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 [12]. 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) [10], which was implemented in Manta [11]. 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 [17], 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 analysed FTRepMI by means of model checking, on the Distributed ASCI Supercomputer, DAS-3 (www.cs.vu.nl/das3/).

The strength of FTRepMI, compared with 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 [15], 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 [16] 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* [14] addresses *CORBA* applications, providing fault tolerance through a centralized component, the *Replication Manager*. It delegates communication and ordering of operations to *TOTEM* [13], 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 [9] takes the same approach to address multi-object spanning multi-process atomic memory operations.

2 FTRepMI

In our model, a parallel program consists of multiple processes, each holding some threads. A process is uniquely identified by its rank r, a numerical identifier which prescribes a total order on processes; ranks are merely ordered identifiers and are not assumed to be consecutive. 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 [10]. Then it waits for all processes to reply, either with a write operation or a *nop*, meaning that 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 first grab the WriteLock of the process on which it resides, and then contact the Local Round Manager (LRM) at this process to determine when to broadcast this local write operation to all processes in the world. 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. We implement the LRM as a separate element in each process, providing a monitor-like functionality.

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 (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. An alternative is to 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 When a thread wants to perform a write operation *op* on the local replica, it must first grab the local WriteLock. 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 pending 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 the process is not idle (i.e. there is an ongoing round), *op* is postponed until the next round, by placing it in the queue $\mathcal{NWO}$, which stores the write operations pertaining to 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 (i.e. `endRound`), 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.

Combining local writes of different processes into one round is a key point of FTRepMI. When the next round becomes the current round, its local write operation (if present) is broadcast as a response to the remote write operation(s) and the remote write operation(s) are interpreted as replies to the local write. In such cases, FTRepMI reaches consensus on operation order in only one phase. Therefore, in case of process crashes, which will be considered in the next section, a recovery procedure is crucial for the consistency of the replicated object.

2.2 Fault Tolerance

For the fault-tolerant version of FTRepMI, we require known bounds on communication delay and on internal processing of messages. Thus, if a process sends a message and does not receive a reply within a certain time, this process has crashed. In other words, we assume a perfectly accurate failure detector.

To provide sequential consistency in the presence of crashes, we must ensure that operations issued by a failing process are executed either by all alive processes or by none. When a process n has collected information from each process in the current round, either in the form of a remote write or by a report from its failure detector that the process has crashed, it inspects whether it needs to start a recovery round. If for at least one crashed process there is no remote write in the $\mathcal{WO}$ of n, then n goes into a recovery procedure by sending a message to all processes considered alive at this point. To answer such requests, each process preserves the queue $\mathcal{CWO}$ of write operations in the previous round, in case the requester is lagging one round behind. The recovery procedure at process n terminates when either it obtains all missing *op*s or it receives replies from all asked processes. If more processes crashed while n was in recovery procedure and n is still missing *op*s, then n starts the recovery procedure again; namely, a newly crashed process may have communicated missing *op*s 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 indeed the case for our implementation of FTRepMI. If this assumption is not valid, more care is required, to ascertain that a request from a crash recovery is only answered by a process 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.

While a process N is joining the world, the contacted process may crash before sending an acceptance notification to process N, but after it sent $join(n)$ to some other processes. Then these processes consider N as part of the world. N may try to find another contact process, which could lead to confusion. The solution is that when N retries to join the world, either it must use a different rank, or it must wait for a sufficiently long time. In the latter case the failure detectors of the processes in the world can detect that N is not part of the world, and in the ensuing crash recovery it is concluded that N did not perform a write.

While a process N is joining the world, the contacted process O may crash after sending an acceptance notification to process 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 round sn. With this permission, O sends not only its world projection but also the number of detected crashes in the current round to N. In order to make sure that N does not have to wait indefinitely before round $1 - sn$ starts, O will start this round, either with a local write from one of its threads, or with a *nop* if such a local write is not present in its $\mathcal{WO}$.

3 Validation and Performance Evaluation

We specified the fault-tolerant version of FTRepMI in the process algebraic language μCRL [3]. We generated the state space belonging to the μCRL specification, for a network of three processes, with respect to several configurations of threads. State space generation was performed with a distributed version of the μCRL toolset on DAS-3, using 32 CPUs. The resulting state space was model checked for a wide range of properties.

To test the performance of FTRepMI, we implemented it on top of Ibis [15]. As a testbed, we used two clusters of DAS-3, both having 2.4 GHz AMD Opteron dual-core nodes interconnected by Myri-10G. The clusters are connected by a wide-area network based on StarPlane [6], with round-trip latencies around 1ms and dedicated 10Gbps lightpaths. As a first test, a process generates 1000 write operations on the replica. In this scenario, we also test for a world of one process; the rest of the processes only read the replica. For 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.

For a performance analysis in terms of network delay and bandwidth, we also look at how the prototype behaves when using both clusters at the same time. We repeat the same tests for an equal distribution of CPUs on 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 also 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. Minimum and maximum times spent until the remaining processes resume their normal operation are between 2 to

200 ms. Please note that recovery time is not performance critical. FTRepMI caters for applications which would require more than FTRepMI's recovery time to recompute a lost result (if at all possible).

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. 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.
10. J. Maassen. *Method Invocation Based Programming Models for Parallel Programming in Java*. PhD thesis, Vrije Universiteit Amsterdam, 2003.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. R. v. Renesse and F. Schneider. Chain replication for supporting high throughput and availability. In *OSDI'04*, pages 91–104. USENIX Association, 2004.
17. A. Schiper. Practical impact of group communication theory. In *Future Directions in Distributed Computing*, volume 2584 of *LNCS*, pages 1–10. Springer, 2003.