Minimal-log total-order broadcast using consensus as a black box

Ana Rita Quelho Ferreira [1], **Luiz Eduardo Buzato**[1]

[1]Universidade Estadual de Campinas - UNICAMP – Campinas, SP – Brazil

{ana.ferreira}@students.ic.unicamp.br,{buzato}@ic.unicamp.br

Abstract. *Access to stable storage is a requisite for uniform total-order broadcast. At critical points of its execution every process involved in the broadcasts has to save its state into stable memory, to guarantee that safe recovery is possible. Unfortunately, access to stable memory is expensive and reduces performance and should, thus, be minimized. In this work, we address this minimization problem by presenting a novel minimal-log total-order broadcast algorithm for the crash-recovery model. Our algorithm, as others in the literature, suppose the existence of a consensus solution, viewed as a black box module.*

Resumo. *Acessos a memória estável são um requisito para difusão com ordenação total uniforme. Em pontos críticos de sua execução, todo processo envolvido nas difusões deve salvar seu estado em memória estável para garantir que, após uma queda, sua recuperação seja possível. Infelizmente, acessos a memória estável são caros e reduzem desempenho e devem, portanto, ser minimizados. Neste trabalho, tratamos deste problema de minimização apresentando um novo protocolo minimal no uso do log para o modelo de falhas de queda-recuperação. Nosso algoritmo, como outros na literatura, supõe a existência de uma solução de consenso, visto como um módulo caixa preta.*

1. Introduction

The construction of available web services encompasses various areas of research, among them an important research area investigates how to best replicate selected sub-systems of a web service. Replication of the critical sub-systems can protect the web service against partial failures, so that if one replica of the critical sub-system fails, the others continue available to provide the service to its clients. In general, in a distributed system, the replication of sub-systems is supported by a replication middleware that is able to deliver messages to all active replicas in the exact same order using a total-order broadcast protocol. One of the common characteristics of any uniform total-order broadcast protocol designed for the crash-recovery failure model is the requirement to write critical information into stable storage. Unfortunately, accessing stable storage is an expensive operation and can become a bottleneck to the practical use of such protocols in total-order toolkits.

Figure 1 shows the context of a total-order broadcast over consensus protocol which supports replicated sub-systems. In consensus protocol, each process proposes an intial value and they all decide for the same value among the proposals. The transport protocol provides means for the processes to exchange messages. We refer to uniform

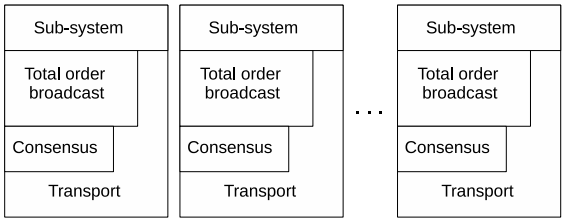

Figure 1. Total order broadcast context

total-order broadcast protocols which use uniform consensus as a black box and are designed for asynchronous crash-recovery systems as TOB/C. In this paper we only consider uniform solutions: those which restrict the behavior of faulty and non-faulty processes as defined in [Défago et al. 2004]. We say a process is in a regular execution while it normally executes the protocol. If a process crashes and, after a period of time, restarts, we say it is recovering during the period of time it is restabilishing to reassume regular executions.

Previous work [Rodrigues and Raynal 2003] presented a solution that is minimal in the number of log accesses and in the complexity of the information in regular executions of the protocol among TOB/Cs. However, the recovery includes a large number of log accesses. Moreover, the log grows indefinitelly. Alternative protocol [Rodrigues and Raynal 2003] and [Mena and Schiper 2005] present faster recoveries at the expense of increasing the number of log accesses in regular executions and raising the complexity of the information. Thus, they are not minimal-log in regular executions. Moreover, both are not minimal in the number of log accesses during recovery.

We have achieved our goal of minimizing the number of log accesses during recovery and also the number of log entries also maintaining the minimalities of the original minimal-log solution [Rodrigues and Raynal 2003].

The solution we propose in this paper is theoretical since it requires a specific restriction usually associated to small nets. As well as [Rodrigues and Raynal 2003] and [Mena and Schiper 2005] we assume each pair of processes are directly connected by communication channels. However, our solution is an important step towards the design of a real web service since the size of its log is limited. In practice it means that it is possible to estimate the maximum size of space in stable memory required to hold the log based on system characteristics.

This rest of the paper is organized as follows. Section 2 specifies uniform total-order broadcast for the crash-recovery model. Section 3 details our minimal-log uniform total-order broadcast solution. Section 4 compares our solution to related work. Section 5 concludes the paper.

2. Specification of uniform total-order broadcast in the crash-recovery model

2.1. Model of Computation

We use the same Model of Computation as [Rodrigues and Raynal 2003] does. The system consists of a finite set of processes $\Pi = \{p, ..., q\}$. The system is static which means

the set of processes does not change after system start-up time. In the crash-recovery model, processes may crash and later recover. Before a process starts the first time it is *down*. When a process starts or recovers from a crash and executes the start/recovery procedure it goes from *down* to *up*. When a process is *up* and crashes it goes from *up* to *down*. If, after some time, an *up* process does not crash forever we say it is *eventually always up*. A process is *bad* if: a)it eventually crashes forever or b) it crashes and recovers infinitely often never concluding the protocol communication. A process is *good* if it is *eventually always up*. A process communicates to another only by message exchange through communication channels. We assume there is a bidirecional channel between each pair of processes, not necessarily FIFO and they can lose or duplicate messages. Message transfer delays are arbitrary and finit. We assume that if a process p send a message m an infinit number of times to a good process q then q receives m an infinit number of times. A process is equipped with a volatile memory and a stable memory, both local. The primitive **log** saves data into the stable storage while the primitive **restore** rescues data from the stable storage. When a process crashes it loses its volatile contents and also the messages which arrive while it is *down*.

2.2. Definitions

We assume there is a Transport module which transmits the messages sent by the processes. We assume that each message is uniquely identified in the system. Thus, duplicated messages can be easly eliminated. We assume the Transport block offers the primitives: **send** which sends the message to its destination address, **multisend** which sends the message to all the other processes and **receive** which receives a message.

We also assume there is a multiple-instance consensus block which we refer to as MI-consensus where each consensus instance is an independent consensus, which is uniquely identified in the system by its instance-id. MI-consensus instances are assumed to be sequentially decided.

As other solutions [Rodrigues and Raynal 2003] and [Mena and Schiper 2005], we assume MI-consensus maintains in stable memory all the consensus instance results. Thus, a process may obtain these results again when recovering from a crash even if all the other processes have also crashed.

It is important to emphatize that if k_p is the smallest not decided instance in the system then for any instance j_p less than k_p, no matter the value the process proposes for j_p it is going to be ignored by MI-consensus since it is already decided.

We assume MI-consensus offers the following primitives, which can be indistinctly invoked in regular executions or upon recovery: **propose**(k_p, v) which allows the process to propose v as its initial value for the instance k_p of consensus and **decided**(k_p, $decision$) that allows the process to get the decision of the instance k_p in $decision$.

We define TOB/C by the following primitives which can be indistinctly invoked in regular executions or upon recovery: **TO-broadcast** (*m*), which allows the process to submit a message *m* to uniform total-order broadcast, waiting until *m* is ordered by the protocol and **TO-deliver** (*i-th*), which allows the process to obtain the *i-th* total-ordered message if it is already ordered, waiting until it is ordered first, otherwise. When recovering from a crash the process may request for earlier delivered messages which it had not properlly handled before crashing.

We say a process TO-broadcasts *m* if it invokes **TO-broadcast**(*m*) for the message *m* and we say a process TO-delivers *m* if it invokes **TO-deliver**(*i-th*) and *m* was delivered as the *i-th* total-ordered message.

We define TOB/C through the following properties, similarly to [Rodrigues and Raynal 2003]. *Uniform Validity*: If a process TO-delivers a message *m* then some process has previously TO-broadcasted *m*. This property concerns about not delivering invalid messages. *Uniform Integrity*: If a process TO-delivers a message *m* then all the good processes eventually deliver *m*. Integrity property is related to the quality of being whole or united. It concerns about ensuring that there are no delivered messages which are not eventually delivered by all the good processes. *Uniform Agreement*: Let p and q processes (identical or not). If p TO-delivers m_i as the i-th total-ordered message and q TO-delivers m_j as the i-th total-ordered message: (a) if i=j then m_i=m_j and (b) if $i \neq j$ then $m_i \neq m_j$. The agreement properties are related to harmony of different opinions. This property ensures both: a process does not order a message in diferent ordinal positions and two different processes order the same message in the same ordinal position. *Termination*: if a good process concludes execution of TO-broadcast(m) primitive then all good processes eventually deliver m.

3. Uniform total-order broadcast

Figure 2 shows the novel TOB/C which we detail next. The state of a process p is composed of: K_p, the current consensus instance at p that contains the smallest consensus instance-id which decision value process p does not know; $Proposal_p$, the set of messages (a) proposed to the last invoked instance of consensus or (b) that is about to be proposed to the current consensus instance K_p, dependig on the moment, and it is initialized with $\perp$ which means that no value has been proposed to any instance of consensus at all; $Unordered_p$, a set of still unordered messages at process p; $Agreed_p$, a queue of all the already ordered messages known by process p; $Gossip$-k_p, the highest current consensus instance-id known by p, which is obtained through the gossiping mechanism where each process disseminates its current instance-id to the others.

To submit a message m to total-order broadcast, a process invokes **TO-broadcast**(m) primitive, which includes m in $Unordered_p$, except if $Agreed_p$ contains m, indicating that m has been already ordered. Note that **TO-broadcast** primitive conclusion implies in ordering the message. This signifies that the underlying MI-consensus has already saved m in its own log as part of one of its decisions.

Also note that if a process crashes before concluding **TO-broadcast** execution, the message may not be delivered. Specially if it crashes before saving it into stable storage and also before sending the message to other processes. Although, it is not a very big problem if, after recovering from a crash, the process TO-broadcasts the last message it has TO-broadcasted before crashing, since messages are uniquely identified in the system. Thus, the client has to maintain the message in it own log until concluding **TO-broadcast** primitive execution.

Every process at task **gossip** sends GOSSIP-M(m) to the other processes to disseminate each message contained on its $Unordered_p$ and sends GOSSIP-R(K_p) to disseminate its current instance of consensus. When a process receives a GOSSIP-M(m) message it includes m in its $Unordered_p$ set only if $Agreed_p$ does not contain m. When

Initial values:

K_p, $gossip - k_p$ = 0; $Unordered_p = \emptyset$; $Proposal_p = \perp$; $\forall\ i\ Agreed_p[i] = \perp$;

upon initialization or recovery ():

restore ($Last - proposal_p$); $Proposal_p \leftarrow Last - proposal_p$

if ($Proposal_p = \perp$) **or** ($Proposal_p = \emptyset$) **then fork task** { $gossip$ }

else

$Unordered_p \leftarrow Proposal_p$; **fork task** { $gossip$ }

while $Proposal_p = Last - proposal_p$ **do**

propose(K_p, $Proposal_p$); **wait until** decided(K_p,$decision$)

newmsgs $\leftarrow decision \setminus Agreed_p$; $Agreed_p \leftarrow Agreed_p$ :: *newmsgs*;

$Unordered_p \leftarrow Unordered_p \setminus Agreed_p$

$Proposal_p \leftarrow Proposal_p \setminus Agreed_p$; $K_p \leftarrow K_p$ + 1

fork task {sequencer}

Task $gossip$:

repeat forever

multisend GOSSIP-R(K_p); $\forall\ m \in Unordered_p$ multisend GOSSIP-M(m)

upon receive GOSSIP-M(m) **from** q:

$Unordered_p \leftarrow (\ Unordered_p \cup \{\ m\ \}\ ) \setminus Agreed_p$;

upon receive GOSSIP-R(k_q) **from** q:

if ($k_q > k_p$) **then** $gossip - k_p \leftarrow$ max ($gossip - k_p$, k_q)

Task $sequencer$:

repeat forever:

wait until ($Unordered_p \neq \emptyset$) **or** ($gossip - k_p > k_p$)

$Proposal_p \leftarrow Unordered_p$;

log ($Proposal_p$); propose(k_p, $Proposal_p$); **wait until** decided (k_p, *decision*)

$newmsgs \leftarrow decision \setminus Agreed_p$; $Agreed_p \leftarrow Agreed_p$ :: *newmsgs*

$Unordered_p \leftarrow Unordered_p \setminus Agreed_p$; $K_p \leftarrow K_p$ + 1;

upon TO-broadcast(m):

$Unordered_p \leftarrow (\ Unordered_p \cup m\ ) \setminus Agreed_p$; **wait until** $m \in Agreed_p$

upon TO-deliver(i):

wait until $Agreed_p$ [i] $\neq \perp$; **return** $Agreed_p$ [i]

Figure 2. Minimal-log total order broadcast over consensus algorithm

a process receives a GOSSIP-R(K_q) message it atualizes its $gossip$-K_p variable with the greatest value among K_p and $gossip$-K_p only if the sender q has a higher current instance number K_q than itself K_p. If $gossip$-k_p is greather than K_p it indicates that K_p has already been decided and, thus, p is left behind. See details in task **sequencer** description. The task **sequencer** invokes consecutive instances of consensus. It only invokes the current instance of consensus in one of the two cases: (a) if there are unordered messages in $Unordered_p$ or (b) if $gossip$-K_p is greather than K_p, indicating that the instance K_p of consensus has been already decided and, thus, p was left behind by another process. This mechanism avoids invoking unnecessary instances of consensus. In the case (b) the process can propose an empty set as the initial value because the initial value will be discarded since the decision is already made.

Before invoking the current consensus instance of consensus, a process copies $Unordered_p$ to $Proposal_p$ and saves $Proposal_p$ into stable memory. After proposing, it waits until the current instance of consensus K_p is decided, increases K_p and appends the ordered messages queue result at the end of the $Agreed_p$ queue except the duplicated messages. It also atualizes $Unordered_p$ by excluding the $Agreed_p$ messages from it. Thus, the new ordered messages are transferred from $Unordered_p$ to $Agreed_p$. This log access operation is minimal during regular executions and is also minimal in the complexity of the information logged because the only information the process logs in regular execution is the last proposal which is strictly necessary for the underlying consensus that restores it from the stable memory during recovery in order to obtain the last proposed value.

Upon initialization or recovery $Last$-$proposal_p$ is restored from the stable memory and copied to $Proposal_p$. If $Last$-$proposal_p$ is $\perp$ it indicates that the process has never proposed a value to any consensus instance before and, thus, the recovery is finished right after starting tasks **gossip** and **sequencer**. If $Last$-$proposal_p$ is an empty set it means that process p had invoked consensus instance for the last time before its crash not because of its unordered messages but because p was left behind and, so, the recovery is finished right after starting tasks **gossip** and **sequencer**. In fact, the task sequencer will continue invoking the next instances of consensus until the process reaches the other processes.

We have included the correction in [Rodrigues and Raynal 2003] pointed by [Mena and Schiper 2005] about starting task **gossip** before (instead of after) invoking the previous instances during initialization or recovery. If $Last$-$proposal_p$ is not equal to $\perp$ neither is an empty set, recovery consists in consecutively invoke consensus for each previous instance occurred before the process crashes proposing the same initial value equal to $Last$-$proposal_p$, waiting for the consensus decision and properly atualizing $Agreed_p$, $Unordered_p$, $Proposal_p$ and K_p, until one of the messages of $Proposal_p$ is ordered. When it happends is because p has reached the consensus instance that was current at the moment of the process crash and the recovery is finished right after starting task **sequencer**.

Although we do not know to which instance of consensus the $Last$-$proposal_p$ was associated we can afirm that if we propose the $Last$-$proposal_p$ as initial value consecutively to all the previous instances, we will be right at one of them. Moreover, if any of the consensus instance results includes one of the messages of the $Last$-$proposal_p$ then recovery is ended since in regular executions the process excludes from $Unordered_p$

the messages that were ordered in previous instances of MI-consensus. Moreover, no messages received from other processes are included in $Unordered_p$ if they are already ordered in p. As $Proposal_p$ is a copy of $Unordered_p$, during recovery, no re-invokation of previous decided instances of consensus will result a message containned in $Last\text{-}proposal_p$.

During all the invocations of previous instances of consensus, no matter the value proposed during recovery, it will be ignored by MI-consensus if the decision of the instance is already made. The process never overwrites a previous proposed value unless the correponding instance of consensus is decided. Thus, it does not affetc MI-consensus previous results.

4. Comparing our solution to related work

Our minimal-log protocol and minimal-log protocol [Rodrigues and Raynal 2003] have equivalent performance in regular executions. They present the same number of messages exchanges m, protocol steps s and log operations l to perform a given operation in regular executions of the protocol since we have made modifications in specific points that do not affect these numbers. Although, during recovery, while [Rodrigues and Raynal 2003] makes l_r log accesses, where l_r is the number of consensus instances invoked by the process before it crashes, our solution makes only one. This represents an important improvement in the recovery performance.

Figure 3 compares the use of stable memory by the log in [Rodrigues and Raynal 2003] and in our solution.

In regular executions of the protocol, a process logs its set of unordered messages before proposing them to consensus as its initial value such as minimal log access solution [Rodrigues and Raynal 2003] does. The number of log accesses, the complexity of the information and the instructions perfomed are essentially the same and both protocols have equivalent performance. During recovery, [Rodrigues and Raynal 2003] reads

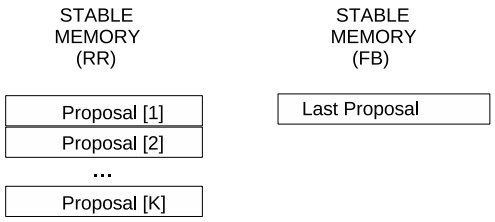

Figure 3. Comparing stable memory usages

from the log each of the values previously proposed for every multiple-instance consensus used before it crashes. In contrast, our algorithm only reads the value proposed for the last invoked instance and hence all the previous proposals may be discarded from the log.

Finally, while [Rodrigues and Raynal 2003] cumulatively maintains all the previous proposed values to consensus, making the log grow indefinitly, our solution only maintains the last proposal. In practice, this means that in [Rodrigues and Raynal 2003] the space required to hold the log is infinit while in our solution it is limited and can be estimated considering some systems characteristics such as the sizes of the messages, specific behaviors and worst cases.

5. Conclusion

We have presented a solution which is minimal-log among uniform total-order broadcast algorithms for asynchronous crash-recovery systems that use multi-instance uniform consensus as a black box, in the following aspects: (a) in the number of log accesses a process makes in regular executions, (b) in the complexity of the information, (c) in the number of log accesses a process makes during recovery and (d) in the number of log entries maintained in the stable storage. The first three aspects are related to performance whilst the second and the last one are associated to the space required for the log in the stable storage device. We have reduced to only one the number of accesses during recovery as well as the number of entries maintained in the log by changing the recover strategy.

The main advantage of our contribution, besides the minimalities, is that the stable memory space required to hold the log is limited and can be previously estimated.

References

Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421.

Mena, S. and Schiper, A. (2005). A new look at atomic broadcast in the asynchronous crash-recovery model. In *SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, pages 202–214, Washington, DC, USA. IEEE Computer Society.

Rodrigues, L. and Raynal, M. (2003). Atomic broadcast in asynchronous crash-recovery distributed systems and its use in quorum-based replication. *IEEE Trans. on Knowl. and Data Eng.*, 15(5):1206–1217.