Implementing Shared Registers in Asynchronous Message-Passing Systems, 1995; Attiya, Bar-Noy, Dolev

Eric Ruppert, York University, www.cse.yorku.ca/∼ruppert

INDEX TERMS: distributed computing, shared memory, read-write register, message-passing, fault-tolerance.

SYNONYMS: simulation, emulation.

1 PROBLEM DEFINITION

A distributed system is comprised of a collection of n processes which communicate with one another. Two means of interprocess communication have been heavily studied. *Message-passing systems* model computer networks where each process can send information over message channels to other processes. In *shared-memory systems*, processes communicate less directly by accessing information in shared data structures. Distributed algorithms are often easier to design for shared-memory systems because of their similarity to single-process system architectures. However, many real distributed systems are constructed as message-passing systems. Thus, a key problem in distributed computing is the implementation of shared memory in message-passing systems. Such implementations are also called simulations or emulations of shared memory.

The most fundamental type of shared data structure to implement is a *(read-write) register*, which stores a value, taken from some domain D. It is initially assigned a value from D and can be accessed by two kinds of operations, read and write(v), where $v \in D$. A register may be either *single-writer*, meaning only one process is allowed to write it, or *multi-writer*, meaning any process may write to it. Similarly, it may be either *single-reader* or *multi-reader*. Attiya and Welch [4] give a survey of how to build multi-writer, multi-reader registers from single-writer, single-reader ones.

If reads and writes are performed one at a time, they have the following effects: a read returns the value stored in the register to the invoking process, and a write(v) changes the value stored in the register to v and returns an acknowledgement, indicating that the operation is complete. When many processes apply operations concurrently, there are several ways to specify a register's behaviour [14]. A single-writer register is *regular* if each read returns either the argument of the write that completed most recently before the read began or the argument of some write operation that runs concurrently with the read. (If there is no write that completes before the read begins, the read may return either the initial value of the register or the value of a concurrent write operation.) A register is *atomic* (or linearizable) if each operation appears to take place instantaneously. More precisely, for any concurrent execution, there is a total order of the operations such that each read returns the value written by the last write that precedes it in the order (or the initial value of the register, if there is no such write). Moreover, this total order must be consistent with the temporal order of operations: if one operation finishes before another one begins, the former must precede the latter in the total order. Atomicity is a stronger condition than regularity, but it is possible to implement atomic registers from regular ones with some complexity overhead [12].

This article describes the problem of implementing registers in an asynchronous message-passing system in which processes may experience crash failures. Each process can send a message, containing a finite string, to any other process. To make the descriptions of algorithms more uniform,

it is often assumed that processes can send messages to themselves. All messages are eventually delivered. In the algorithms described below, senders wait for an acknowledgement of each message before sending the next message, so it is not necessary to assume that the message channels are first-in-first-out. The system is totally asynchronous: there is no bound on the time required for a message to be delivered to its recipient or for a process to perform a step of local computation. A process that fails by crashing stops executing its code, but other processes cannot distinguish between a process that has crashed and one that is running very slowly. (Failures of message channels [3] and more malicious kinds of process failures [15] have also been studied.)

A *t-resilient* register implementation provides programmes to be executed by processes to simulate read and write operations. These programmes can include any standard control structures and accesses to a process's local memory, as well as instructions to send a message to another process and to read the process's buffer, where incoming messages are stored. The implementation should also specify how the processes' local variables are initialized to reflect any initial value of the implemented register. In the case of a single-writer register, only one process may execute the write programme. A process may invoke the read and write programmes repeatedly, but it must wait for one invocation to complete before starting the next one. In any such execution where at most t processes crash, each of a process's invocations of the read or write programme should eventually terminate. Each read operation returns a result from the set D, and these results should satisfy regularity or atomicity.

Relevant measures of algorithm complexity include the number of messages transmitted in the system to perform an operation, the number of bits per message, and the amount of local memory required at each process. One measure of time complexity is the time needed to perform an operation, under the optimistic assumption that the time to deliver messages is bounded by Δ and local computation is instantaneous (although algorithms must work correctly even without these assumptions).

2 KEY RESULTS

Implementing a Regular Register

One of the core ideas for implementing shared registers in message-passing systems is a construction that implements a regular single-writer multi-reader register. It was introduced by Attiya, Bar-Noy and Dolev [3] and made more explicit by Attiya [2]. A write(v) sends the value v to all processes and waits until a majority of the processes ($\lfloor \frac{n}{2} \rfloor + 1$, including the writer itself) return an acknowledgement. A reader sends a request to all processes for their latest values. When it has received responses from a majority of processes, it picks the most recently written value among them. If a write completes before a read begins, at least one process that answers the reader has received the write's value prior to sending its response to the reader. This is because any two sets that each contain a majority of the processes must overlap. The time required by operations when delivery times are bounded is 2Δ.

This algorithm requires the reader to determine which of the values it receives is most recent. It does this using *timestamps* attached to the values. If the writer uses increasing integers as timestamps, the messages grow without bound as the algorithm runs. Using the bounded timestamp scheme of Israeli and Li [13] instead yields the following theorem.

Theorem 1 (Attiya [2]). *There is an $\lceil \frac{n-2}{2} \rceil$-resilient implementation of a regular single-writer, multi-reader register in a message-passing system of n processes. The implementation uses $\Theta(n)$ messages per operation, with $\Theta(n^3)$ bits per message. The writer uses $\Theta(n^4)$ bits of local memory and each reader uses $\Theta(n^3)$ bits.*

Theorem 1 is optimal in terms of fault-tolerance. If $\lceil \frac{n}{2} \rceil$ processes can crash, the network can be partitioned into two halves of size $\lfloor \frac{n}{2} \rfloor$, with messages between the two halves delayed indefinitely.

A write must terminate before any evidence of the write is propagated to the half not containing the writer, and then a read performed by a process in that half cannot return an up-to-date value. For $t \geq \lceil \frac{n}{2} \rceil$, registers can be implemented in a message-passing system only if some degree of synchrony is present in the system. The exact amount of synchrony required was studied by Delporte-Gallet *et al.* [6].

Theorem 1 is within a constant factor of the optimal number of messages per operation. Evidence of each write must be transmitted to at least $\lceil \frac{n}{2} \rceil - 1$ processes, requiring $\Omega(n)$ messages; otherwise this evidence could be obliterated by crashes. A write must terminate even if only $\lfloor \frac{n}{2} \rfloor + 1$ processes (including the writer) have received information about the value written, since the rest of the processes could have crashed. Thus, a read must receive information from at least $\lceil \frac{n}{2} \rceil$ processes (including itself) to ensure that it is aware of the most recent write operation.

A t-resilient implementation, for $t < \lceil \frac{n}{2} \rceil$, that uses $\Theta(t)$ messages per operation is obtained by the following adaptation. A set of $2t+1$ processes is preselected to be data storage servers. Writes send information to the servers, and wait for $t+1$ acknowledgements. Reads wait for responses from $t+1$ of the servers and choose the one with the latest timestamp.

Implementing an Atomic Register

Attiya, Bar-Noy and Dolev [3] gave a construction of an atomic register in which readers forward the value they return to all processes and wait for an acknowledgement from a majority. This is done to ensure that a read does not return an older value than another read that precedes it. Using unbounded integer timestamps, this algorithm uses $\Theta(n)$ messages per operation. The time needed per operation when delivery times are bounded is 2Δ for writes and 4Δ for reads. However, their technique of bounding the timestamps increases the number of messages per operation to $\Theta(n^2)$ (and the time per operation to 12Δ). A better implementation of atomic registers with bounded message size is given by Attiya [2]. It uses the regular registers of Theorem 1 to implement atomic registers using the "handshaking" construction of Haldar and Vidyasankar [12], yielding the following result.

Theorem 2 (Attiya [2]). *There is an $\lceil \frac{n-2}{2} \rceil$-resilient implementation of an atomic single-writer, multi-reader register in a message-passing system of n processes. The implementation uses $\Theta(n)$ messages per operation, with $\Theta(n^3)$ bits per message. The writer uses $\Theta(n^5)$ bits of local memory and each reader uses $\Theta(n^4)$ bits.*

Since atomic registers are regular, this algorithm is optimal in terms of fault-tolerance and within a constant factor of optimal in terms of the number of messages. The time used when delivery times are bounded is at most 14Δ for writes and 18Δ for reads.

3 APPLICATIONS

Any distributed algorithm that uses shared registers can be adapted to run in a message-passing system using the implementations described above. This approach yielded new or improved message-passing solutions for a number of problems, including randomized consensus [1], multi-writer registers [4], and snapshot objects [[[**cross-ref to article on snapshots**]]]. The reverse simulation is also possible, using a straightforward implementation of message channels by single-writer, single-reader registers. Thus, the two asynchronous models are equivalent, in terms of the set of problems that they can solve, assuming only a minority of processes crash. However there is some complexity overhead in using the simulations.

If a shared-memory algorithm is implemented in a message-passing system using the algorithms described here, processes must continue to operate even when the algorithm terminates, to help other processes execute their reads and writes. This cannot be avoided: if each process must stop

taking steps when its algorithm terminates, there are some problems solvable with shared registers that are not solvable in the message-passing model [5].

Using a majority of processes to "validate" each read and write operation is an example of a quorum system, originally introduced for replicated data by Gifford [10]. In general, a quorum system is a collection of sets of processes, called quorums, such that every two quorums intersect. Quorum systems can also be designed to implement shared registers in other models of message-passing systems, including dynamic networks and systems with malicious failures. For examples, see [7, 9, 11, 15].

4 OPEN PROBLEMS

Although the algorithms described here are optimal in terms of fault-tolerance and message complexity, it is not known if the number of bits used in messages and local memory is optimal. The exact time needed to do reads and writes when messages are delivered within time Δ is also a topic of ongoing research. (See, for example, [8].) As mentioned above, the simulation of shared registers can be used to implement shared-memory algorithms in message-passing systems. However, because the simulation introduces considerable overhead, it is possible that some of those problems could be solved more efficiently by algorithms designed specifically for message-passing systems.

5 EXPERIMENTAL RESULTS

None.

6 DATA SETS

None.

7 URL to CODE

None.

8 CROSS REFERENCES

Distributed Snapshots.

[[[Entry editors: please feel free to add others. I think there is one on linearizability and one on Lamport's registers paper.]]]

9 RECOMMENDED READING

[1] J. Aspnes, *Randomized protocols for asynchronous consensus*, Distributed Computing, 16 (2003), pp. 165–175.

[2] H. Attiya, *Efficient and robust sharing of memory in message-passing systems*, Journal of Algorithms, 34 (2000), pp. 109–127.

[3] H. Attiya, A. Bar-Noy, and D. Dolev, *Sharing memory robustly in message-passing systems*, J. Assoc. Comput. Mach., 42 (1995), pp. 124–142.

[4] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, Wiley-Interscience, second ed., 2004.

[5] B. Chor and L. Moscovici, *Solvability in asynchronous environments*, in Proc. 30th Symposium on Foundations of Computer Science, 1989, pp. 422–427.

[6] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg, *The weakest failure detectors to solve certain fundamental problems in distributed computing*, in Proc. 23rd ACM Symposium on Principles of Distributed Computing, 2004, pp. 338–346.

[7] S. Dolev, S. Gilbert, N. A. Lynch, A. A. Shvartsman, and J. L. Welch, *GeoQuorums: Implementing atomic memory in mobile ad hoc networks*, Distributed Computing, 18 (2005), pp. 125–155.

[8] P. Dutta, R. Guerraoui, R. R. Levy, and A. Chakraborty, *How fast can a distributed atomic read be?*, in Proc. 23rd ACM Symposium on Principles of Distributed Computing, 2004, pp. 236–245.

[9] B. Englert and A. A. Shvartsman, *Graceful quorum reconfiguration in a robust emulation of shared memory*, in Proc. 20th IEEE International Conference on Distributed Computing Systems, 2000, pp. 454–463.

[10] D. K. Gifford, *Weighted voting for replicated data*, in Proc. 7th ACM Symposium on Operating Systems Principles, 1979, pp. 150–162.

[11] S. Gilbert, N. Lynch, and A. Shvartsman, *Rambo II: rapidly reconfigurable atomic memory for dynamic networks*, in Proc. International Conference on Dependable Systems and Networks, 2003, pp. 259–268.

[12] S. Haldar and K. Vidyasankar, *Constructing 1-writer multireader multivalued atomic variables from regular variables*, J. Assoc. Comput. Mach., 42 (1995), pp. 186–203.

[13] A. Israeli and M. Li, *Bounded time-stamps*, Distributed Computing, 6 (1993), pp. 205–209.

[14] L. Lamport, *On interprocess communication, Part II: Algorithms*, Distributed Computing, 1 (1986), pp. 86–101.

[15] D. Malkhi and M. Reiter, *Byzantine quorum systems*, Distributed Computing, 11 (1998), pp. 203–213.