

## 4 Registers and Shared Memory

### 4.1 Read/Write Registers

**Motivation.** *Read/write registers* are a particularly 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.

R/W 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. Herlihy [Her91] defines a hierarchy of wait-free objects.

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.** R/W registers were formalized by Lamport [Lam86].

**Definition 4.1 (R/W Register).** A read/write register  $x$  is characterized by two operations:

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

*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  $\text{ok}$ , we say that it *writes*  $v$ .

**Definition 4.2 (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 R/W 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 R/W 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 R/W 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 R/W register is *atomic* (with one writer) whenever it 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  $w_1$ .

More generally (with multiple writers), we require that the *read* and *write* operations are *linearizable*, which means that there exists an *equivalent* totally ordered *sequential* execution of all *read* and *write* operations. More precisely, this means that there exists a permutation  $\pi$  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  $\pi$ .

## 4.2 Reductions Between R/W Registers

Most variations of R/W 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 [AW04].

**Algorithm 4.3 (Implementation of Multi-valued R/W Register from Binary R/W Registers).** Given  $k$  binary R/W registers  $b_0, b_1, \dots, b_{k-1}$  with operations *read* and *write*, simulate a multi-valued R/W register  $B$  with operations *READ* and *WRITE*. Initially, all binary registers are 0, except for  $b_0 = 1$ .

*WRITE*( $B, v$ ):

```
for  $i = 0, 1, \dots, k - 1$  do
  if  $i = v$  then
    write( $b_i, 1$ )
  else
    write( $b_i, 0$ )
return ok
```

*READ*( $B$ ):

```
for  $i = 0, 1, \dots, k - 1$  do
  if read( $b_i$ ) = 1 then
    return  $i$ 
return 0
```

**Lemma 4.4.** *Algorithm 4.3 implements a multi-valued R/W register with safe semantics, but not regular semantics.*

To provide regular semantics, we can change only the *WRITE* algorithm so that the register contains either the previous or the concurrently written value at any time. The modified *WRITE* algorithm is as follows:

```

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

```

### 4.3 Fault-tolerant Implementation

Here we consider a *fault-tolerant* distributed implementation of a R/W register by  $n$  storage servers  $\mathcal{P} = \{P_1, \dots, P_n\}$  in an asynchronous network. The data stored in the register is replicated at all servers, but some of them may fail by crashing silently. The data is read and written by clients through sending messages to the servers over an asynchronous point-to-point network. (The servers do not communicate with each other.) We assume that clients do not fail.

Define a quorum system  $\mathcal{Q}$  on  $\mathcal{P}$  and assume that any set  $\mathcal{P} \setminus Q$  of servers may fail, for any  $Q \in \mathcal{Q}$ . Then the clients need only talk to a quorum of servers for accessing the data. This provides an abstraction of fault-tolerant data storage, i.e., for files or disk blocks.

**Algorithm 4.5 (Fault-tolerant Implementation of SRSW Atomic R/W Register).** A register  $x$  stores a value  $v$ ; clients are a single reader and a single writer. The writer maintains a timestamp  $\tau$ . Every server  $P_i$  stores local copies  $v_i$  and  $\tau_i$ .

- For writing  $v$  to  $x$ , the writer increments  $\tau$ , and executes the following steps: it picks a quorum  $Q$ , sends  $(\text{write}, v, \tau)$  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 terminates the write.

Upon receiving  $(\text{write}, v, \tau)$ , a server  $P_i$  sets  $(v_i, \tau_i) \leftarrow (v, \tau)$  and returns an `ack` message.

- To read from  $x$ , the 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 as the result.

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

**Lemma 4.6.** *Algorithm 4.5 implements a SRSW atomic R/W register on a quorum system  $\mathcal{Q}$ .*

*Proof.* The quorum used by the writer has non-empty intersection with the quorum used by the reader. □

The algorithm works only for a single reader and for a single writer. The message complexity of every operation is  $2|Q|$ .

Suppose there are multiple clients  $C_1, C_2, \dots$  reading from the register. The above algorithm emulates a regular read/write register, but not an atomic one. (What happens with multiple readers or multiple writers?) The next algorithm is atomic for multiple readers, but involves that clients write during a read operation (one can show that this is necessary).

**Algorithm 4.7 (Fault-tolerant Implementation of a MRSW Atomic R/W Register [ABND95]).**

Let the *writer* be  $C_w$ ; it stores a timestamp  $\tau$ . Every  $P_i$  maintains  $(v_i, \tau_i)$ .

```

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  $C_j$ :       // server  $P_i$ 
  if  $\tau > \tau_i$  then
     $(v_i, \tau_i) \leftarrow (v, \tau)$ 
    send (ack) to  $C_j$ 

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

```

**Theorem 4.8.** *Algorithm 4.7 implements a MRSW atomic R/W register on a quorum system  $\mathcal{Q}$ .*

*Proof.* 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.  $\square$

## 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.
- [Her91] M. Herlihy, *Wait-free synchronization*, ACM Transactions on Programming Languages and Systems **11** (1991), no. 1, 124–149.
- [Lam86] L. Lamport, *On interprocess communication*, Distributed Computing **1** (1986), no. 2, 77–85, 86–101.