

4 Shared Memory

4.1 Registers

Motivation. *Registers* or *read/write registers* are a simple and useful abstraction for shared data storage. In the so-called *shared-memory model*, processes access concurrent data objects asynchronously. *Wait-free* implementations of such objects guarantee that any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes.

Registers may be used for communication and process synchronization, but because of their limited operations, objects with richer and more powerful operations have also been considered, like (binary) test-and-set operations or (multi-valued) read-modify-write operations [HS08].

In practice, the storage may take the form of shared memory (RAM) in a multiprocessor system or storage devices (disks) connected to clients over a network.

Definitions. Registers were formalized by Lamport [Lam86].

Definition 1 (Register). A *register* x is characterized by two operations:

write(x, v) \rightarrow ok: writes a value v to register x and returns the symbol ok;

read(x) \rightarrow v : reads the register x and returns its value v .

W.l.o.g. we consider only one register and assume that every process executes at any time only one operation. An operation is *invoked* at some point in time and *returns* at a later point in time. When a write operation with value v returns ok, we say that it *writes* v .

The *sequential specification* of a register requires that each read operation returns the value written by the most recent preceding write operation.

Definition 2 (Precedence). For two operations o_1 and o_2 , we say that

- o_1 *precedes* o_2 whenever o_1 returns before o_2 is invoked (they are *sequential*), and
- o_1 *is concurrent with* o_2 when neither operation precedes the other one.

Many variations of registers are considered:

Domain: binary and multi-valued;

Concurrent access: single-reader single-writer (SRSW), multiple-readers single-writer (MRSW), and multiple-readers multiple-writers (MRMW);

Semantics: safe, regular, and atomic (see next).

Semantics. The most important aspect of a register is its behavior under concurrent access. W.l.o.g. assume there is an initial *write* operation.

Safe: A register is *safe* when every *read* not concurrent with a *write* returns the most recently *written* value. *Reads* that are concurrent with at least one *write* may return any value in the domain.

Regular: A register is *regular* if it is *safe* and any *read* concurrent with a *write* returns either the most recently *written* value or a *concurrently written* value.

Atomic: A register is *atomic* whenever the *read* and *write* operations are *linearizable*, which means that there exists an *equivalent* totally ordered *sequential* execution of them. In other words, there exists a permutation π of all invocations and responses in the execution such that the sequential specification of every register holds and such that for any two operations o_1 and o_2 where o_1 precedes o_2 in the execution, o_1 also precedes o_2 in π .

(For one writer only, a simpler definition is to require that the register is *regular* and ensures that if an operation r_1 returns a value written by w_1 , an operation r_2 returns a value written by w_2 , and r_1 precedes r_2 , then w_2 does not precede w_1 .)

4.2 Reductions among Registers

Most variations of registers are equally powerful in the sense that they can be reduced to each other in the shared-memory model. In particular, a multi-valued MRMW atomic register can be implemented with binary SRSW safe registers [HS08].

Algorithm 3 (Emulation of a multi-valued MRSW register from binary MRSW registers). Given k binary registers b_0, b_1, \dots, b_{k-1} with operations *read* and *write*, simulate a multi-valued register B with operations *READ* and *WRITE*. The domain of B is $\{0, \dots, k-1\}$. The emulation uses a unary encoding. Initially, all binary registers are 0, except for $b_0 = 1$.

<pre> WRITE(B, v): for i = 0, 1, ..., k - 1 do if i = v then write(b_i, 1) else write(b_i, 0) return ok </pre>	<pre> READ(B): for i = 0, 1, ..., k - 1 do if read(b_i) = 1 then return i return 0 </pre>
--	---

Theorem 4. *Algorithm 3 implements a multi-valued MRSW register with safe semantics, but not regular semantics.*

To provide regular semantics, we change only the *WRITE* algorithm so that the register contains either the previous or the concurrently written value at any time:

```

WRITE(B, v):
  write(b_v, 1)
  for i = v - 1, v - 2, ..., 0 do
    write(b_i, 0)
  return ok

```

Theorem 5. *Algorithm 3 with the modified WRITE implements a multi-valued MRSW regular register from binary MRSW regular registers, and a multi-valued MRSW atomic register from binary MRSW atomic registers.*

4.3 Fault-tolerant Distributed Implementations

Here we consider a *fault-tolerant* distributed implementation of a register by n storage servers $\mathcal{P} = \{P_1, \dots, P_n\}$ in an asynchronous network. Some servers may fail by crashing silently. A protocol emulates the shared data object despite the failure of some servers. The data is read and written by clients through sending messages to the servers over an asynchronous network that provides a reliable point-to-point FIFO channel between every client and every server. The servers do not communicate with each other. For tolerating faults, the data of the register is stored collectively by all servers, using replication or erasure coding. We assume that clients do not fail. Wait-free termination here means that a client completes every operation independently from server failures and independently of the speed of other clients.

Let \mathcal{Q} be a quorum system on \mathcal{P} . The next algorithm implements an abstract fault-tolerant data storage system without a centralized controller. It tolerates the failure of a set $\mathcal{P} \setminus Q$ of servers for any $Q \in \mathcal{Q}$. The clients need only talk to a quorum of servers for accessing the data.

Algorithm 6 (Distributed implementation of a MRSW regular register). A register x stores a value v ; clients are a single reader and a multiple writers. The writer maintains a timestamp τ . Every server P_i stores local copies v_i and τ_i .

- For writing v to x , the writer increments τ , and executes the following steps: it picks a quorum Q , sends (write, v, τ) to all $P_i \in Q$, and waits for an `ack` from all servers in Q ; if not enough `ack` messages arrive in time, the writer repeats those steps with a different quorum until `ack` messages arrive from all servers in the quorum. Then it outputs `ok` and terminates the write.

Upon receiving (write, v, τ) , a server P_i sets $(v_i, \tau_i) \leftarrow (v, \tau)$ and returns an `ack` message to the client.

- To read from x , a reader executes the following steps: it picks a quorum Q , sends a `(read)` message to all $P_i \in Q$, and waits for a message $(\text{value}, v_i, \tau_i)$ from all servers in Q ; if not enough `value` messages arrive in time, the reader repeats those steps with a different quorum until `value` messages arrive from all servers in the quorum. Then it selects the value from the message with the highest timestamp and outputs that value.

Upon receiving the `(read)` message from a client, a server P_i returns the message $(\text{value}, v_i, \tau_i)$ to the client.

The message complexity of every operation is $2|Q|$.

Theorem 7. *Algorithm 6 is a wait-free implementation of a MRSW regular register on \mathcal{P} .*

Proof sketch. The quorum used by the reader has non-empty intersection with the quorum used in the most recent write that precedes the read. If a write exists concurrent with some read, the reader may also return the concurrently written value. Wait-freedom follows because there exists a quorum of correct servers. \square

Algorithm 6 can be modified to emulate an SRSW *atomic* register: The reader additionally

maintains a value/timestamp pair (v, τ) . If the reader receives a `value` message containing a higher timestamp than τ , it sets (v, τ) to the value/timestamp pair from the message. Finally, the reader outputs v . This emulates an atomic register only for a single reader. When there are multiple clients C_1, C_2, \dots reading from the register, synchronizing the value/timestamp pair between the readers requires an additional step.

The next algorithm is atomic for multiple readers; it synchronizes the reader timestamp by causing clients to write during a read operation (one can show that this is necessary).

Algorithm 8 (Distributed implementation of a MRSW atomic register [ABND95, AW04]).

Let the *writer* be C_w ; it stores a timestamp τ . Every P_i maintains (v_i, τ_i) .

```

write(x, v):                                     // executed by writer  $C_w$  only
     $\tau \leftarrow \tau + 1$ 
    send (write, v,  $\tau$ ) to  $P_1, \dots, P_n$ 
    wait for an (ack) message from all  $P_i$  in some quorum  $Q \in \mathcal{Q}$ 
    return ok

read(x):                                         // client  $C_j$ 
    send (read) to  $P_1, \dots, P_n$ 
    wait for (value,  $v_i, \tau_i$ ) messages from all  $P_i$  in some quorum  $Q \in \mathcal{Q}$ 
    let  $(v, \tau)$  be the received  $(v_i, \tau_i)$  pair with the largest  $\tau_i$ 
    send (write, v,  $\tau$ ) to  $P_1, \dots, P_n$ 
    wait for an (ack) message from all  $P_i$  in some quorum  $Q \in \mathcal{Q}$ 
    return v

upon receiving (write, v,  $\tau$ ) from  $C_w$ :         // server  $P_i$ 
    if  $\tau > \tau_i$  then
         $(v_i, \tau_i) \leftarrow (v, \tau)$ 
        send (ack) to  $C_w$ 

upon receiving (read) from  $C_j$ :                 // server  $P_i$ 
    send (value,  $v_i, \tau_i$ ) to  $C_j$ 

```

Theorem 9. Algorithm 8 implements a MRSW atomic register on a quorum system \mathcal{Q} .

Proof sketch. The only problem is a *write* operation w concurrent with multiple *reads*, say, r_1 and r_2 . In this case, observe that if $r_1 \rightarrow v$ precedes r_2 or $w(v)$ precedes r_2 , then $r_2 \rightarrow v$ because r_1 and w both write to a quorum that intersects with the quorum from which r_2 obtains its value. □

References

- [ABND95] H. Attiya, A. Bar-Noy, and D. Dolev, *Sharing memory robustly in message-passing systems*, Journal of the ACM **42** (1995), no. 1, 124–142.
- [AW04] H. Attiya and J. Welch, *Distributed computing: Fundamentals, simulations and advanced topics*, second ed., Wiley, 2004.
- [HS08] M. Herlihy and N. Shavit, *The art of multiprocessor programming*, Morgan Kaufmann, 2008.
- [Lam86] L. Lamport, *On interprocess communication*, Distributed Computing **1** (1986), no. 2, 77–85, 86–101.