SinfoniaEx : Fault-Tolerant Distributed Transactional Memory

Mahdi Tayarani Najaran
Computer Science Department
University of British Columbia
tayarani@cs.ubc.ca

Charles Krasic
Computer Science Department
University of British Columbia
krasic@cs.ubc.ca

Abstract

We present SinfoniaEx, a powerful paradigm for designing distributed applications. SinfoniaEx is an extension to Sinfonia, a service that provides fault-tolerant atomic access to distributed memory, and is suitable for cloud environments. SinfoniaEx is built over the same design principles of Sinfonia, while extending the interface to allow applications to share system resources, i.e. memory nodes.

1 Introduction

Sinfonia is a distributed memory service based on short-lived minitransactions [2]. In Sinfonia, a set of memory nodes export an unstructured address space, where applications access the memory using light-weight minitransactions that may have read/write/compare items. SinfoniaEx extends Sinfonia by providing a set of new transaction items which simplify applications built over it, especially distributed data structures.

We extend the basic service provided by Sinfonia via providing a set of new minitransaction items. SinfoniaEx provides applications with memory management items along with dictionary items. By using memory management items, application rely on memory nodes to handle memory mappings, which allows memory nodes to be shared between different types of applications using SinfoniaEx. Dictionary items are designed to maintain backward compatibility with Sinfonia.

2 Sinfonia

Sinfonia is a service allowing applications to share data in a fault-tolerant, scalable and consistent manner. It has been shown to provide an efficient programming model for classic distributed systems problems, such as file systems, group communications, and scalable storage [2, 1].

A set of Sinfonia *memory nodes* export an unstructured linear address space, where a data pointer consists of the tuple $\{$*memnode-id,offset*$\}$. Applications access the exported memory atomically via short-lived light-weight *minitransactions* [1]. Each minitransaction uses a *two-phase commit*, which is designed to provide high performance by using at most two network round-trips to complete (regardless of the commit succeeding or not).

A transaction consists of three groups of items, categorized by the kind of operations they perform: *read items*, *compare items* and *write items*. Read items read and return data from the specified addresses, compare items compare user provided data with data stored in memory, and write items write provided user data to specified memory locations. A transaction begins with empty items and the application adds items to each category before calling commit (note that each item can be pointing to data on different memory nodes). Once a transaction is committed, it either fails if and only if at least one compare item failed (i.e. the data in the compare buffer does not match data stored in the referenced memory location), or succeeds, in which case data in the write buffers are written to memory and read items contain buffers holding data read from memory.

The application trying to commit a transaction is the *coordinator* of the transaction, and memory nodes referenced by the items of the transaction are *participants*. Committing consists of two phases. In the first phase, the coordinator notifies participants of the items in the transaction that affect them. Each participant processes the items and sends its vote to the coordinator which makes the final decision. If the participant manages to lock memory locations referenced by the transaction and the compare items are exactly the same as data stored in memory, the participant votes to succeed. If compare fails, the vote is to fail and locks are released, and if locking failed in the first place, the vote is to abort. Once the

[1]We use the terms transaction and minitransaction interchangeably. Not to be mistaken with traditional database transactions.

coordinator receives votes from all participants, it decides the fate of the transaction, which is either a success, failure, or retry. If all participants voted to succeed, commit has succeeded and the application is notified. If at least one participant voted to fail, commit has failed and the application is notified. If at least one participant voted to abort, the transaction is aborted and retried after waiting a random amount of time. In all cases, the final decision made by the coordinator is sent to the participants so they can cleanup state.

Minitransactions provide consistency by locking memory locations they reference while committing is in progress. Unlike conventional full feature transactions, minitransactions are meant to be light-weight. The two-phase commit ensures locks on memory locations are short-lived, lasting at most two network round-trips (one and a half to be accurate), providing atomic transactions with high performance as the basic memory access primitive. Sinfonia also provides fault tolerance and some optimizations techniques discussed in [2] to further increase reliability and performance of Sinfonia.

3 SinfoniaEx

This section describes our extensions to Sinfonia, creating a more general service we call *SinfoniaEx*, available with an open source implementation (Section 5).

Sinfonia memory nodes export a raw unstructured address space, providing access to any arbitrary part of memory to any arbitrary transaction. This forces applications to keep track of memory maps themselves, i.e. what parts of memory of each memory node are holding application data and what parts are un-allocated. This increases the complexity of the applications using Sinfonia, forcing them to implement their own memory allocation/deallocation functions, namely $malloc$ and $free$, that have long existed in all operating systems and are highly optimized.

Additionally, since each application handles its own memory mapping, a set of memory nodes used by one application have to be exclusive to that application, and can not be shared with other applications to avoid data corruption. For example consider the distributed B-tree in [1]. Memory nodes hosting B-tree A cannot be used to host B-tree B, even though they are two instances of the same application. Moreover, memory nodes hosting B-tree A cannot be used to host distributed file system B, even though they're both built over Sinfonia.

SinfoniaEx is an extension to Sinfonia. As with Sinfonia, SinfoniaEx relies on two-phase short-live minitransactions. However, apart from read, compare and write items, SinfoniaEx transactions may include memory management and dictionary items. The combination of these items obviate the need for applications or external services to perform these operations, while maintaining the same effectiveness of minitransactions. Figure 1 shows SinfoniaEx items. Notice the first three are the same as Sinfonia, while the rest are specific to SinfoniaEx. In the remainder of this section, we explain each set of new items and changes made to the commit protocol.

```
//Memory operation items
read(memnode-id, offset, len)
compare(memnode-id, offset, len, data)
write(memnode-id, offset, len, data)

//Memory management items
malloc(memnode-id, pointer-id, len, data)
free(memnode-id, offset)

//Dictionary operation items
lookup(memnode-id, key)
compare_key(memnode-id, key, value, len)
put(memnode-id, key, value, len)
remove(memnode-id, key)
compare_non_exist(memnode-id, key)

//Access macros
buf = get_read_item(memnode-id, offset)
buf,len = get_lookup_item(memnode-id, key)
offset = get_malloc_offset(memnode-id, pointer-id)
```

Figure 1: SinfoniaEx items: read items are locations to read and return data, compare items are locations to compare to given data, write items are to write given data to, malloc items are to allocate memory of requested size, initialized to data (if provided) or 0 otherwise, free items are to be freed, lookup items are keys to lookup and return value, compare_key items are keys to be compared to the given value, put items replace key value with given value, remove items are to be removed and compare_non_exist ensure given item does not exist prior to committing. Once commit succeeds, access macros can be used to retrieve read data, looked up data and offset of newly allocated memory locations.

3.1 Memory Management Items

A transaction can allocate/de-allocate memory using *malloc* and *free* items, respectively. A malloc item takes the *memnode-id* of the memory node that should host the newly allocated location, along with a user provided handle *pointer-id*. Once the transaction commits successfully, the offset of the newly allocated pointer can be retrieved using the *get_malloc_offset* macro by providing the same handle. An optional data pointer may also be provided to malloc in order to initialize the newly allocated memory with user specified values. Free items simply take *memnode-id* and *offset* of the memory location to release upon successful commit.

```
1   $tx$ ← new SinfoniaEx_Tx()
2   tx.malloc( 0, 238, 100, nil )
3   rand_memnode = rand() mod #memnodes
4   tx.malloc( rand_memnode, 3, 200, “some data” )
5   tx.compare_non_exist( 3, 123456 )
    tx.put( 3, 123456, ”a string”, 10 )
6   tx.commit()
7   **if** tx.success() **then**
8       $offset1$ ← tx.get_malloc_offset( 0, 238 )
9       $offset2$ ← tx.get_malloc_offset( rand_memnode, 3 )
10      tx.end()
11      $tx$ ← new SinfoniaEx_Tx()
12      tx.free( 0, offset1 )
13      tx.free( rand_memnode, offset2 )
14      tx.commit()
15  **else**
16      tx.end()
17      $tx$ ← new SinfoniaEx_Tx()
18      tx.lookup( 3, 123456 )
19      tx.remove( 3, 123456 )
20      tx.commit()
21      $buf, len$ ← tx.get_lookup_item( 3, 123456 )
22  tx.end()
```

Figure 2: SinfoniaEx Example.

3.2 Dictionary Items

In a distributed application, different application nodes may wish to communicate by storing and retrieving meta data stored at specific memory locations. For example, an application node can store meta data about the root of a distributed tree at a specific location in memory. Other application nodes can then look for the root in the same specific location. However, in SinfoniaEx application nodes have no control over where to store the shared meta data, and are forced to store it in the location specified by the memory node. Hence, SinfoniaEx provides a set of three dictionary items.

The *lookup*, *compare_key* and *put* items are identical to Sinfonia's read, compare and write items, and simply take a pointer and buffer holding data (if required). Based on their semantics, we decided to rename them to those used for key/value operations, where the key is the memory pointer and value is the buffer. Sinfonia provides no way for an application to distinguish whether a memory location is in use and holds valid data or not. So to complete the set, we added *remove* and *compare_non_exist* items. Put stores provided data to given address, and marks the address as occupied. Lookup, compare_key and remove only succeed on occupied locations, where remove marks the address as unoccupied. Finally, a compare_non_exist item ensures the provided memory location is un-occupied prior to committing the transaction.

Figure 2 illustrates an example using most of SinfoniaEx's items. The first transaction allocates two memory pointers, one on memory node 0 (line 2) and one on a random node (line 3-4), and tries to put data to a memory node 3 with offset 123456 only if it does not already exist (line 5-6). If the transaction succeeds, offsets of the allocated pointers are stored (line 9-10), which are then freed with another transaction (line 12-15). The first transaction can only fail if memory location 123456 on node 3 is occupied, in which case a new transaction is issued to read and remove this data (line 18-22).

3.3 Minitransactions

Figure 3 lists the commit protocol for both the coordinator and participant of a minitransaction in SinfoniaEx. Most of the protocol is the same as commit in Sinfonia. We have marked the lines we had to change for SinfoniaEx with a (*). Handling node failures in SinfoniaEx is the same as Sinfonia and is beyond the scope of this paper. However, for completeness reasons, we still present them in Figure 3 (lines 13,21-22,29-30). For details on failure handling refer to [2].

The coordinator's task remains unchanged (lines 1-9), consisting of two phases. In the first phase, the coordinator sends the transaction items affecting each participant to the corresponding participants with an EXEC&PREPARE message accompanied by a transaction id (line 4). The coordinator then waits for replies from all participants, after which it makes the final decision. If all participant' votes were OK, commit succeeds, else it has either failed or has been aborted (in which case committing should be retried). The final decision is then sent to the participants and commit ends.

On the receiving side, when a participant receives EXEC&PREPARE (lines 10-25), it tries to issue a non-blocking lock on all valid memory locations referenced by items of the transaction. Items referencing invalid memory locations are dealt with in the next step (line 14). If locking fails, the participant's vote becomes BAD-LOCK. Once all locks are acquired, all items are checked to find invalid memory references, which may be caused by bugs or locations released by other transactions, and data provided by compare and compare_key items are compared to what is stored in memory. If any items fail these tests, they are stored in $data$ and reported to the coordinator with a BAD-CMP vote (line 16). If all goes well, the participant votes to OK, allocates memory for the malloc items, and returns the location of the newly allocated memory with the read and lookup items with its vote (lines 19-20,25).

In the second phase, upon receiving COMMIT (lines 26-34), if the coordinator decided to commit, data provided by write_items and put_items are written to memory, and free_items are freed. However, if the decision was to abort, memory locations allocated by the malloc_items should be released. Regardless of the vote, all locks still held by the transaction are released at this

```
For coordinator $p$:
commit($tx$)
1    $tx.tid$ ← new unique minitransaction id
     //Phase 1
2    $D$ ← set of memory nodes referred by items of $tx$, i.e. $tx.items$
3    foreach $q \in D$ do
         //sends $msg$, $tid$, $tx.items$ handled by $q$ and $D$ to $q$
4        send ($q$, EXEC&PREPARE, $tx.tid$, $\pi_q(tx.items)$, $D$)
5    replies ← wait for replies from all nodes in $D$
     //Phase 2
6    if $\forall q \in D : replies[q].vote$==OK then $action$ ← COMMIT
7    else $action$ ← ABORT
8    foreach $q \in D$ do send ($q$, COMMIT, $tid$, $action$)
9    return $action$ //does not wait for reply of COMMIT

Code for each participant memory node $q$:
upon receive (EXEC&PREPARE, $tid$, $items$, $D$) from $p$ do
10   $in\_doubt \leftarrow in\_doubt \cup \{(tid, items)\}$
11   $data \leftarrow \emptyset$
12   if try-lock($items$)==FAIL then $vote$ ← BAD-LOCK
     //forced-abort is used with recovery
13   else if $tid \in forced\_abort$ then $vote$ ← BAD-FORCED
14*  else if any items referring invalid locations or
         $compare$ and $compare\_key$ items don't match data then
15       $vote$ ← BAD-CMP
16       $data$ ← {failed items}
17   else $vote$ ← OK
18   if $vote$==OK then
19*      $pointers$ ← malloc($items.malloc$)
20*      $data \leftarrow \{items.read, items.lookup, pointers\}$
21*      add ($tid$, $D$, $items.\{write, put, free\}$) to $redo\_log$
22       add $tid$ to $all\_log\_tids$
23   else
24       release locks acquired above
25   send-reply ($tid$, $vote$, $data$) to $p$
upon receive (COMMIT, $tid$, $action$) from p do
26   $items \leftarrow in\_doubt.find(tid)$
27   if not $found$ then return //recovery coordinator executed first
28   $in\_doubt \leftarrow in\_doubt - \{(tid, items)\}$
29   if $tid \in all\_log\_tids$ then
30       $decided \leftarrow decided \cup \{(tid, action)\}$
31   if $action$==COMMIT then
32*      apply($item.write$, $items.put$, $items.free$)
33*  else free($items.malloc$) //release memory since $tx$ failed
34   release any locks still held for items
```

Figure 3: Commit protocol for SinfoniaEx transactions.

point.

3.4 Load Balancing

Balancing the load of different memory nodes is an important feature of SinfoniaEx, which is desired by many applications. Each SinfoniaEx memory node uses a set of reserved keys to store meta data about its current load information, such as allocated and free memory byte counts, number of requests handled per time unit, etc, accessible through transactions. Considering applications decide which memory nodes to use when allocating memory locations, they can benefit from this information to select the most suitable memory nodes.

Moreover, applications can use transactions to migrate data to and from memory nodes. By periodically reading this meta data, applications can shift around data to evenly spread their load between different memory nodes.

3.5 Security

SinfoniaEx allows different applications to share the same memory nodes. Albeit, sharing memory nodes may pose security threats, since an application can virtually access any other application's private data. SinfoniaEx is based on the same assumptions as Sinfonia, in which applications operate in a data center and their designers are trustworthy, rather than malicious (in contrast to WANs and peer-to-peer systems) [2]. Nonetheless, security credentials can be added to transactions and memory nodes can authenticate applications while restricting their access to memory locations, a task left for future work.

4 Conclusion & Future Work

SinfoniaEx extends Sinfonia by providing a set of new transaction items, while preserving the same principles such as short-lived minitransactions and fault tolerance.

Using SinfoniaEx allows applications to share the same memory nodes. We take advantage of this feature to host multiple distributed data structures and application data on the same memory nodes.

5 Availability

SinfoniaEx is free software part of the project QStream available at:

`http://www.qstream.org`

References

[1] AGUILERA, M. K., GOLAB, W., AND SHAH, M. A. A practical scalable distributed b-tree. *Proc. VLDB Endow. 1* (Aug. 2008), 598–609.

[2] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP '07: Proc. of twenty-first ACM SIGOPS symposium on Operating systems principles* (New York, NY, USA, 2007), ACM, pp. 159–174.