Libra: A Library for Reliable Distributed Applications

Jinsong Ouyang and Gernot Heiser
School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia
{jinsong,gernot}@cse.unsw.edu.au
Telephone: +61-2-385-4011
Facsimile: +61-2-385-5995

Abstract

This paper describes libra, a library to support efficient reliable distributed applications. libra is designed to meet two objectives: to simplify the development of reliable distributed applications, and to achieve fault-tolerance at low run-time cost. The first objective is met by the provision of fault-tolerance transparency and a simple, easy to use high-level message passing interface. Fault-tolerance is provided to applications transparently by libra and is based on distributed consistent checkpointing and rollback-recovery integrated with a user-level network communication protocol. The second objective is met by the use of protocols which minimise communication overhead for taking a consistent distributed checkpoint and catching messages in transit, and impose low overhead in terms of running times. The paper presents measurements backing up these claims.

Keywords: *fault-tolerance, checkpointing, rollback-recovery, message passing, reliable distributed applications*

1 Introduction

In recent years the abundance of networked computers has established distributed computing as a mainstream paradigm suitable to achieve high utilisation of available computing resources. In a setting consisting of a potentially large number of computers connected, by an unreliable network, fault-tolerance becomes a major issue. The challenge is to incorporate fault-tolerance into applications at low cost, in terms of both, runtime performance and of programming effort required to construct the application software. The combined complexity of dealing with network communications **and** fault-tolerance makes the development of efficient reliable distributed software difficult.

In order to address these issues we have developed a library called libra which transparently provides fault-tolerance to distributed applications. Libra implements our distributed consistent checkpointing and rollback-recovery protocols, including a user-level network communication protocol [1]. The library exports high level message-passing primitives which hide the complexity of fault-tolerant network communications from the application. This approach, besides significantly simplifying the application programmer's task, allows us to interweave message passing tightly with distributed checkpointing and rollback-recovery, and thus implement them efficiently.

A variety of approaches to checkpointing and rollback-recovery have been proposed in the literature. Some are based on *independent checkpointing* ' [2–4] while others use *consistent* or *coordinated checkpointing* [5–10]. Our approach [1] implemented in libra belongs to the latter category, and is distinguished from other consistent checkpointing schemes by the following features:

- minimum communication overhead (see Sect. 4.1) for constructing a distributed consistent checkpoint and catching messages in transit. A message is *in transit* if it was sent within the previous checkpoint interval and is received within the current checkpoint interval.
- tolerance to message losses due to site failures or unreliable non-FIFO networks, and
- reduced run-time overhead (see Sect. 4.2), enhancing the efficiency of reliable distributed applications.

The same motivations drove the work of other researchers [11–13] who developed reusable components for reliable systems. Libra differs from these not only by the underlying mechanisms, but also by offering fault-tolerance transparency together with a simple, high-level message-passing interface.

In the following section we outline our mechanisms for providing fault-tolerance to distributed applications. In Section 3 we summarise the architecture of libra and some aspects of its implementation. Section 4 presents performance results to demonstrate the efficiency of the library, and Section 5 contains our conclusions.

2 Checkpointing and Rollback-Recovery

Our model assumes a distributed environment where nodes fail by stopping and which is based on an unreliable network which may loose or reorder messages.

In our distributed consistent checkpointing protocol, each distributed checkpoint is uniquely identified by an increasing *checkpoint sequence number* (CSN) [1,6,8–10] and a *status bit*. The status bit on a node is set when the local checkpoint is part of the latest committed distributed checkpoint. libra tags each normal (i.e. application-level) message with the current CSN and status bit of the sender. Synchronised by the coordinator, a variant of a two-phase commit protocol is employed, where the second phase proceeds lazily and therefore does not require extra messages. If any message is received with a CSN greater than the local one, a local checkpoint is taken. If the message's CSN is less than the local one, the message was in transit during the checkpoint and must be logged. If the CSNs agree but the the message's status bit is set while the local one is not, the local checkpoint is committed. Communication overhead for a distributed checkpoint is thus reduced to that of systems using a one-phase commit, while stable storage is utilised more efficiently, as previous checkpoints can be discarded once the present checkpoint is committed.

To prevent message loss following a rollback, messages in transit during a distributed checkpoint need to be discovered and logged as part of the current checkpoint. While other approaches [5, 9] require additional messages to catch such messages in transit, we avoid this overhead by integrating the checkpointing algorithms with the network communication protocol [1]. We employ a *user-level reliable transmission protocol* (URTP), which cooperates with the checkpointing algorithms (e.g. it does the logging of messages in transit). The protocol implementation uses threads to provide non-blocking asynchronous communication.

A second novelty is the use of an *acknowledgement counter* (AC) to record the number of message packets originating from the local node between two checkpoints and which have not been acknowledged. Each node maintains two ACs: PAC and CAC. An AC is incremented by the number of packets used when sending a message, and is decremented by the same amount once the last package of that message has been acknowledged. The PAC is updated while there exists no uncommitted local checkpoint, otherwise the CAC. On commit, the CAC becomes the PAC and the CAC is set to zero. The local node does not inform the coordinator of the local checkpoint having been taken until its PAC becomes zero (indicating that all messages originating at that node between the last two checkpoints have been logged as required). This guarantees that all messages in transit have been logged once the coordinator commits. The coordinator assumes failure and initiates a rollback-recovery if some nodes fail to respond within a timeout interval. Other failures, such as unacknowledged messages, are detected by the URTP.

Rollbacks are also uniquely identified, by a *recovery sequence number* (RSN), to avoid *livelocks* and maximise parallelism during recovery. The RSN is also tagged on every message. A one-phase commit protocol is used for the distributed rollback. If a message (either a specific rollback request or a normal message) is received with a RSN greater than the local RSN, a local rollback-recovery is performed and an acknowledgement sent to the coordinator. If an RSN is received which is less than the local one, the message was sent before the sender performed its rollback and is therefore discarded.

3 Library architecture and implementation

We have implemented libra on an ethernet network of Sun workstations running SunOS 4.1 (a port to Solaris 2.5 is in progress). Here we outline the architecture of libra and some aspects of the implementation. More details can be found in a forthcoming technical report [14].

Distributed applications use threads and libra's message-passing and memory allocation primitives; fault tolerance is then automatically provided by the library. Table 1 shows the library interface (functions for configuring parameters, such as the number of participating nodes, checkpoint frequency and timeout intervals, have been omitted from the table for simplicity).

Message passing	*Memory*	*Initialisation*
ft_send(char *msg, int size, int dest) ft_recv(char *msg, int size, int *sender)	char *malloc(int size) free(char *addr)	ft_init(int my_id, ...)

Table 1: Functionality provided by libra

The functions ft_send and ft_recv provide basic message passing, threads are created by the library to perform the actual send operation without blocking the application. In order to detect missing, duplicate, and other unexpected packets, libra maintains on each node a *sending sequence number vector* (SSV) as well as a *receiving sequence number vector* (RSV), both of length n, the number of nodes in the distributed system. When sending a packet to node j, libra increments SSV[j] on the sender i, and tags this on the packet. Whenever a packet is received, the SSV[j] tagged on the packet is compared with the local value of RSV[i]. If these agree, the packet is valid and RSV[i] is incremented. If SSV[j] = RSV[i]-1, a duplicate packet has been received and is ignored, as are unexpected packets recognised by other cases of non-matching sequence numbers. Initiating and committing checkpoints and rollbacks, and handling the message logs, are performed transparently by libra (through background threads and the application's calls to ft_send and ft_recv).

The functions malloc and free, exported by libra, replace the normal memory management functions. Their use by the application ensure that all application data are checkpointed.

The application needs to call ft_init so libra can initialise its internal data structures. This call, when executed on the coordinator node (node 0) will create a coordinator thread, cp_coor, which initiates distributed checkpoints, and commits or aborts them. On other nodes a cp_node thread is set up. This thread performs local checkpoints, as requested by the coordinator. On the coordinator node, a separate thread rr_coor is responsible for rollback-recoveries; this thread initiates, coordinates and commits or aborts the recovery as appropriate. Local recovery action is performed by thread rr_node running on non-coordinator nodes.

Checkpoints save the full state of the application, comprising of register set, stacks, static and heap data. *Copy-on-write* and *incremental checkpointing* [6] techniques are used to reduce the checkpointing latency as well as memory and disk use.

It should be clear from the above that libra makes development of reliable distributed applications easy, as a simple, high-level message passing interface is provided and fault tolerance is completely transparent to the user. The resulting applications are efficient as the library minimises overhead as much as possible.

4 Performance evaluation

We have analysed the performance of libra with respect to communication and running time overheads. For this purpose, we chose three long-running and large size message-passing applications with quite different communication patterns: CST, a program for maintaining a balanced concurrent search tree (a $2^{B-2} - 2^B$ search tree) [15], QSORT, a distributed quicksort implementation, and FFT, the Fast Fourier Transform of between 64k and 2M data points. CST exchanges many small messages, while FFT passes infrequent but large messages; QSORT is somewhere in between these two extremes. Each application is distributed by using one or more *server* and a number of *client* processes.

4.1 Communication overhead

We classified the existing approaches based on consistent checkpointing into five categories according to how consistent checkpoints are taken and how messages in transit are caught. We chose a typical representative from each category to compare communication overhead. The choices are: 1. a variant of Chandy and Lamport's one-phase commit snapshot algorithm [5] for non-FIFO systems [9]; 2. Mattern's one-phase commit snapshot algorithm with the *deficiency counting* termination detection method (DCTD) for catching messages in transit [9]; 3. Mattern's one-phase commit algorithm with the *vector counter principle* (VCP) for catching messages in transit [9]; 4. Elnozahy *et al.*'s two-phase commit algorithm [6] (which does not catch messages in transit); 5. our algorithm as described above [1]. Because we are just comparing the communication overheads at this stage, there was no need to implement these five algorithms in a real network. We therefore simulated the three programs (CST, QSORT, and FFT) on a single machine, using the light-weight process library

provided in Sun OS 4.1. Each program was implemented in five versions, one for each of the checkpointing algorithms examined. The message overheads for the five algorithms for the three benchmarks are shown in Table 2.

	Benchmark statistics			Communication overheads				
Program	# processes	# chkpt	# msg_tran	Chandy	DCTD	VCP	Elnozahy	URTP & AC
CST	110	9	114	435	332	288	436	218
QSORT	101	9	100	400	300	321	400	200
FFT	65	5	45	256	173	192	256	128

Table 2: Communication overheads due to checkpointing in terms of average number of messages required to perform one consistent checkpoint. The table also gives the number of processes (= simulated nodes), the number of checkpoints performed, and the average number of messages in transit during one checkpoint.

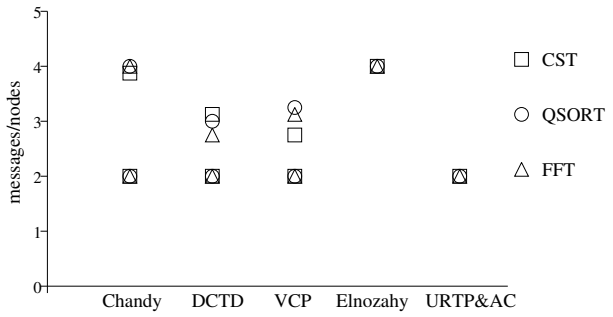

Figure 1: Average per node communication overhead due to one checkpoint. The lower points represent the overhead for checkpointing only, while the upper points include the overhead for catching messages in transit. Elnozahy does not catch messages in transit (hence no upper points), while our algorithm does not require any extra messages for catching messages in transit.

Figure 1 shows the normalised message costs for the five checkpointing algorithms (averaged over several runs). The pure checkpointing overhead (lower points) is the same in all cases, except Elnozahy, where it is twice as high as for the other algorithms. The additional overhead for catching messages in transit differs significantly between the algorithms: in Chandy and Lamport's algorithm, the overhead depends on the number of incident output channels for each process; with the DCTD method, the number is equal to the number of messages in transit during a checkpoint; with the VCP based mechanisms, the number depends on the number of processes which receive messages in transit during a checkpoint. Elnozahy *et al.* did not address this issue, hence no data are available. Our algorithm (URTP & ACs) does not cause any further message passing for catching messages in transit, and hence exhibits the lowest communication overhead among the investigated methods. The Figure also shows that the communication overhead is not much affected by the type (communication pattern) of distributed application.

4.2 Time overhead

In order to determine the runtime overhead imposed by libra we implemented the applications CST, QSORT, and FFT on top of libra. The programs were run on a network of four Sun workstations; all I/O was handled by a single NFS file server. For CST a larger network was simulated by running four "parallel" processes on each machine.

Table 3 shows the overhead as a function of checkpointing frequency. All times are averages of three runs on an otherwise essentially empty system. As we measured wallclock times, there are several different contributions to the total overhead: 1. I/O cost of writing checkpoints, 2. communication cost of network (NFS) I/O, 3. communications overhead resulting from libra maintaining checkpoint consistency, and 4. the increase in page fault handling cost resulting from copy-on-write and incremental checkpointing. The second of these could be avoided by writing checkpoints to a local disk. Still,

Program	No checkpoint	10-min Interval		5-min Interval		2-min Interval	
	Time (min)	Time (min)	Overhead (%)	Time (min)	Overhead (%)	Time (min)	Overhead (%)
CST	23.4	23.5	0.5	24.4	4.5	24.8	6.5
QSORT	62.2	64.4	3.5	65.6	5.5	67.3	8.3
FFT	59.3	60.4	1.9	61.5	3.8	64.8	9.4

Table 3: Time overheads as a function of checkpoint frequency

Program	State size	10-min Interval		5-min Interval		2-min Interval	
		Ckpt size		Ckpt size		Ckpt size	
	[MB]	[MB]	[%]	[MB]	[%]	[MB]	[%]
CST	2.3	0.5	21.7	0.9	39.1	0.6	26.1
QSORT	8.6	3.4	39.5	2.5	29.1	1.2	13.9
FFT	8.0	4.5	56.3	4.5	56.3	4.5	56.3

Table 4: Average size of a checkpoint for each application.

the overhead is generally quite low, below 10 % even with the shortest checkpointing interval where over 30 checkpoints were written.

It is worth noting that lower overheads have been reported before [6,16], even though message overheads were higher in those projects. We believe that this is due to the following reasons: 1. those projects used special-purpose multicomputer architectures whose operating systems had been optimised for supporting parallel applications while libra runs on a general-purpose network operating system; 2. in their experiments, no mechanisms were used to catch messages in transit, and the corresponding message logging overhead was therefore not included; 3. libra is implemented on top of Sun OS 4.1 light-weight process library which has no kernel support for threads. As a result, in the present implementation user threads are blocked when a write system call is performed during checkpoints. This even happens when the so-called "non-blocking I/O library" is used. The latter problem will be resolved with the port of libra to Solaris.

Statistics of checkpoint sizes are presented in Table 4. The *state size* column represents the total size of the distributed program state of the applications, i.e. the total amount of data to be checkpointed. This is also equal to libra's memory overhead for implementing fault-tolerance. The remaining columns give the average total size actually needed for each checkpoint with respect to different checkpoint intervals, both absolute and in terms of the size of the program state. The beneficial effects of using incremental checkpointing are obvious.

5 Conclusions

We have presented libra, a library for implementing reliable distributed applications. We have explained how fault tolerance is provided transparently to applications communicating via a simple, high-level message-passing interface, making application development easy. We have shown that the overhead imposed by the library compares favourably to other approaches.

References

[1] J. Ouyang and G. Heiser. Checkpointing and recovery for distributed shared memory applications. In *Proceedings of the 4th International Workshop on Object Orientation in Operating Systems*, pages 191–9, Lund, Sweden, August 1995. IEEE.

[2] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11:462–491, 1990.

[3] S. W. Smith, D. B. Johnson, and J. D. Tygar. Completely asynchronous optimistic recovery with minimal rollbacks. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, pages 361–370, June 1995.

[4] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[5] K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[6] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[7] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13:23–31, January 1987.

[8] K. Li, J. F. Naughton, and J. S. Plank. Checkpointing multicomputer applications. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 66–75, September 1991.

[9] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18:423–434, August 1993.

[10] J. S. Plank and K. Li. Ickp—a consistent checkpointer for multicomputers. *IEEE Parallel and Distributed Technology*, 2(2):62–67, 1994.

[11] Y. Huang, C. Kintala, and Y. M. Wang. Software tools and libraries for fault tolerance. *IEEE Bulletin of the Technical Committee on Operating Systems and Application Environments*, 7(4):5–9, 1995.

[12] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under UNIX. In *Proceedings of the 1995 Winter USENIX Conference*, pages 213–223, January 1995.

[13] J. S. Plank, M. Beck, and G. Kingsley. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Bulletin of the Technical Committee on Operating Systems and Application Environments*, 7(4):10–14, 1995.

[14] J. Ouyang. Architecture and implementation of libra, a library for reliable distributed applications. Technical report, University of NSW, University of NSW, Sydney 2052, Australia, 1996. To appear.

[15] A. Colbrook, E. Brewer, C. Dellarocas, and W. Weihl. An algorithm for concurrent search trees. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume 3, pages 138–141, August 1991.

[16] G. Cabillic, G. Muller, and I. Puaut. The performance of consistent checkpointing in distributed shared memory system. In *Proceedings of the 14th Symposium on Reliable Distributed Systems*, pages 96–105, October 1995.