

3 Quorum Systems (cont.)

3.4 Read/Write Registers

Motivation. *Read/write registers* are a particularly simple and useful abstraction for shared data storage and interprocess communication in shared-memory multiprocessor systems. In the shared-memory model, processes typically 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 on 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.

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

Definition 3.1 (R/W register). A read/write register x is characterized by two operations:

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

read(x) $\rightarrow v$: reads the value of x , returns it, and saves it in v .

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

Definition 3.2 (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: An 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: An R/W register is *regular* if it is *safe* and any *read* concurrent with a *write* returns either the most recently *written* value or the *concurrently written* value.

atomic: An R/W register is *atomic* if it is *regular* and if the *read* and *write* operations are linearizable. Informally, this means that there exists an “equivalent” totally ordered *sequential* execution of all *read* and *write* operations. Formally, we require that if an operation $read_1$ returns a value written by $write_1$, an operation $read_2$ returns a value written by $write_2$, and $read_1$ precedes $read_2$, then $write_2$ does not precede $write_1$.

Reductions. 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 [AW98].

3.5 Fault-tolerant Implementation

Here we consider the *fault-tolerant* distributed implementation of R/W registers by n servers in an asynchronous network. This provides an abstraction of fault-tolerant data storage, i.e., for files or disk blocks.

Recall Algorithm 3.2; it implements a SRSW atomic register using quorums. (What happens with multiple readers or multiple writers?)

Algorithm 3.3 (Implementation of a MRSW Atomic R/W Register [ABND95]). The system consists of servers $\mathcal{P} = \{P_1, \dots, P_n\}$, a quorum system \mathcal{Q} on \mathcal{P} , and clients C_1, C_2, \dots , among them a designated *writer* C_w . They are linked by asynchronous point-to-point channels. C_w stores a timestamp τ and P_i maintains (x_i, τ_i) .

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

```

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

upon receiving (write,  $x, \tau$ ) from  $C_j$ : // server  $P_i$ 
  if  $\tau > \tau_i$  then
     $(x_i, \tau_i) \leftarrow (x, \tau)$ 
    send (ack) to  $C_j$ 

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

```

Theorem 3.4. *Algorithm 3.3 implements a MRSW atomic R/W register among a group \mathcal{P} of servers, in the presence of a set $\mathcal{P} \setminus Q$ of faulty servers, for any $Q \in \mathcal{Q}$.*

Proof. The only problem is a write concurrent with multiple reads, say, $read_1$ and $read_2$. In this case, observe that if $read_1 \rightarrow v$ or $write(v)$ precedes $read_2$, then $read_2 \rightarrow v$ because $read_1$ and $write$ both write to a quorum that intersects with the quorum from which $read_2$ 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.
- [AW98] H. Attiya and J. Welch, *Distributed computing: Fundamentals, simulations and advanced topics*, McGraw-Hill, London, 1998.
- [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.