BRRL: A Recovery Library for Main-Memory Applications in the Cloud

Tuan Cao, Benjamin Sowell, Marcos Vaz Salles,
Alan Demers, Johannes Gehrke
Cornell University
Ithaca, NY 14853, USA
{tuancao, sowell, vmarcos, ademers, johannes}@cs.cornell.edu

ABSTRACT

In this demonstration we present BRRL, a library for making distributed main-memory applications fault tolerant. BRRL is optimized for cloud applications with frequent *points of consistency* that use data-parallelism to avoid complex concurrency control mechanisms. BRRL differs from existing recovery libraries by providing a simple table abstraction and using schema information to optimize checkpointing. We will demonstrate the utility of BRRL using a distributed transaction processing system and a platform for scientific behavioral simulations.

Categories and Subject Descriptors

H.2.2 [**Information Systems**]: Database Management—*Recovery and restart*

General Terms

Performance, Reliability

1. INTRODUCTION

In order to provide low latency and high throughput, an increasing number of applications, from online games to OLTP applications and key-value storage systems, are storing their state entirely in main memory. However, these applications must still be made durable and typically use some combination of checkpointing and logging to write their data to stable storage. For applications that are not based on traditional databases or do not need all of the ACID properties, this approach requires developers to write and maintain complex recovery logic themselves. In this demonstration, we present the Big Red Recovery Library (BRRL), a library to automate the process of checkpoint-recovery for distributed main memory applications. BRRL allows developers to easily make their applications fault tolerant without having to write custom code.

Unlike existing checkpointing libraries such as Libckpt [5] and PORCH [6] which are designed solely for single-node applications, BRRL is primarily intended for distributed applications deployed on the cloud or other large shared-nothing clusters. Many of these applications have embraced data-parallelism in order to scale to a very large number of nodes and are frequently designed with limited or periodic synchronization in place of traditional lock-based concurrency control. BRRL takes advantage of these *points of consistency* to take efficient checkpoints without interfering with application performance [3]. Additionally, BRRL includes an efficient and low-latency implementation of logical logging to ensure that checkpointed applications can recover to the point of failure.

Another key way in which BRRL differs from previous work is that it uses schema information to select the best checkpointing algorithms for mutable and immutable state. Developers implement their state using a BRRL provided table abstraction, and they annotate the schema with access pattern details to guide BRRL's optimization decisions. For instance, updates to append-only tables can be logged without the need for additional checkpoints. BRRL uses the high-performance Wait-Free Ping-Pong checkpointing algorithm [3], and it reorganizes the physical storage of tables so that access to checkpointing data is cache optimized. We have shown previously that this can lead to dramatic performance improvements [3]. Unlike BRRL, most existing approaches provide a generic memory abstraction and use compiler support to add appropriate checkpointing logic [1, 2]. While these approaches require very little information from the developer, their generality may lead them to choose sub-optimal checkpointing algorithms.

In this demonstration, we will provide an overview of the BRRL library and show how it can be applied to two very different applications. First, we will show how a main-memory OLTP system can be made fault-tolerant with minimal effort without compromising performance. Second, we will apply BRRL to a scalable agent-based simulation platform we developed called BRACE [8] and show how we can take advantage of global points of consistency to avoid the need for logging.

2. OVERVIEW OF BRRL

BRRL is a library for making application state durable. Unlike existing recovery libraries that present a generic memory abstraction [1, 2], BRRL uses semantic information about the application to optimize logging and checkpointing. BRRL is suitable for a wide variety of data-intensive applications, but it does have several requirements. BRRL applications must store their data in tables and access them only through BRRL API functions. They must also have relatively frequent points of consistency at which the application is consistent and can be safely checkpointed. Many applications have these properties, including main-memory transaction processing systems such as H-Store [7] and time-stepped simulations [3].

BRRL uses *logical logging* to persist state during the time between checkpoints in order to roll the application forward to a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'11, June 12–16, 2011, Athens, Greece.
Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...$10.00.

```
class Customer {
immutable:
    int c_id;
    double c_phone[16];
    ...
mutable:
    double c_balance;
    double c_ytd_payment;
    ...
};
```

```
class OrderLine {
immutable:
    int ol_o_id;
    int ol_w_id;
    int ol_quantity;
    double ol_amount;
    char ol_dist_info[24];
    ...
};
```

Figure 1: Example BRRL Tables

Method	Description
init(*recoveryPolicy*)	Initialize BRRL with the appropriate policy.
pointOfConsistency()	Indicates that the application state is consistent
logNewRecord(*record*)	Logs a new record as an uninterpreted string of bytes.
recoverTable(*tableID*)	Recovers a table from a checkpoint
getLogIterator()	Gets an iterator to replay the log.

Table 1: The BRRL API

consistent state in the event of a failure. Though logical logging can increase recovery time compared to physical logging, it introduces less overhead during normal operation, and we can reduce the replay time by taking very frequent checkpoints. To facilitate logging, applications must produce log records for any non-deterministic behavior necessary for re-execution. This includes the content of all external messages and the outcome of any random decisions. For instance, in a DBMS, the logical log would include the type or code and parameters for each transaction as well as their serialization order.

2.1 The BRRL API

BRRL is implemented as a pre-compiler and a C++ library with a simple interface. Application developers specify their state in a set of BRRL tables using a syntax similar to structure definitions in C with scalar types. All of the state in these tables is checkpointed automatically so that it can be recovered in the event of a failure. Figure 1 shows how two TPC-C tables would be specified in BRRL. Each attribute in these tables is classified as either `mutable` or `immutable` based on whether it is ever updated. Immutable attributes can never be updated unless the entire tuple is deleted. In Figure 1, all of the attributes in `OrderLine` are `immutable` since the table is append only, but the `c_balance` and `c_ytd_payment` attributes in the `Customer` table are `mutable` as they are updated whenever a customer makes a purchase or a payment. The BRRL compiler uses these annotations to compile BRRL tables into a set of C++ classes that are optimized for checkpoint-recovery. As BRRL may reorganize the storage layout, applications access their state using get and set methods generated by the compiler. Additionally, rows have no public constructors and must be created and deleted by calling methods on the appropriate table.

Table 1 shows additional BRRL methods to identify points of consistency, write log records, and recover state in the event of a crash. When the library is initialized (`init`), the application must specify high-level information about the application, including the checkpoint interval, the number of log servers to use, whether or not to use group commit. The application then calls `pointOfConsistency` at every point of consistency. Note that BRRL does not necessarily take a checkpoint each time this method is called, but every checkpoint will start at a point of consistency. The application logs non-deterministic events using `logNewRecord`. The format of the log records is determined by the application – BRRL never executes the log. The recovery portion of the API consists of a method to return the most recent checkpoint of a persistent table (`recoverTable`) and an iterator to access the

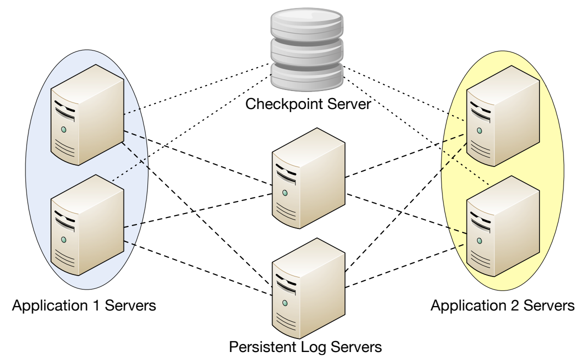

Figure 2: The BRRL Architecture

logical log (`getLogIterator`). It is the application's responsibility to use these methods to recover to a consistent state.

2.2 Checkpointing and Recovery in BRRL

BRRL includes components for efficient logging and checkpointing. The high-level architecture is shown in Figure 2. Logging is implemented using a set of shared *persistent log servers*. When the application calls `logNewRecord`, the BRRL library sends the record to one or more log servers (the number is configurable). When one of these servers receives a log record, it stores the record in main memory and flushes it asynchronously to disk. This allows for very low-latency logging since the log server returns as soon as the log record has been written to memory, and we expect the network latency to be much lower than the cost of a disk seek. This approach is similar to the *buffered logging* technique proposed by the RamCloud project [4]. For applications that cannot tolerate the loss of any of the log tail in the event of a catastrophic failure or power outage, the application servers can use group commit and wait until the log servers have flushed each record to disk.

Notice that since the persistent log servers do not have to perform any computation and can write to disk asynchronously, they can be shared by many applications. This is shown in Figure 2, where two applications share the same two log servers and the same shared disk. This sharing is particularly beneficial for cloud computing environments where many applications are multiplexed on the same infrastructure. This sharing can yield better resource utilization than approaches such as *k*-safety that rely on many additional replicas [7].

By default BRRL checkpoints data to a shared disk, though any stable-storage media would be suitable. We use the Wait-Free Ping-Pong checkpointing algorithm which we developed for frequently consistent main-memory applications [3]. This algorithm maintains two copies of the application state in memory and collects updates in one of them for each checkpoint period (thereby "ping-ponging" between them). BRRL automatically interleaves attributes from the original state and the two copies in order to minimize the number of cachelines that must be accessed during an update. This was shown to improve performance dramatically [3]. BRRL only applies these optimizations to mutable state in order to reduce the memory and update overhead. Immutable state is only written to the checkpoint once when it is first created. Deletes are persisted by writing a delete record with the checkpoint that can be replayed during recovery.

3. DEMONSTRATION

We will demonstrate how to use BRRL to make two different applications durable. In the first part of the demonstration, we will demonstrate a main-memory transaction processing system with

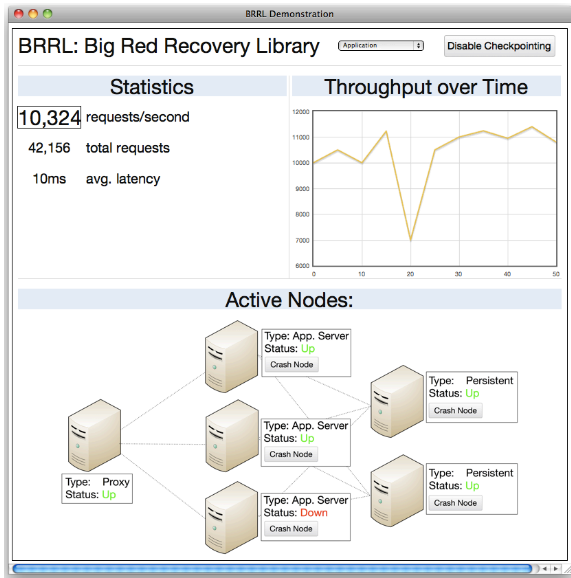

Figure 3: Interface for the TPC-C Demonstration

the TPC-C benchmark, and in the second part we will demonstrate the recovery procedure using our behavioral simulation platform, BRACE [8]. All of our demonstrations will be run in the cloud using Amazon's web services (AWS) platform, though we will have a local backup in case of network problems.

3.1 Making TPC-C Durable

We will demonstrate a distributed main-memory implementation of TPC-C for the cloud that uses BRRL to provide durability. Our implementation horizontally partitions the data by warehouse and distributes the partitions across several processors. We can replicate these partitions to provide high availability, but checkpointing is still necessary to provide durability in the event of correlated failures such as power outages.

All transactions are written as stored procedures in C++, and each processor executes transactions sequentially in a single thread. This method was shown to be effective in main-memory transaction processing systems [7]. Client requests include a transaction identifier and list of parameters and are passed through a proxy that serializes them and distributes them to the appropriate partitions. The proxy, application servers, and persistence servers are implemented on Amazon EC2 instances, and the checkpoint server is implemented using Amazon's Elastic Block Storage (EBS) service.

Using our transaction processing system, we will show that it is easy to use the BRRL API, and that the overhead for providing durability is modest, even for distributed applications. To demonstrate the first point, we will show the code of the stored procedures that process each of the TPC-C transactions. We will show that it is natural to use the BRRL persistent table to store and access tuples and that it is straightforward to implement additional indices on top of the BRRL API.

Additionally, we will demonstrate the performance of BRRL using the monitoring application shown in Figure 3. This tool reports the transaction throughput and latency measured at the proxy, and includes a plot of throughput over time. By turning checkpointing on and off, we will show that throughput remains high even when checkpointing is enabled. We will also demonstrate what happens to the performance in the event of a crash. Our tool includes buttons to fail individual nodes, and we will show that throughput quickly returns to its pre-crash rate in the event of a failure.

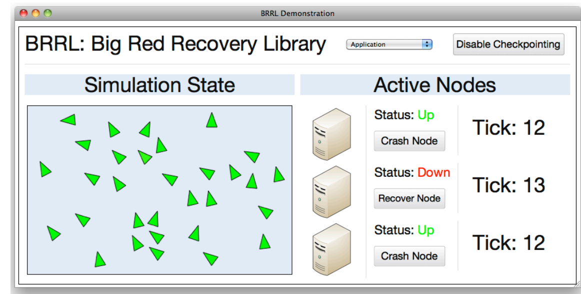

Figure 4: Interface for the BRACE Demonstration

3.2 Recovering a BRACE Simulation

In order to show that BRRL provides very short recovery times for a wide range of applications, we will also show a version of the BRACE simulation platform that uses BRRL to provide durability. BRACE is a simulation framework we developed for behavioral simulations [8]. Behavioral simulations are typically spatial, and BRACE assigns the task of simulating disjoint regions of space to different nodes. Each node must synchronize with its spatial "neighbors" at the end of each time-step, which creates a global point of consistency. As BRACE simulations are otherwise deterministic, we can use a global checkpointing policy and turn off logical logging.

We will demonstrate BRACE using a simulation of schooling fish, as shown in the interface in Figure 4. This application includes information about the status of each machine in the cluster and the tick each machine is executing. When a machine crashes, we will show that the entire system can be quickly restarted from the most recent global checkpoint and re-executed to the point of failure. The viewer will be able to verify that the tick numbers and fish positions return to their values as of the point of failure.

Acknowledgments. This material is based upon work supported by the New York State Foundation for Science, Technology, and Innovation under Agreement C050061, by the National Science Foundation under Grants 0725260 and 0534404, by the iAd Project funded by the Research Council of Norway, by the AFOSR under Award FA9550-10-1-0202, and by Microsoft.

4. REFERENCES

[1] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated Application-level Checkpointing of MPI Programs. In *Proc. PPoPP*, 2003.

[2] G. Bronevetsky, M. Schulz, P. Szwed, D. Marques, and K. Pingali. Application-level Checkpointing for Shared Memory Programs. In *Proc. ASPLOS*, 2004.

[3] T. Cao, M. V. Salles, B. Sowell, Y. Yue, A. Demers, J. Gehrke, and W. White. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proc. SIGMOD*, 2011.

[4] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramclouds: scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, 2010.

[5] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under UNIX. In *Proc. USENIX Winter Technical Conference*, 1995.

[6] B. Ramkumar and V. Strumpen. Portable checkpointing for heterogeneous architectures. In *In Symposium on Fault-Tolerant Computing*, pages 58–67. Kluwer Academic Press, 1997.

[7] M. Stonebraker, S. Madden, D. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proc. VLDB*, 2007.

[8] G. Wang, M. V. Salles, B. Sowell, X. Wang, T. Cao, A. Demers, J. Gehrke, and W. White. Behavioral simulations in mapreduce. In *Proc. VLDB*, 2010.