Integrating Checkpointing with Transaction Processing

Y. M. Wang P. Y. Chung Y. Huang E. N. Elnozahy *

Abstract

Many industrial applications consist of different components that were designed and implemented separately. Software reuse and component software architecture often dictate this style of building applications out of existing modules. In such an application, some modules may use databases to store critical data, while other components may use regular files or volatile memory either by design or to improve performance. Therefore, global coordination is necessary for maintaining data consistency and recoverability across multiple datastores. We describe a way of integrating checkpointing with transactions to ensure consistency and recoverability of data stored outside the databases. Our approach transforms an existing checkpoint library into a transactional resource manager that can participate in such a global coordination.

1 Introduction

In this paper, the term "checkpointing" refers to the action of recording critical memory and file state at a given point of program execution on stable storage [17]. One question is commonly asked by software developers of client/server applications: *"If I already use database transactions for data persistence, why do I need checkpointing?"* Indeed, if an application can always store all state changes into a database at the end of each transaction, the ACID property protects the data. However, not all applications can be implemented in this manner. For example, an application may need to reuse existing data processing tools which only operate on regular Unix files. In the next section, we describe several client/server applications from the telecommunication industry that store their state changes in files and volatile memory, in addition to databases. This additional storage is often necessary because the application consists of different components that were designed and implemented separately. As component software architecture becomes more popular [12, 14], more applications are likely to follow this style. Additionally, storing information outside a database may sometimes be necessary to improve performance. In these applications, checkpointing is an efficient technique to provide recovery for critical data that are stored in regular files or volatile memory outside the control of the database system.

Storing application state in multiple datastores raises the issue of global data consistency. Each datastore needs to be managed by a transactional resource manager which can participate in a global coordination protocol [6] to ensure that date updates are either all committed or all aborted. Usually, each resource manager is also responsible for providing concurrency control for multiple applications accessing the datastore at the same time. In this paper, we focus on a special case where the files and volatile memory updated by an application are not shared by any other application. This allows a simple implementation of transactional resource managers on a per-application basis in the form of checkpoint libraries linked with the application.

2 Example Applications

In our experience with telecommunication client/server applications, we have observed that the following examples require more than one datastore to save their states.

- When a caller leaves a message in a voice mail system, a Unix tool samples the voice and stores it as a regular file. A database that stores the callee's records and mailboxes is updated to reflect the call and include the name of the voice sample file. In this example, mailboxes are stored in a database to support user queries. The voice samples are not stored in the database because they are generated by an existing Unix tool. Also, a preliminary implementation to store the voice files in a commercial database system showed a 300% performance degradation. Thus, transactional support for the voice files is necessary to ensure their consistency with the mailboxes.

*Y. M. Wang is with AT&T Labs. P. Y. Chung and Y. Huang are with Bell Labs, Lucent Technologies. E. N. Elnozahy is with Dept. of Computer Science, Carnegie Mellon University. This work was done while this author was a consultant at Bell Laboratories.

- In the accounting process of a telecommunications operations system, a transaction typically involves changes to a database and several Unix files. The database contains user account information. The files contain information such as passwords and services that are needed by other Unix applications such as the login and inetd processes. Again, in this example, part of critical state information is stored in regular files so that they can be directly used by other existing applications. Transactional support for files is then required for ensuring consistency.

- In a telephone trouble ticket system, a transaction typically involves updating customer information and repair activity status in a database, as well as updating a data structure in memory which has an entry for each trouble ticket. The data structure is shared by two processes for inter-process communication: one process handles user inputs and the other handles connection to a switch and test of phone circuits. Since the shared data structure contains temporary data that is frequently accessed and updated by the two processes, its persistence is provided by a simple checkpointing mechanism instead of more expensive database operations. Transactional support for the checkpointing mechanism is then necessary to ensure system-wide consistency.

- A network switch system is responsible for assigning trunks and recording usages for each phone call. A transaction in such a system typically involves updating three pieces of critical state information. The start time and end time of each phone call are stored in a database for billing purpose. The status of the trunks are recorded in a set of hardware registers. For each trunk, the usage information and indices to current user records are stored in a data structure in memory: an entry is generated by each call-setup transaction and later deleted by the corresponding call-hangup transaction. In this example, permanent information such as billing data is stored in a database, while temporary information such as trunk status and assignments are stored in memory. The trunk information needs to be made persistent through a checkpointing mechanism because it is used by call-hangup transactions to tear down call connections. Again, the checkpointing mechanism needs to be able to participate in a global coordination so that the temporary data can be consistent with the permanent data.

3 Global Transaction Processing

The examples in the previous section show that it is important to support transactions that update multiple datastores including volatile memory, files and databases. In particular, if a transaction is aborted and the database updates are rolled back, updates to volatile memory and files should also be rolled back. To provide ACID properties for these transactions, we need the concept of *global transaction processing* [6]. In the rest of this paper, we base our discussions on the following simplified model of global transaction processing, which consists of four main components: clients, a server application program, a number of resource managers, and a transaction manager. A client sends a request to the server application program. The program starts a transaction by calling begin_transaction to notify the transaction manager, which then invokes the start() function of each resource manager. The application program performs updates on various resources directly using APIs supplied by resource managers. In normal situations, the application program calls commit_transaction at the end. The transaction manager then invokes a two-phase commit protocol [13] as follows: in the first phase, the transaction manager calls the `prepare()` function of each involved resource manager. Each resource manager returns the status VoteCommit if it can commit; otherwise, it returns VoteAbort. The transaction manager decides to commit the transaction if it receives VoteCommit from all resource managers; otherwise, it decides to abort the transaction. In the second phase, it calls either the commit() or rollback() function of each resource manager based on that decision.

3.1 Implementing Checkpoint Libraries as Transactional Resource Managers

Libft [9] is a checkpoint library for specifying and storing critical memory data. Libfcp – the persistent state checkpointing part of libckp [17] – is a library for undoing file updates upon a rollback. Both libraries were originally designed for applications without transaction semantics. The main contribution of this paper is to consider critical memory and files as two private datastores accessed by a transaction processing application, in addition to other shared datastores such as databases. As pointed out earlier, when a transaction processing application "owns" the volatile memory and files that it is updating, resource managers for these two datastores do not need to handle concurrency control and therefore can be implemented on a per-application basis. We next describe how to extend libft to a transactional critical-memory manager libft_RM and to extend libfcp to a transactional Unix-file manager libfcp_RM.

3.1.1 Transactional critical-memory manager libft_RM

Libft provides two function calls critical() and ftmalloc() for specifying critical static variables and dynamically allocated memory, respectively. It also provides a checkpoint() call for saving the specified data. The library is linked with the server application program which initiates transactions. To serve as the resource manager for critical volatile data, the library libft_RM enhances libft with a list of callbacks to support transaction semantics, including ft_start(), ft_prepare(), ft_commit(), ft_rollback(), ft_recover() and a few others.

Each new transaction generates an ft_start() call to inform libft_RM of the id of the transaction that it is now joining. When the application wants to commit the transaction, the transaction manager calls ft_prepare() to ask libft_RM to prepare to commit the volatile state change. In the ft_prepare() routine, libft_RM invokes the checkpoint() call to save all critical data in a temporary checkpoint file, records a Ready record in the log and returns VoteCommit. After the transaction manager makes the decision to commit, it calls ft_commit() which then overwrites the previous checkpoint file with the temporary file, and logs a Commit record. (If archive recovery is supported, the previous checkpoint file should also be retained.) If a decision to abort is made at any point, the routine ft_rollback() is called to discard the temporary file if it exists and restores the latest committed checkpoint.

Upon a failure, the libft_RM in the restarted application identifies itself to the transaction manager which then invokes ft_recover to ask libft_RM for in-doubt transactions. If libft_RM sees a Ready record without the corresponding Commit record, it returns the id of that transaction. Based on its knowledge of the outcome of that transaction, the transaction manager invokes ft_commit() or ft_rollback() accordingly. Finally, libft_RM restores the latest committed checkpoint to finish recovery.

3.1.2 Transactional Unix-file manager libfcp_RM

In this section, we describe how to implement per-application transactional file updates. First, we would like to point out that libfcp_RM is not used to protect the internal integrity of a file system. It is used by applications to make a sequence of file updates atomic within a transaction. For instance, an application can have the following structure:

```
begin_transaction();
...
fprintf(fp, "User name %s\n", name);
fprintf(fp, "Phone number %d\n", id);
fwrite(billing_record, 80, 1000, fp);
...
end_transaction();
```

By using libfcp_RM, the three file operations either all succeed or none of them takes effect.

Libfcp_RM adopts the in-place update approach [18]. Its status can be represented by the state diagram shown in Fig. 1. Initially, it is in STEADY state until the transaction manager invokes fcp_start() and the application starts updating the files through system calls such as write() and rename(). It enters the ACTIVE state when the first update occurs. Every file update operation is intercepted by libfcp_RM, and an idempotent undo operation is added to the UNDO log. For instance, before the application appends a byte to a file called myfile, libfcp_RM takes its original file size s_o and generates the undo operation "truncate myfile to size s_o". To ensure that any uncommitted updates can be undone, the UNDO log has to be flushed to the disk before the actual update occurs.

In the ACTIVE state, the application may abort the transaction, which causes fcp_rollback() to be invoked by the transaction manager. In this case, libfcp_RM adds an Abort record to the UNDO log and enters the ABORT state. By applying the UNDO operations in a backward order, libfcp_RM restores the files to their original contents and sizes before the transaction. Upon completion, it discards the UNDO log and returns to the STEADY state.

Normally, in the ACTIVE state, the application would commit the transaction and the transaction manager starts a two-phase commit protocol by calling fcp_prepare(). In the fcp_prepare() routine, libfcp_RM adds a Ready record to the end of the UNDO log, returns VoteCommit, and enters the READY state. If the transaction manager decides to commit, it calls fcp_commit() in which libfcp_RM simply discards the UNDO log to complete the transaction and goes back to the STEADY state. If the decision is to abort, the same fcp_rollback() described previously is executed.

A process failure can occur at any stage. When libfcp_RM is back on-line, it examines the UNDO log to determine at which state the failure happened. If the UNDO log does not exist, the failure must have occurred in the STEADY state. Since the files are in non-volatile storage, no actions need to be taken. If a Ready

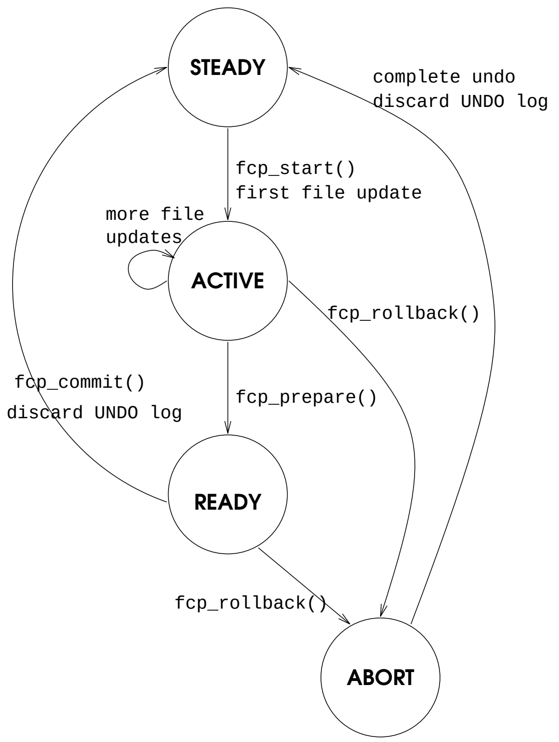

Figure 1: Transactional file operation state diagram

record appears at the end of the UNDO log, this means that the last transaction is in-doubt and its outcome should be resolved by communicating with the transaction manager. In other cases, the process must have failed in either ACTIVE or ABORT state, and the transaction should be aborted by executing fcp_rollback(). Note that fcp_rollback has to apply all the undo operations, and it is possible that another failure could occur during that process. Therefore, it is important to make undo operations idempotent so that they can be applied again without affecting correctness.

3.1.3 The interactions between transaction and resource managers

In some applications, the application program itself acts as the transaction manager and so can directly invoke the callbacks. In other applications where transaction manager runs as a separate process, the stubs for these callbacks are installed with the transaction manager and, when called, in turn invokes the callbacks. If the database involved also supports these callbacks, the standard two-phase commit protocol described previously can be applied. Otherwise, linear commit [6] should be used: transaction manager first invokes ft_prepare() and fcp_prepare(). If they both returns VoteCommit, then the transaction manager transfers commit authority to the database by asking it to commit. If the database successfully commits, ft_commit() and fcp_commit are called; otherwise, ft_rollback() and fcp_rollback() are called.

4 Related Work

The volatile memory checkpointing mechanism discussed in this paper is similar to the data persistence service provided by various object-oriented systems. For example, in the Dome (Distributed Object Migration Environment) project, variables declared as special Dome types are saved when a checkpoint call is made [15] and are restored as they are declared in a restarting Dome environment. In CORBA (Common Object Request Broker Architecture), the Persistent Object Service (POS) allows an object to be made persistent by inheriting a particular class that provides an interface for clients to control the object's persistence [16]; the object must also provide a mechanism for externalizing its state. In Microsoft's ActiveX technologies, *Structured Storage* provides interfaces for objects to store and retrieve their persistent data, and the IPersist* interfaces allow clients to control objects' persistence [2]. The product ObjectStore Object Persistence Engine, developed by Object Design Inc., is a tool that provides APIs for specifying and storing persistent objects in C++ and Java applications. Compared to these techniques, libft_RM is a lower-level approach that works with memory locations instead of objects. The advantages are that it can be used with both C and C++ applications, and it can handle critical data inside imported libraries without source code. The main disadvantage is that it does not support the concept of data encapsulation.

The implementations described previously can be modified to provide the complete list of callbacks specified in the standard XA interface [3, 6]. They can then be used with any transaction monitors that support the XA interface, such as BEA Tuxedo [1] and Transarc Encina [4]. The Session-XA product [7] from Gresham Computing allows an application program to save context data for a transaction session, i.e., a series of transactions carried out by a particular user over a period of time. At the end of a transaction, the application program can pack all session context data into a record and invoke a Session-XA API to save it. At the start of the next transaction of the same session, another API call is made to reload the data. In contrast, libft_RM is mainly used for maintaining the entire context data of an application program instead of a particular session; also, a single checkpoint() call at the end of a transaction automatically pack and save all declared critical data into a checkpoint file.

5 Summary

We used several applications drawn from the telecommunication industry to demonstrate the need for storing critical state information in multiple datastores in transaction processing applications. In general, permanent data that require a strong support for concurrency control and flexible queries are stored in databases. Large amount of information that needs not be queried and information that needs to be shared by other system utilities or application tools are stored in regular files. Non-permanent data structures that are frequently accessed and updated are stored in volatile memory. To support the global coordination required for ensuring data consistency across these multiple datastores, implementations that transform existing checkpoint libraries into per-application transactional resource managers were described. While the critical-memory manager libft_RM checkpoints the entire critical data at the end of each committed transaction, the Unix-file manager libfcp_RM adopts the approach of in-place updates with undo logs to checkpointing files.

References

[1] J. M. Andrade, M. T. Carges, T. J. Dwyer, and S. D. Felts. *The Tuxedo System.* New York: Addison-Wesley, 1996.

[2] D. Chappell. *Understanding ActiveX and OLE.* Redmond, Washington: Microsoft Press, 1996.

[3] G. Chen. Distributed transaction processing standards and their applications. *Computer Standards and Interfaces*, (17):363–373, 1995.

[4] Transarc Corporation. Encina product overview. http://www.transarc.com/.

[5] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebreaker, and D. Wood. Implementation techniques for main-memory database systems. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 1–8, June 1984.

[6] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* San Mateo, CA: Morgan Kaufmann Publishers, 1993.

[7] Gresham Computing. Session-XA for Tuxedo. http://www.gresh.com/.

[8] R. Hagmann. A crash recovery scheme for a memory resident database system. *IEEE Trans. Comput.*, C-35(9):839–843, September 1986.

[9] Y. Huang and C. Kintala. Software implemented fault tolerance: Technologies and experience. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 2–9, June 1993.

[10] J. E. Moss. Checkpoint and restart in distributed transaction systems. In *Proc. of the 3rd Symp. on Reliability in Distributed Software and Database Systems*, pages 85–89, 1983.

[11] P. Ng. A commit protocol for checkpointing transactions. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 22–31, 1988.

[12] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide.* New York: John Wiley & Sons, 1996.

[13] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, Englewood Cliffs, N.J., 1991.

[14] D. Rogerson. *Inside COM.* Redmond, Washington: Microsoft Press, 1997.

[15] E. Seligman and A. Beguelin. High-level fault tolerance in distributed programs. Tech. Rep. No. CMU-CS-94-223, Dept. of Computer Science, Carnegie Mellon University, 1994.

[16] J. Siegel. *CORBA Fundamentals and Programming.* New York: John Wiley & Sons, 1996.

[17] Y. M. Wang, Y. Huang, K. P. Vo, P. Y. Chung, and C. Kintala. Checkpointing and its applications. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 22–31, June 1995.

[18] W. E. Weihl. Transaction-processing techniques. In *Distributed Systems, Ed. S. Mullender*, pages 329–352. Addison-Wesley, Reading, Massachusetts, 1993.