Homework Assignment 5
Sample Solution

1. In the examples below, the operations are performed by the same process P on two different local copies (L1 and L2) of the same data store.
*(a) Does the following example satisfy **monotonic reads consistency**? Explain why or why not.*
*(b) Does the following example satisfy **monotonic writes consistency**? Explain why or why not.*

L1	$W(x_1)$				$WS(x_1;x_3)$
L2		$WS(x_1;x_2)$	$R(x_2)$	$R(x_1)$	

*(c) Does the following example satisfy **read-your-writes** consistency? Explain why or why not.*
*(d) Does the following example satisfy **writes-follow-reads** consistency? Explain why or why not.*

L1		$WS(x_1;x_2)$	$R(x_2)$	
L2	$W(x_1)$			$WS(x_2;x_3)$

(a): It **does not** satisfy **monotonic reads consistency** since process *P* first reads in L2 $R(x_2)$, i.e., the most recent version of *x* at that time, and then it reads $R(x_1)$, i.e., an older version of *x*. This violates the monotonic-read consistency condition that any successive read operation on *x* must return the same (previously read) value or a more recent one.
(b): It **does not** satisfy **monotonic writes consistency** as the write $WS(x_1;x_3)$ in L1 only considers the initial write $W(x_1)$ in L1 and not the write on L2 $WS(x_1;x_2)$.
(c): **Yes**, it does satisfy **read-your-writes** consistency. The read $R(x_2)$ in L1 returns the write of x_2 which in turn considers the effect of the write $W(x_1)$ performed in L2 as indicated by the write operation $WS(x_1;x_2)$ in L1.
(d): **Yes**, it does satisfy **write-follows-reads** consistency as the last write in $WS(x_2;x_3)$ in L2 is consistent with the last value read in L1 $R(x_2)$.

2.
*(a) Explain the difference between a **primary-backup remote-write** and a **primary-backup local-write** protocol in terms of whether the data or the operations are sent from one replica to another and when that is done.*
(b) Which of the two protocols requires the backups to perform the write operations before the client initiating the write operation is informed that the writes are completed? How does this affect the response time seen by the initiating client and what the client can do?

(a) In a **primary-backup remote write** protocol, when a process *P* wants to perform an operation on a data item, it **forwards** the operation to the primary server. The primary server updates its local copy, forwards the update to the other servers and only after the update is performed in all the servers, the primary sends back an acknowledgment to process *P*.

In a **primary-backup local-write** protocol, the primary copy is allowed to migrate to processes. When a process *P* wants to perform an operation on a data item, it first locates the primary copy of the item, movies it to its own location, performs all the changes it needs to do, then the changes are propagated to the other servers.

(b) The **primary-backup remote write** protocol. It affects the response time as the initiating client has to block until it receives the acknowledgement that its write was performed.

*3. Consider a k **fault-tolerant group** containing **seven** process replicas.*
*(a) If clocks are **synchronized**, how many faults can be tolerated if the replicas are subject to **crash** faults? Justify your answer.*
*(b) If clocks are synchronized, how many faults can be tolerated if the replicas are subject to **arbitrary** (**Byzantine**) faults? Justify your answer.*
*(c) If clocks are **not synchronized**, how many faults can be tolerated if the replicas are subject to **crash** faults? Justify your answer.*
*(d) If clocks are not synchronized, how many faults can be tolerated if the replicas are subject to **arbitrary** (**Byzantine**) faults? Justify your answer.*

(a) The system can tolerate up to **6** faults. If clocks are **already synchronized**, up to 6 replicas can crash, i.e., stop responding, and we can still have 1 replica that is good and will produce the right result. (n=7=k+1 → **k=6**)

(b) The system can tolerate up to **3** faults. If clocks are **already synchronized**, a majority k+1 = 4 is needed to outvote k = 3 bad processes. In order that the results can be compared, all processes need to produce their results at approximately the same time, which synchronized clocks support. (n=7=2k+1 → **k=3**)

(c) The system can tolerate up to **6** faults. One process is needed to perform the operations and produce the result. Bad results are not produced, so any one result is good. (n=7=k+1 → **k=6**)

(d) The system can tolerate up to **2** faults. (n=7=3k+1 → **k=2**). There are several good justifications, given below:

- If clocks are not synchronized, we need 3k+1 processes to synchronize them, so that the values being voted are produced at (approximately) the same time and, thus, can be compared.
- If clocks are not synchronized, two different good processes can produce two different results because, e.g., the results are not produced at (approximately) the same time. A bad process can send two different results to each of the two good processes to make each of them think they won the vote, whereas actually they produced two different results.
- If clocks are not synchronized, two different good processes can produce two different results because, e.g., the results are not produced at

(approximately) the same time. If n = 3k+1 and there are k bad processes, then n-k = (3k+1)-k = 2k+1 process are good. Because good processes can produce different results, we need a majority of the 2k+1 processes to be good, so that k+1 good processes outvote k good processes.

*4. To protect **Remote Procedure Call (RPC)** against server crashes, you need to decide whether to support at-most-once semantics or at-least-once semantics for the application.*
*(a) Give an example application in which you would choose **at-most-once** semantics and explain why.*
*(b) To protect RPC against server crashes, what additional mechanism(s) would you use to provide **at-most-once** semantics?*
*(c) Give an example application in which you would choose **at-least-once** semantics and explain why.*
*(d) To protect RPC against server crashes, what additional mechanism(s) would you use to provide **at-least-once** semantics?*

(a) **Remote banking** is a good example in which at-most-once semantics is necessary. If it is not implemented you may end with duplicated transactions.
(b) For **remote banking** using **at-most-once** semantics we need to provide transactional mechanisms. In addition we could provide a mechanism to detect duplicated messages. This can be achieved as follows:

- Every time a user wants to perform a remote banking operation, it establish a session with the server. This session should be unique and a session ID is associated with the session.
- Every operation the user does during such session should also have a unique identifier, i.e., a unique operation ID.
- In this way, if the client accidently sends two identical (duplicated) requests, the server can detect it and perform a single request.

(c) **Compiling a program** is an example in which at-least-once semantics is desired, as there is no harm is compiling a program again.
(d) We as clients could retransmit requests in case there is timeout from the server side. As servers, if they crash, as soon as they recover they should retry any incomplete operation they may have log as incomplete.