the Defense Advanced Research Projects Agency under contract NAG2-591.

References

1. Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M., Mach: A New Kernel Foundation for UNIX Development, *Proceedings of the Summer 1986 USENIX Conference*, June 1986.
2. anon, A Measure of Transaction Processing Power, in *Readings in Database Systems*, Morgan Kaufmann Publishers, 1988, 300-312.
3. Auslander, M., Larkin, D. and Scherr, A., Evolution of MVS, Vol. 25, September 1981.
4. Baker, M. and Ousterhout, J., Availability in the Sprite Distributed File System, *Operating Systems Review 25*, 2 (April 1991).
5. Bartlett, J., A NonStop Kernel, *Proceedings of the 8th Symposium on Operating System Principles*, December 1981.
6. Bhide, A., Elnozahy, E. and Morgan, S., Implicit Replication in a Network File Server, *IEEE Workshop on Management of Replicated Data*, November 1990.
7. Borg, A., Blau, W., Graetsch, W., Herrman, F. and Oberle, W., Fault Tolerance Under UNIX, *ACM Transactions on Computer Systems 7*, 1 (February 1989).
8. Douglis, F. and Ousterhout, J., Transparent Process Migration: Design Alternatives and the Sprite Implementation, *Software -- Practice & Experience 21*, 8 (July 1991).
9. Gray, J., A Census of Tandem System Availability between 1985 and 1990, *IEEE Transactions on Reliability 39*, 4 (October 1990).
10. Jewett, D., Integrity-S2 -- A Fault-tolerant UNIX Platform, Field Failures in Operating Systems, *Digest 21st International Symposium on Fault Tolerant Computing*, June 1991.
11. Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L. and Williams, M., Replication in the Harp File System, *Proceedings of the 13th Symposium on Operating System Principles*, October 1991.
12. Long, D., Carroll, J. and Park, C., A Study of the Reliability of Internet Sites, *Proceedings of the 10th Symposium on Reliable Distributed Systems*, September 1991.
13. Mckusick, M. K., Joy, W. N., Leffler, S. J. and Fabry, R. S., A Fast File System for UNIX, *ACM Transactions on Computer Systems 2*, 3 (August 1984).
14. Mogul, J. C., A Recovery Protocol for Spritely NFS, *To appear in the Proceedings of the USENIX Workshop on File Systems*, May 1992.
15. Ousterhout, J., Cherenson, A., Douglis, F., Nelson, M. and Welch, B., The Sprite Network Operating System, *IEEE Computer 21*, 2 (February 1988).
16. Rosenblum, M. and Ousterhout, J., The Design and Implementation of a Log-structured File System, *Proceedings of the 13th Symposium on Operating System Principles*, October 1991.
17. Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D. and Lyon, B., Design and Implementation of the Sun Network File System, *Proceedings of the the Summer 1985 USENIX Technical Conference*, June 1985.
18. Seltzer, M. I., Personal Communication, April 1992.
19. Stonebraker, M. and Rowe, L., The Design of POSTGRES, *Proceedings of the 5th ACM SIGMOD Conference*, June 1986.
20. Stonebraker, M., The POSTGRES Storage System, *Proceedings of the 13th International Conference on Very Large Data Bases*, September 1987.
21. Sullivan, M., *Unpublished survey of software errors reported in 4.1 and 4.2BSD UNIX*, 1990.
22. Sullivan, M. and Chillarege, R., Software Defects and Their Impact on System Availability -- A Study of Field Failures in Operating Systems, *Digest 21st International Symposium on Fault Tolerant Computing*, June 1991.
23. Webber, S. and Beirne, J., The Stratus Architecture, *Digest 21st International Symposium on Fault Tolerant Computing*, June 1991.
24. Welch, B. and Ousterhout, J. K., Pseudo Devices: User-Level Extensions to the Sprite

File System, *Proceedings of the the Summer 1988 USENIX Technical Conference*, June 1988.

Author Information

Mary Baker is a Ph.D. student in computer science at the University of California at Berkeley. She currently works as a research assistant on the Sprite network operating system project directed by Prof. John Ousterhout. Her interests include operating systems, distributed systems, computer architecture, and raising the cholesterol levels of friends and colleagues. Her email address is mgbaker@cs.berkeley.edu.

Mark Sullivan will receive his Ph.D. in computer science from the University of California at Berkeley in Summer 1992. He currently works as a research assistant on the POSTGRES project under the direction of Prof. Mike Stonebraker. His interests include database management systems, fault tolerance, operating systems and travel to exotic countries. His email address is sullivan@cs.berkeley.edu.

Both authors can be reached via U. S. Mail at the Computer Science Division, 571 Evans Hall, U. C. Berkeley, Berkeley, CA 94720.