FTRepMI: Fault-tolerant, Sequentially-consistent Object Replication for Grid Applications

Ana-Maria Oprescu, Thilo Kielmann, and Wan Fokkink

Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands
{amo,kielmann,wanf}@cs.vu.nl

Abstract. We introduce FTRepMI, a simple fault-tolerant protocol for providing sequential consistency amongst replicated objects in a grid, without using any centralized components. FTRepMI supports dynamic joins and graceful leaves of processes holding a replica, as well as fail-stop crashes. Performance evaluation shows that FTRepMI behaves efficiently, both on a single cluster and on a distributed cluster environment.

1 Introduction

Object replication is a well-known technique to improve the performance of parallel object-based applications [13]. Java's remote method invocation (RMI) enables methods of remote Java objects to be invoked from other Java virtual machines, possibly on different hosts. Maassen introduced Replicated Method Invocation (RepMI) [11], which was implemented in Manta [12]. He obtained a significant performance improvement by combining object replication and RMI. His mechanism, however, uses a centralized *sequencer node* for serializing write operations on object replicas, which makes it vulnerable to crashes.

We present FTRepMI, an efficient and robust, decentralized protocol for RepMI in which processes progress in successive rounds. To increase efficiency, local writes at different processes are combined in one round. Inspired by virtual synchrony [18], FTRepMI provides object replication [7], while offering flexible membership rules governing the process group (called *world*) that is dedicated to an object. A process interested in an object can obtain a replica by joining its world; when it is no longer interested in the object, it can leave the world. Each member of a world can perform read/write operations on the replica. In case of a write, the replicated object needs to be updated on all processes in its world. A failure detector is used to detect crashed processes, and an iterative mechanism to achieve agreement when a crash occurs. In case of such a process failure, the other processes continue after a phase in which processes query each other whether they received a write operation from the crashed process.

FTRepMI provides sequential consistency for the replicated object, meaning that all involved processes execute the same write operations, in the same order. We slightly digress from Lamport's definition [9] in that we provide the means

to ensure sequential consistency (by executing all write operations on all processes in the same order), but we do not enforce it ourselves for read operations on the replicated object (the programmer still has to use Java's *synchronized* methods for both read and write). We sketch a correctness proof, and moreover we analyzed FTRepMI by means of model checking, on the Distributed ASCI Supercomputer, DAS-3 (`www.cs.vu.nl/das3/`).

The strength of FTRepMI, compared to other decentralized fault-tolerant protocols for sequential consistency, is its simplicity. This makes it relatively straightforward to implement. Simplicity of protocols, and decentralized control, is of particular importance in the dynamic setting of grid applications, where one has to take into account extra-functional requirements like performance, security, and quality of service. A prototype of FTRepMI has been implemented on top of the Ibis system [16], a Java-based platform for grid computing. Results obtained from testing the prototype on DAS-3 show that FTRepMI behaves efficiently both on a single cluster and on a distributed cluster environment.

Related work Orca [1] introduced conceptually shared, replicated objects, extended in [4] with a primary/backup strategy, and partial replication with sequential consistency. *Isis* [2], on which *GARF* [5] relies for communication management, proposes a framework for replicated objects with various consistency requirements. *Isis* presents a per-request chosen-coordinator/cohorts design approach, providing for fault tolerance and automatic restart. Chain replication [17] is a particular case of a primary/backup technique improved by load-balancing query requests. It is latency-bound as a result of "chaining" latencies between servers, leading to performance problems in multi-cluster grids. As support for fault tolerance it uses a central master. *Eternal* [15] addresses *CORBA* applications, providing fault tolerance through a centralized component. It delegates communication and ordering of operations to *TOTEM* [14], a reliable, totally ordered multicast system based on *extended virtual synchrony* as derived from the virtual synchrony model of *Isis*. General quorum consensus, used in [8], is another replication technique, allowing for network partitions and process failure. RAMBO [10] takes the same approach to address atomic memory operations. It supports multiple objects, each shared by a group of processes; groups may overlap. It is considerably more sophisticated, but induces a cost on crash-free operations of eight times the maximum point-to-point network delay.

2 FTRepMI

In our model, a parallel program consists of multiple processes, each holding some threads. A process is uniquely identified by a rank r, which prescribes a total order on processes. All processes can send messages to each other; we assume non-blocking, reliable and stream-based communication. Processes can crash, but their communication channels are reliable. There is no assumption on how the delivery of messages from a broadcast is scheduled.

During crash-free runs, FTRepMI uses a simple communication pattern to dissipate a write operation in the world. The protocol proceeds in successive

rounds, in which the processes progress in lockstep, based on binary-valued logical clocks. If a process receives a local write while idle (this includes reading the replica), it broadcasts the write; we use function shipping [11]. Then it waits for all processes to reply, either with a write operation or a *nop*, meaning that the process does not have a write operation to perform during this round. If a process receives a remote write while idle, it broadcasts a *nop* and waits for messages from all other processes (except the one which triggered this *nop*). Processes apply write operations in some (*world*-wide) pre-defined order (e.g. ascending, descending), based on the ranks attached to these write operations.

FTRepMI can in principle support multiple replicated objects. To simplify our presentation, however, we assume there is only one shared object. The *world* consists of the processes holding a replica of the object, at a given moment in time. The world projection at each process is a set of ranks. Processes can join or leave the world, so at a given moment, this set of ranks may contain gaps.

2.1 The Protocol - Crash-free Runs

As previously explained, a process interested in accessing a replicated object has to first join the world of that object. If there is no world for that object, the process will start one. (This could be done using an external directory, e.g. a file on stable storage, but the exact details are outside the scope of this paper.) When trying to access the replica, a thread invokes the FTRepMI protocol. If it is a read request, the thread can simply read the local copy. In case of a write access, the thread must grab the local lock to contact the Local Round Manager (LRM) at this process. The LRM is at the heart of FTRepMI: it keeps track of the local and remote writes the process received, is in charge of controlling which local writes pertain to which round, and executes writes on the local replica.

Dealing with world changes Processes are allowed to join or leave the world. This is achieved by two special operations, called *join* and *leave*. In our implementation of FTRepMI, for a smooth integration of these operations into the protocol, they are processed as if they were write operations.

Handling a join request When a new process N, with rank n, wants to join the world, it contacts an external directory (e.g. stable storage) for the contact address of another process O which already hosts a replica. Upon receiving N's request, O constructs a special local operation $join(n)$, which contains information about the joining process. This operation is handled by the protocol as a local write operation. Its special semantics is noticed upon its execution: all processes add n to their R set. The contact process O sends N the initialization information (the current state of the replicated object, the current sequence number, its world projection), and N joins the world in the round in which $join(n)$ is performed. In case O no longer accepts join requests because it is leaving, process N stops and retries to join via another contact process.

If a process gets many *join* requests within a short time, its local writes (or the *join*s) may be delayed for multiple rounds. To avoid this, one can alternatively piggyback *join*s onto other messages. A process would then have to maintain four

queues of *join* requests: local requests which need to be acknowledged, remote requests, and local and remote requests pertaining to the next round.

Handling a leave request When a process O, with rank o, wants to leave the world, it performs a special operation $leave(o)$ that results in the removal of rank o from R on all other processes. It is handled as a local write operation.

Dealing with write operations Local and remote write operations on the replica are handled differently. A local write needs to be communicated to the other processes, while the arrival of a remote write may get a process into action, if this write operation is for the current round and the process is idle (i.e. did not receive remote writes or generate a local write for the current round yet). In the latter case, the process generates as a local write for the current round a special *nop* operation. An alternative would be to generate *nop*s at regular time intervals. However, finding suitable intervals is not easy, and this would lead to unnecessary message flooding during periods in which the entire system is idle.

Handling a local write A thread wanting to perform a write operation *op* on the local replica, must first grab the local write lock. Then it asks the LRM at this process to start a new round. If the process is idle, *op* is placed in the queue $\mathcal{WO}$ at this process, which contains write operations waiting to be executed on the local replica in the current round in the correct order; each write operation is paired with the rank of the process where it originates from. Then the next round is started, and the thread that performs *op* broadcasts *op* to all other processes. If there is an ongoing round, *op* is postponed until the next round, by placing it in the queue $\mathcal{NWO}$, which stores write operations for the next round.

Handling a remote write A round can also be started by the arrival of a remote write operation, which means this process has so far been idle during the corresponding round, while another process generated a write operation for this round and broadcast it to all other processes. When this remote operation arrives, the LRM is invoked. If the process is idle, it starts a new round and broadcasts a *nop* to all other processes. During its current round, a process also buffers operations pertaining to the next round ($\mathcal{NWO}$).

Starting a new round When the current round at a process has been completed, the time stamp is inverted, $\mathcal{NWO}$ is cast to $\mathcal{WO}$, and $\mathcal{NWO}$ is emptied. If the new $\mathcal{WO}$ contains a local write, then the process initiates the next round, in which this write is broadcast to the other processes. If $\mathcal{WO}$ does not contain a local write, but does contain one or more remote writes, then the process also initiates the next round, in which it broadcasts *nop* to the other processes. If $\mathcal{WO}$ is empty, then the process remains idle.

2.2 Fault Tolerance

For the fault-tolerant version of FTRepMI, we require known bounds on communication delay, so that one can implement a perfectly accurate failure detector.

Fault tolerant consistency is provided by ensuring that operations issued by a failing process are executed either by all alive processes or by none. When a process n has gathered information from/about each process in the current round, either by a remote write or by a crash report from the local failure

detector, it checks if recovery is needed. If for some crashed process there is no operation in the $\mathcal{WO}$ of n, then n starts the recovery procedure by broadcasting a SOS-message. To answer such messages, each process preserves the queue $\mathcal{CWO}$, i.e. $\mathcal{WO}$ of the previous round, in case the requester is lagging one round behind. Recovery procedure at process n terminates when either it obtains all missing ops or it has received replies from all asked processes. If more processes crashed while n was in recovery procedure and n is still missing ops, then n continues the recovery procedure; namely, a newly crashed process may have communicated missing ops to some processes but not to n. After the recovery procedure ends, all crashed processes for which n still does not hold a remote write in $\mathcal{WO}$ are deleted from n's world, while crashed processes whose missing operations were recovered are taken to the next round. A process q in crash recovery broadcasts a message $S(q, sn_q)$, to ask the other processes for their list of write operations in q's current round sn_q. A process p that receives this message sends either $\mathcal{WO}$ or $\mathcal{CWO}$ to q; if $sn_p = sn_q$, then p sends $\mathcal{WO}$, and if $sn_p = 1 - sn_q$, then p sends $\mathcal{CWO}$. Note that, since q's crash recovery is performed at the end of q's current round, q cannot be a round further than p, due to the fact that p did not send a write in that round yet. Therefore, in the latter case, sn_q must refer to the round before p's current round.

This recovery mechanism assumes that the time-out for detecting crashed processes ensures that at the time of such a detection no messages of the crashed process remain in the system. This is the case for our current implementation of FTRepMI. If this assumption is not valid, a request from a crash recovery must be answered by a process only when it has received either a write or a crash detection for each process in the corresponding round. That is, a process can always dissipate $\mathcal{CWO}$, while it can dissipate $\mathcal{WO}$ only after collecting information on all processes in its world projection.

As a process n is joining the world, the contact process may crash before sending the join accept to n, but after it sent $join(n)$ to some other processes. Now these processes consider n as part of the world, but n is unaware of this and may try to find another contact process. To avoid such confusion, when n retries to join the world, either it must use a different rank, or it must wait for a sufficient time. Then, the alive processes detect that n is no longer in the world, and in the ensuing recovery it is decided that n did not perform a write.

As a process n is joining the world, the contact process o may crash after sending the join accept to n, but before it has sent a $join(n)$ to the other active processes. Then n could join the world while no process is aware of this. The solution is that o gives permission to n to join the world in the round $1 - sn$ after the round sn in which o broadcast $join(n)$, and only after o received in this round $1 - sn$ a remote write or crash detection for each process that took part in previous round sn (i.e. o becomes certain that all active processes that took part in the previous round have received $join(n)$). o's join accept contains not only o's world projection but also the number of detected crashes in the current round. To ensure that n will not wait indefinitely for round $1 - sn$ to start, o will start it with a nop, if o's $\mathcal{NWO}$ is empty (i.e. no process is in round $1 - sn$).

3 Validation

Model checking analysis We specified the fault-tolerant version of FTRepMI in the process algebraic language μCRL [3]. We performed a model checking analysis for three types of properties, on a network of three processes, with respect to several configurations of threads. We used a distributed version of the μCRL toolset on 32 CPUs of DAS-3. First, we verified that the order in which processes execute their writes complies to the order in which they occur in the programs on the threads. Second, we verified that two processes will never execute different writes in the same round. Third, we verified that FTRepMI is deadlock-free, that is, if one process executes writes, and another process does not crash, then the other process will eventually execute the same writes.

Correctness proof We will now argue the correctness of FTRepMI. We focus on sequential consistency, deadlock-freeness and joins. Note that given two active processes, either they are in the same round, or one process is one round ahead of the other. (That is why two round numbers are enough.)

Sequential consistency Suppose that processes p and q have completed the same round, and have not crashed. We will now argue that they performed the same writes at the end of this round. That is, if p performs a write operation WO, then q also performs WO. The operation WO must be a local write at some process r in this round. If r does not crash, it will communicate WO to q, and we are done. So let's consider what happens if r crashes in this round, before communicating WO to q. Then q will detect, either in the previous or in the current round, that r crashed. In the first case, q is guaranteed to receive (e.g. from p) in a crash recovery at the end of the previous round, r's local write in that round, after which q shifts r's crash to the current round. Since p obtains WO, r has managed to communicated WO to some processes $s_1, \ldots, s_k$ in the current round. If at least one of these processes does not crash, WO is communicated to q in the crash recovery procedure that q performs at the end of the current round, and we are done. So let's consider what happens if $s_1, \ldots, s_k$ all crash without replying to q's SOS-message. Since q detects these crashes, and did not receive a write operation for r in the current round yet, q will start a crash recovery for the current round once again. This iterative crash recovery mechanism guarantees that ultimately q will receive WO.

Deadlock-freeness Suppose that process p does not crash, and is not idle, and that all active processes are in p's current or next round. We will now argue that eventually p will progress to the next round. Each active process is guaranteed to become active in p's current round, either by the local write (possibly nop) at p in this round, or by a write from some other process. If none of the processes active in this round crash, then p is guaranteed to receive a write from each of these processes, and complete this round. So suppose that one or more processes crash before sending a write to p in this round. The failure detector of p will report these crashes, meaning that at the end of the round p starts the crash recovery procedure, and asks all processes it thinks to be active in this round for their $\mathcal{WO}$ (if they are in p's current round) or $\mathcal{CWO}$ (if they are in the next round). Active processes that have not crashed will eventually answer with an

SOSReply-message. And those that crash before sending an *SOSReply* to p will be reported by p's failure detector, possibly leading to an iteration of p's crash recovery procedure. Since only a limited number of processes can join a round, eventually p will complete its crash recovery procedure, and thus the round.

Joins Suppose a process N is allowed by its contact process O to join the world. Then all alive processes participating in the round that N joins are informed of this fact. Namely, in the previous round, O has broadcast $join(n)$ to all processes that participated in that round. In the current round, O only gives permission to N to join the world after having received a remote write or crash detection for all processes that participated in the previous round. These processes can only have progressed to the current round after having received $join(n)$. And they will pass on this information to other processes that join in the current round.

4 Performance Evaluation

We tested the performance of the FTRepMI Ibis-empowered prototype on a testbed of two DAS-3 clusters, both having 2.4 GHz AMD Opteron dual-core nodes interconnected by Myri-10G; the clusters are connected by a StarPlane-based [6] wide-area network, with 1ms round-trip latencies and dedicated 10Gbps lightpaths. As a first test, a process generates 1000 write operations on the replica; the rest of the processes only read the replica. Up to 16 CPUs performance is dependent only on the network delay; for 32 CPUs, bandwidth becomes a bottleneck. Second, we analyze the performance of two processes each generating 1000 write operations on the replica. To validate the advantage of combining in the same execution round write operations issued by different processes, each process computes the time per operation as if there were only 1000. There is no significant performance overhead when more writers are present in the system. Third, to analyze FTRepMI in terms of network delay and bandwidth, we repeat the same tests for an equal distribution of CPUs over two clusters. We also look at how distributing the "writing" processes over the clusters affects the performance. We found that FTRepMI performs efficiently also on a wide-area distributed system. The performance penalty incurred is maximum 10%.

We then analyzed the performance of crash-recovery for the simple scenarios of one process generating 1000 write operations on the replica in worlds of up to 32 processes, equally distributed on two clusters. Half-way through the computation (i.e., after 500 write operations are executed), all processes in one cluster (not containing the writer) crash. The remaining processes spend between 2 to 200 ms before resuming normal operation. Note that recovery time is not performance critical. FTRepMI caters for applications which require more than FTRepMI's recovery time to recompute a lost result (if at all possible).

Future work Scalability can be improved by decreasing the size of exchanged messages. Tests on a wide-area network with higher latency (e.g., using clusters from Grid5000) would add more insight on the performance of FTRepMI. We also plan to develop a version of FTRepMI that does not require the presence of perfectly accurate failure detectors.

Acknowledgments We thank Niels Drost and Rena Bakhshi for their helpful comments, and Stefan Blom for his help with the μCRL model checking exercise.

References

1. H. Bal, F. Kaashoek, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE TSE*, 18(3):190–205, 1992.
2. K. Birman. Replication and fault-tolerance in the Isis system. In *SOSP'85*, pages 79–86. ACM, 1985.
3. S. Blom, W. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J. van de Pol. μCRL: A toolset for analysing algebraic specifications. In *CAV'01*, volume 2102 of *LNCS*, pages 250–254. Springer, 2001.
4. A. Fekete, M. F. Kaashoek, and N. Lynch. Implementing sequentially consistent shared objects using broadcast and point-to-point communication. *J. ACM*, 45(1):35–69, 1998.
5. B. Garbinato, R. Guerraoui, and K. R. Mazouni. Implementation of the GARF replicated objects platform. *Distributed Systems Engineering*, 2(1):14–27, 1995.
6. P. Grosso, L. Xu, J.-Ph. Velders, and C. de Laat. Starplane: A national dynamic photonic network controlled by grid applications. *Emerald Journal on Internet Research*, 17(5):546–553, 2007.
7. R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *Ada-Europe'96*, volume 1088 of *LNCS*, pages 38–57. Springer, 1996.
8. M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM TOCS*, 4(1):32–53, 1986.
9. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE TOC*, 28(9):690–691, 1979.
10. N. A. Lynch and A. A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *DISC'02*, volume 2508 of *LNCS*, pages 173–190. Springer, 2002.
11. J. Maassen. *Method Invocation Based Programming Models for Parallel Programming in Java*. PhD thesis, Vrije Universiteit Amsterdam, 2003.
12. J. Maassen, R. v. Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for parallel programming. *ACM TOPLAS*, 23(6):747–775, 2001.
13. J. Maassen, T. Kielmann, and H. Bal. Parallel application experience with replicated method invocation. *Concurrency and Computation: Practice and Experience*, 13(8-9):681–712, 2001.
14. L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Commun. ACM*, 39(4):54–63, 1996.
15. P. Narasimhan, L. Moser, and P. Melliar-Smith. Strongly consistent replication and recovery of fault-tolerant Corba applications. *Computer System Science and Engineering*, 17(2):103–114, 2002.
16. R. v. Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. Bal. Ibis: A flexible and efficient Java-based grid programming environment. *Concurrency and Computation: Practice and Experience*, 17(7-8):1079–1107, 2005.
17. R. v. Renesse and F. Schneider. Chain replication for supporting high throughput and availability. In *OSDI'04*, pages 91–104. USENIX Association, 2004.
18. A. Schiper. Practical impact of group communication theory. In *Future Directions in Distributed Computing*, volume 2584 of *LNCS*, pages 1–10. Springer, 2003.