Institute of Computer Science, University of Bern
Distributed Algorithms, Spring 2020

Prof. Christian Cachin
Jovana Mićić

Solution for exercise 6

6.1 Atomic register execution (5pt)

Study Algorithm 4.9 [CGR11, Sec. 4.4.4] for an (N, N) atomic register in the fail-stop model (with a perfect failure detector $\mathcal{P}$). It illustrates how timestamp/process identifiers are used for linearizing the write operations.

Describe (or draw) two executions, A and B, of this protocol with five processes p, q, r, s, and t. Decide on an order among $\{p, q, r, s, t\}$. Initially the register stores a value $\perp$. Process p starts operation *write*$_p(x)$ at the same time as process q starts *write*$_q(y)$. Processes r and s execute *read*$_r()$ and *read*$_s()$; both of these operations are concurrent to the two *writes*. Process t executes *read*$_t()$ such that both *writes* precede this operation.

a) In execution A, we observe *read*$_r() \rightarrow x$ and *read*$_s() \rightarrow y$.

b) In execution B, we observe *read*$_r() \rightarrow y$ and *read*$_s() \rightarrow x$.

What does *read*$_t()$ return in A and in B?

Solution. Lets assume that processes are ordered as $\{p, q, r, s, t\}$ and these are corresponding ranks $\{1, 2, 3, 4, 5\}$.

Figure 1 shows the possible steps of the execution A. Processes p and q start writing at the same time and broadcast a message to other processes. It may happen that the write message from process q to process r is delayed (1). Because of that, process r reads value x. Process s reads value y because process q has a higher rank than process p. For the same reason, process t reads value y.

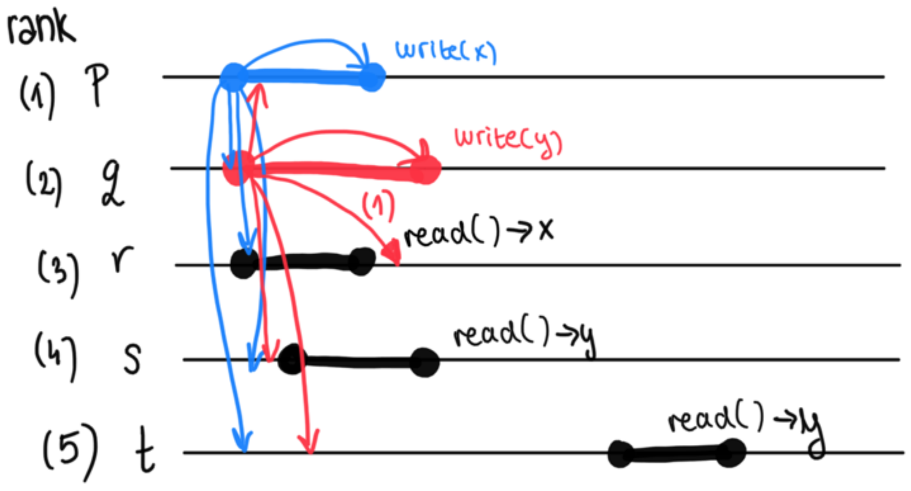


Figure 1. Execution A.

Figure 2 shows the possible steps of the execution B. Processes p and q start writing at the same time and broadcast a message to other processes. It may happen that the write message from process q to process s is delayed (1). Because of that, process s reads value x. Process r reads value y because process q has a higher rank than process p. For the same reason, process t reads value y.

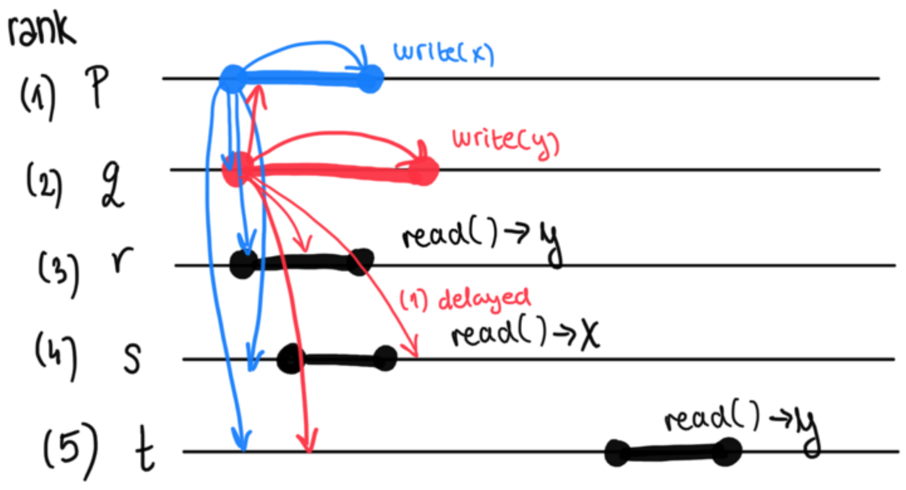
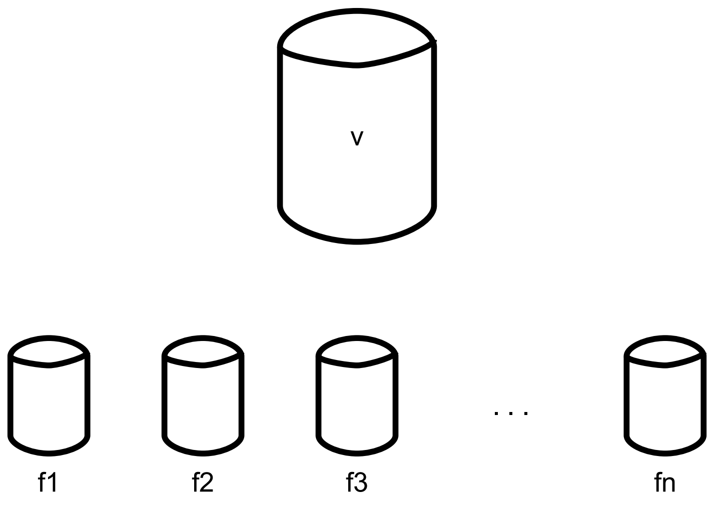


Figure 2. Execution B.

6.2 Erasure-coded storage (5pt)

Practical distributed storage systems often use *erasure coding* to reduce the space taken by redundantly stored data.

An (k, n)-*erasure code* maps a large data value (such as a disk sector or a file, which consists of k information blocks) to n so-called *fragments* of smaller size. The erasure code ensures that an encoded value can be reconstructed from any k fragments (or code blocks). Practical erasure codes are based on Reed-Solomon codes over finite fields, for example.



⟹

More precisely, an (k, n)-erasure code with domain $\mathcal{V}$ is given by two algorithms *encode* and *decode*:

- Algorithm $\mathit{encode}_{k,n}(v)$, when given a (large) value $v \in \mathcal{V}$, produces a vector $[f_1, \ldots, f_n]$ of n fragments, with $f_i \in \mathcal{F}$. A fragment is typically much smaller than the input, and any k fragments contain all information of v, that is, $|\mathcal{V}| \approx k|\mathcal{F}|$. (In other words, we consider only *maximum-distance separable codes* here.)
- For an n-vector $F \in \big(\mathcal{F} \cup \{\perp\}\big)^n$, whose entries are either fragments or the symbol $\perp$, algorithm $\mathit{reconstruct}_{k,n}(F)$ outputs a value $v \in \mathcal{V}$ or $\perp$. Output $\perp$ means that the reconstruction failed. In other words, if one computes $F \leftarrow \mathit{encode}_{k,n}(v)$ for some $v \in \mathcal{V}$ and then erases up to $n - k$ entries in F by setting them to $\perp$, algorithm $\mathit{reconstruct}_{k,n}(F)$ outputs v. Otherwise, the algorithm outputs $\perp$.

The replication method of the storage protocols considered in the class can be seen as a $(1, n)$-erasure code. The RAID-5 encoding scheme found in practical disk controllers corresponds to an $(n - 1, n)$-erasure codes, which can be implemented solely by XOR operations. (Analogously, RAID-6 implements an $(n - 2, n)$-erasure code.)
Tasks:

a) Consider a distributed storage system of n nodes, of which $f < n/2$ may fail by crashing. Pick a suitable erasure code and describe the *storage efficiency* of the system, i.e., the ratio of stored information and provisioned space that must be provisioned.

b) Modify the majority-voting protocol implementing a regular register (Algorithm 4.2 [CGR11]) to implement an erasure-coded *safe* register.

c) Why is it difficult to extend this protocol to regular semantics?

Solution.

a) In the distributed storage system of n nodes, where $f < n/2$ may crash, the stored value must be recoverable, even if f of the processes that acknowledge the write, crash later. Suitable erasure code in this case would be $(f + k, n)$-erasure code. Therefore, the resilience of the described system is $n \geq 2f + k$. The storage efficiency of the system is k/n, where k is the size of a fragment and n is the total number of stored fragments.

b) Algorithm 1 shows the modification of Algorithm 4.2 [CGR11]. A value is first encoded using an $(f + k, n)$-erasure code and then sent to other process as a WRITE message. The *write* operation completes when $f + k$ process acknowledge writing. When a process reads the value, it waits for k reads from processes and then reconstructs value.

c) The difficulty of extending this protocol to the regular semantics is that the data of some value v is spread across different processes. In the case of many concurrent writes with different values, it may happen that each response gives a fragment from another value. In that case, none of the values being written can be reconstructed. This difficulty does not arise with the protocol based on replication. (Replication can be viewed as an $(1, n)$-erasure code.)

References

[CGR11] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming (Second Edition)*, Springer, 2011.

Algorithm 1 Modification of Algorithm Majority Voting (Algorithm 4.2).

```
**upon event** $\langle$ *onrr-Write* $\mid v \rangle$ **do**
    $wts := wts + 1$;
    $acks := 0$;
    $\mathcal{F} := encode_{k,n}(v)$;
    **forall** $p \in \pi$ **do**
        **trigger** $\langle$ *pl-Send* $\mid p$, [WRITE, $wts$, $\mathcal{F}[p]$] $\rangle$;

**upon even** $\langle$ *pl-Deliver* $\mid q$, [ACK, $ts'$] $\rangle$ **such that** $ts' = wts$ **do**
    $acks := acks + 1$;
    **if** $acks \geq f + k$ **then**
        $acks := 0$;
        **trigger** $\langle$ *onrr-WriteReturn* $\rangle$;

**upon event** $\langle$ *pl-Deliver* $\mid q$, [VALUE, $r$, $ts'$, $v'$] $\rangle$ **such that** $r = rid$ **do**
    $readlist[q] := (ts', v')$;
    **if** $\#(readlist) \geq k$ **then**
        $v := reconstruct_{k,n}(readlist)$;
        **trigger** $\langle$ *onrr-ReadReturn* $\mid v \rangle$;
    **else**
        **trigger** $\langle$ *onrr-ReadReturn* $\mid \perp \rangle$;
```