Exam
Distributed Systems

5 February 2010, 9:00am – 12:00pm

Part 2 – Prof. R. Wattenhofer

Family Name, First Name: . . . . . .

ETH Student ID Number: . . . . . .

Task	Maximum	Achieved	Initials
8	28		
9	21		
10	12		
11	29		
Subtotal	90		

Part 2: . . . . . .

Question 8: Multiple Choice *(28 Points)*

Each statement is either *true* or *false*. Indicate your answer by checking the corresponding box. A correct answer gives **1 point**, a wrong answer gives **-1 point**, no answer gives **0 points**. Overall you cannot get less than 0 points for this question.

8 a) (7 points) General questions

true	false	
☐	☐	Leader election can be solved with a consensus algorithm
☐	☐	Every thinkable RMW-register can be simulated using CAS.
☐	☐	If an algorithm is f-resilient, it must consist of at least $2f + 1$ synchronous rounds.
☐	☐	CAS is the only RMW-register with consensus number ∞.
☐	☐	If two RMW-registers have the same consensus number, then the behavior of one register can be simulated using the other one.
☐	☐	The size of a quorum is always bigger or equal to the size of a majority set.
☐	☐	A quorum is a set containing $2f + 1$ processes/servers.

8 b) (4 Points) Given a lock free consensus algorithm and two processes with different input. The root of the execution tree...

true	false	
☐	☐	... might be univalent.
☐	☐	... must be bivalent.
☐	☐	... can be critical.
☐	☐	... must be critical.

8 c) (3 Points) If RMW-register x can be constructed from y and x has consensus number c...

true	false	
☐	☐	... then y has consensus number c.
☐	☐	... then y has consensus number at least c.
☐	☐	... then y is non-trivial.

8 d) (4 Points) The Anderson Queue Lock (ALock)...

true	false	
☐	☐	... is a linked list.
☐	☐	... uses `fetch-and-inc`.
☐	☐	... does not scale well.
☐	☐	... is not ideal for NUMA.

8 e) (4 Points) Zyzzyva...

true	false	
☐	☐	... optimistically executes an operation and checks replies for inconsistencies.
☐	☐	... is able to handle up to $f < n/3$ Byzantine failures.
☐	☐	... is completely decentralized and does not require a lead server.
☐	☐	... outperforms PBFT (according to the numbers presented in the lectures)

8 f) (4 Points) Theory and practice: given a database that is replicated among n servers...

true	false	
☐	☐	It is not possible to use any algorithm based on synchronous rounds in a system with asynchronous communication.
☐	☐	Assuming only crash failures, 2PC is a good choice to handle coordinated progress.
☐	☐	The system may stall forever if Paxos is used for synchronization.
☐	☐	PBFT guarantees that all correct servers have the same content in their databases.

Question 9: RMW-registers *(21 Points)*

9 a) (3 points) Old hardware often supports only RMW register like `fetch-and-inc` or `fetch-and-swap`. Newer hardware supports rather `compare-and-swap`. Why?

A new hardware contains a new type of RMW-registers, so called "double RMW registers" (dRMW). If some RMW register executes a function $f(x)$ atomically, then its dRMW counterpart executes two functions $[f_1(x), f_2(y)]$ atomically on two registers. The pseudo-code below describes a generic dRMW.

```
1  class GenericDoubleRMW{
2    /**
3     * Atomically executes "f1" and "f2" for the registers "r1" and "r2".
4     * This method works with any two registers. The method returns the
5     * values that were stored in "r1" and "r2" before "f1" and "f2"
6     * were executed.
7     */
8    ValuePair execute( Register r1, Register r2, Function f1, Function f2 ){
9      atomic{
10       int previous1 = r1.get();
11       int previous2 = r2.get();
12       r1.set( f1.eval( previous1 ) );
13       r2.set( f2.eval( previous2 ) );
14       return new ValuePair( previous1, previous2 );
15     }
16   }
17 }
```

Find the consensus-number for the following dRMW-registers.

9 b) (6 points) The `double-fetch-and-add` register: A process can use any function of the form $f_a(x) = x + a$ to call the dRMW. Prove your answer.

Example: calling `GenericDoubleRMW.execute(` x, y, f_3, f_8 `)` would atomically increase the value of register x by 3 and of y by 8.

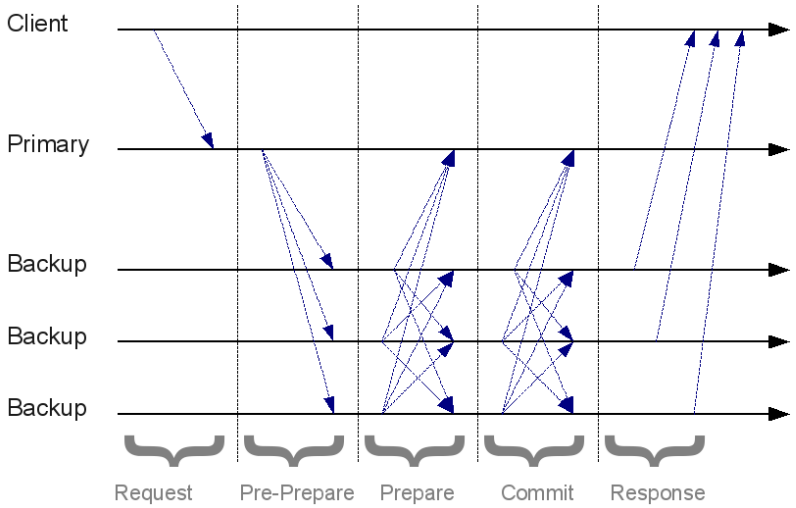
9 c) (12 points) The `double-fetch-and-set` register: A process can use any function of the form $f_s(x) = s$ to call the dRMW. Sketch a proof.

Example: calling `GenericDoubleRMW.execute(` x, y, f_4, f_7 `)` would atomically set the value of register x to 4 and of y to 7.

Hint: show that a consensus number c is between a lower and an upper bound $b_{lower} \leq c \leq b_{upper}$. You get full points if you show that $b_{lower} = b_{upper}$, partial results give points as well.

Question 10: PBFT *(12 Points)*

Recall how PBFT (Practical Byzantine Fault-Tolerance) works:



10 a) (4 points) What are the view- and sequence-numbers good for?

10 b) (5 points) Why is there a prepare phase?

10 c) (3 points) Give a concrete example where PBFT would fail without the prepare phase.

Question 11: Synchronization *(29 Points)*

11 a) (10 points) List and explain (one sentence each) five synchronization patterns. Describe a different use case for each pattern.

Study the source code that is attached to this question. It is a linked list offering a method to `insert` and to `remove` items. The list is ordered and does not allow an item to be twice in the list.

11 b) (5 points) What kind of synchronization pattern is used in this list?

11 c) (8 points) The current implementation has flaws and there are (at least) two scenarios where the list gets corrupted if modified concurrently by two threads. Describe two of these scenarios, what are the effects?

11 d) (6 points) How could the scenarios you found in b) be repaired without changing the synchronization pattern? Write down ideas (50% of the points) and implement them (50% of the points). You may use pseudo-code or real code.

```
1  class LinkedList{
2    // the head of the list, it can never be removed
3    Node head = new Node();
4
5    class Node{
6      int value;
7      Node next;
8      /* Only one thread can 'acquire' the lock at a a time.
9       * Threads need to 'release' the lock to free it. */
10     Lock lock = new Lock();
11   }
12
13   // insert 'x', but only if not already in this list
14   void insert( int x ){
15     Node node = findPredecessor( x );
16     if( node.next == null || node.next.value != x ){
17       Node insert = new Node();
18       insert.value = x;
19       insert.next = node.next;
20       node.next = insert;
21     }
22     node.lock.release();
23   }
24
25   // remove 'x' from this list
26   void remove( int x ){
27     Node node = findPredecessor( x );
28     Node next = node.next;
29     if( (next != null) && (next.value == x) ){
30       node.next = next.next;
31     }
32     node.lock.release();
33   }
34
35   Node findPredecessor( int x ){
36     Node previous = head;
37     Node current = head.next;
38     while( true ){
39       if( (current == null) || (current.value >= value ) ){
40         previous.lock.acquire();
41         return previous;
42       }
43       else{
44         previous = current;
45         current = previous.next;
46       }
47     }
48   }
49 }
```