1 SUPPLEMENTARY MATERIAL

1.1 Example Applications

An example implementation of the banking database is shown in Algorithm 1, and the voting app is shown in Algorithm 2 with the abstractions provided by Reliable CRDTs.

Algorithm 1 Banking database using Reliable CRDTs

```
function check_invariant(balance)
    return true if balance ≥ 0 else false                  ▷ Balance must be nonnegative

function withdraw(amount, account : counter)                ▷ Account is a CRDT counter
    balance ← account.query_stable()
    if balance > amount then
        return account.decrement_safe(amount)  ▷ Safe decrement, only returning true if the operation is successful

function deposit(amount, account: counter)
    account.increment(amount)  ▷ Since depositing never breaks the invariant, we can use the regular increment and
    assume it will eventually be included in the stable state

function display_balance(account : counter)
    return account.query_prospective()                      ▷ Displaying the balance is not critical

function calculate_interest(account : counter)
    amount ← account.query_stable()          ▷ Interest is calculated based on the settled cash in the account
    account.increment(amount × interest_rate)

function transfer(account1, account2: counter)
    u1 ← account1.decrement_safe(amount)
    if u1.success then
        u2 ← account2.increment(amount)
    return u1.success ∧ u2.success
```

Algorithm 2 Voting system using Reliable CRDTs

```
function cast_vote(candidate : counter)
    candidate.increment(amount)
function display_vote(candidate : counter)
    return candidate.query_prospective()  ▷ Displaying the vote count before the voting closes is allowed to be incon-
    sistent
function decide_winner(all_candidates : listofcounter)
    votes : List
    for all candidate in all_candidates do
        votes.add(candidate.query_stable())  ▷ Need to make sure all replicas have received the same vote counts, and
        any conflicting votes are excluded
    return all_candidates[votes.index(max(votes))]          ▷ Winner is decided based on the stable state
```

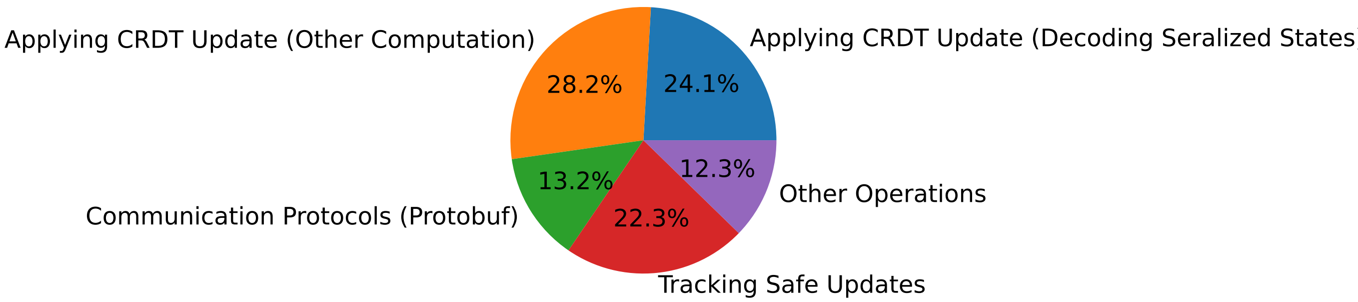

Fig. 1. CPU time breakdown for each component during an experiment run with PN-Counter, balanced access patterns, 50% safe updates, $b = 500$ 100 objects and 4 nodes.

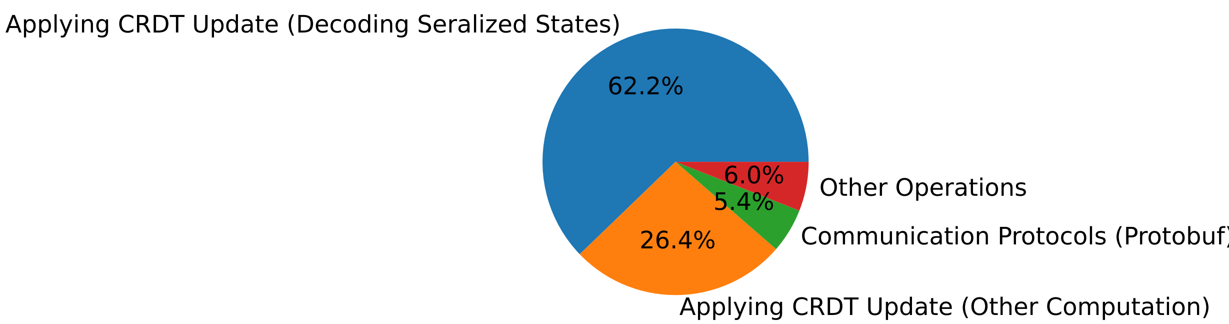

Fig. 2. CPU time breakdown for each component during an experiment run with OR-Set, balanced access patterns, 50% safe updates, $b = 500$ 100 objects and 4 nodes.

1.2 Execution Time Analysis

We profile the system for 60 seconds during an experiment run when the system was saturated with *dotnet-trace* tool. The breakdown for the cost of each operation is shown in Figure 1 for PN-Counter.

We note that the majority (52.3%) of total CPU time was used for applying CRDT updates on the stable and prospective state (see Figure 3a and Figure 3b, *MergeSharp* is the CRDT library that we used). Furthermore, the implementation of the CRDT is not optimal where its states are serialized into JSON strings before they are transmitted for updates, resulting in a 24.1% of the total CPU time spent on serializing and deserializing the JSON.

It is worth noting that tracking safe updates took a significant 22.3% of the total CPU time, which can be seen in Figure 3a. We use a *ConcurrentDictionary* class to map between the updates and the client that requested them, indicating that the dictionary is not optimized for highly contested operations.

Finally, The communication among nodes is conducted with Protobuf, which consumed a significant 13.2% of the total CPU time, as shown in Figure 3c. The DAG operations, such as handling blocks and traversing the graph did not consume a significant amount of CPU time, as they were only a part of the remaining 12.3% of the total CPU time.

For OR-Set, the CPU time breakdown is shown in Figure 3. Applying CRDT updates took over 80% of the total CPU time, showing the bottleneck of the current implementation is in the CRDT library.

Function Name	Total CPU [unit, %]	Self CPU [unit, %]	Module	Category
Process(58945) (PID: 58945)	58185 (100.00%)	0 (0.00%)	Multiple modules	
System.Threading.PortableThreadPool+WorkerThread.WorkerThreadStart()	24669 (42.40%)	0 (0.00%)	System.Private.Co...	Json | Protobuf
System.Threading.ThreadPoolWorkQueue.Dispatch()	24417 (41.96%)	0 (0.00%)	System.Private.Co...	Json | Protobuf
System.Threading.Tasks.Task.ExecuteWithThreadLocal(ref System.Threading.Tasks.Task, S...	21609 (37.14%)	0 (0.00%)	System.Private.Co...	Json
System.Threading.ExecutionContext.RunFromThreadPoolDispatchLoop(System.Thread...	21609 (37.14%)	0 (0.00%)	System.Private.Co...	Json
BFTCRDT.SafeCRDTManager.HandleAfterConsensusUpdates.AnonymousMethod__0()	21606 (37.13%)	186 (0.32%)	BFT-CRDT	Json
System.Collections.Concurrent.ConcurrentDictionary<T, System.ValueTuple<T, ui...	12944 (22.25%)	0 (0.00%)	System.Collection...	
BFTCRDT.SafeCRDT.ApplyUpdateStable(MergeSharp.NetworkProtocol)	8097 (13.92%)	102 (0.18%)	BFT-CRDT	Json

(a) Applying Updates on the Stable State and Tracking Safe Updates

Function Name	Total CPU [unit, %]	Self CPU [unit, %]	Module	Category
Process(58945) (PID: 58945)	58185 (100.00%)	0 (0.00%)	Multiple modules	
System.Threading.PortableThreadPool+WorkerThread.WorkerThreadStart()	24669 (42.40%)	0 (0.00%)	System.Private.Co...	Json | Protobuf
BFTCRDT.DAG.ManagerServer.HandleReceived.AnonymousMethod__0()	24582 (42.25%)	0 (0.00%)	BFT-CRDT	Json | Logging | Pr...
BFTCRDT.DAG.DAG.HandleMessage(BFTCRDT.DAG.DAGMessage)	22601 (38.84%)	0 (0.00%)	BFT-CRDT	Json | Logging
BFTCRDT.DAG.DAG.ReceivedBlock(BFTCRDT.DAG.VertexBlock, bool)	22590 (38.82%)	2 (0.00%)	BFT-CRDT	Json
BFTCRDT.DAG.ConnectionManager.ReceivedBlock(BFTCRDT.DAG.VertexBlock)	22579 (38.81%)	106 (0.18%)	BFT-CRDT	Json
BFTCRDT.SafeCRDTManager.HandleReceivedUpdateFromBlockSync(System.EventHa...	22473 (38.62%)	50 (0.09%)	BFT-CRDT	Json
MergeSharp.ReplicationManager.HandleReceivedSyncUpdate(object, MergeShar...	22306 (38.34%)	40 (0.07%)	MergeSharp	Json

(b) Applying Updates on the Prospective State

Name	Total CPU [unit, %]
Process(58945) (PID: 58945)	58185 (100.00%)
BFT-CRDT	57850 (99.42%)
MergeSharp	34983 (60.12%)
System.Private.CoreLib	30359 (52.18%)
System.Collections.Concurrent	14230 (24.46%)
System.Text.Json	14024 (24.10%)
protobuf-net	7670 (13.18%)
protobuf-net.Core	7658 (13.16%)
System.Net.Sockets	3175 (5.46%)
System.Threading.Channels	2948 (5.07%)
System.Linq	128 (0.22%)
System.Security.Cryptography	77 (0.13%)
CRDTClassProxyAssembly46a06a38-6823-4a71...	52 (0.09%)
Microsoft.Extensions.Logging	1 (0.00%)
Microsoft.Extensions.Logging.Abstractions	1 (0.00%)

(c) Overall CPU Time Breakdown

Fig. 3. Profiler Screenshots for the Execution Time Analysis of experiment in Figure 1