Using Atomic Actions in Replica Groups to Simplify the Replication Implementation

Leon Bouwmeester
Intelligent Networked Computing Laboratory
HP Laboratories Bristol
HPL-93-96
November, 1993

In a distributed systems infrastructure, replication can be supported by the use of groups. This requires complex protocols to implement certain ordering and delivery guarantees. Using atomic actions in the infrastructure allows the use of simpler protocols since failures of these protocols to deliver these guarantees can now be managed within the atomic actions. Side effects of using atomic actions are that a group has its own responsibility of maintaining its collection of group members and that a group member can be multi-threaded.
1 Introduction

In distributed computing systems, reliability can be achieved by using replication, atomic actions, or a combination of both. Atomic actions allow failures to be handled by rolling back to a previous checkpoint, which involves all modifications made to the object’s state between the checkpoint and the failure occurrence being undone.

Replication can be supported by the use of groups. A replica group allows a number of replicas to be used transparently as a single entity [5]. If some of the replicas fail, the remaining replicas can complete the computation. A group is said to be $K$-resilient if it can survive up to $K$ failures.

Replica groups require complex protocols implementing ordering and delivery guarantees to ensure state consistency. These protocols have an impact on the efficiency of the replica group. In this paper, we demonstrate that efficiency could be improved if the execution of these protocols is allowed to fail occasionally. In our protocol, we assume that failures are rare. If, however, failures occur then another mechanism is needed to manage these failures. An atomic action supports such an optimistic approach: failures are assumed to be rare, but if they occur, we just let them happen and try to recover afterwards. To guarantee computational progress, we must assume that a replica group is invoked by only a few other replica groups.

In this paper, we will demonstrate that it is possible to use simple protocols in the replication infrastructure while state consistency is ensured by relying on the serialisation property of atomic actions.

The rest of this paper is organised as follows: in section 2 we motivate the use of atomic actions in replication infrastructures. In section 3, we describe the failure model on which our protocol will be based. The functionality of the protocol is described in section 4, followed by the conclusions and suggestions for future work in section 6.

2 Motivation for the use of atomic actions in replication

A replicated computation can be implemented in two different ways:

Active replication In active replication, all replicas process all input messages concurrently so that their internal states are closely synchronised and, in the absence of faults, output messages can be obtained from any replica [6].

Passive replication In passive replication, only one of the replicas, the primary replica, processes the input messages and provides the output messages. The internal states
of the other replicas are regularly updated by means of checkpoints from the primary replica [6].

2.1 Support in active replication

In active replication, to implement $K$-resilience each invocation request or invocation result must be received and ordered by at least $K + 1$ replicas before any replica is allowed to communicate with any other group [10]. If this guarantee is not met, this could lead to state inconsistencies.

To improve performance, the computation is allowed to continue as soon as an invocation result has been received and ordered by $N$ group replicas, with $0 < N < K + 1$. The idea is to use atomic actions to manage any failures that the execution of this mechanism gives rise to.

To maintain a consistent state, it must be ensured that $N$ comprises at least the majority of the group members. Obtaining the majority also allows us to remove the responsibility for ordering from the invoked group as ordering is now imposed by relying on the serialisability property of atomic actions. Consider the following figure where one group, the client group, request a service from another group, the server group:

![Figure 1: Processing requests without ordering](image)

Suppose that client 1 and client 2 are sending write requests $w_1$ and $w_2$ respectively (whether or not the clients are also groups is not of any importance in this example) and suppose that no failures will occur. Each server member processes incoming requests in FIFO-order. In the example shown, server group member\(^1\) $A$ is processing the requests in the order $< w_1, w_2 >$ and both server group members $B$ and $C$ are processing the requests in the order $< w_2, w_1 >$.

\(^1\)Sometimes we speak of server member rather than server group member. These descriptions have the same meaning.
To schedule interfering requests, a request must first obtain the appropriate lock. Write locks are exclusive whereas read locks can be shared.

The request of client 1 is only processed at server member A because the request happens to be blocked at the other two server members, because the locks are not obtained. Client 1 is not able to obtain the majority of server members since only server member A replies; the other server members reply with a lock-conflict because the request of client 2 has obtained the locks. Client 1 has to abort and sends an abort message to the server group.

Conversely, client 2 is able to obtain the majority of server members and is allowed to commit the action. Eventually, server members that were replying with lock-conflicts will also process the request of client 2. The server group has reached a consistent state.

Using this mechanism for achieving order also deals with group member failures and network partitions: as long as the majority of server members can be obtained, the action is allowed to commit. Otherwise, the action must be aborted (see also section 3).

One of the side effects of obtaining the majority of server members is that the server group should have at least $2K + 1$ server members to implement $K$-resilience [2]. Furthermore, it is important to assume that only a few client groups invoke the server group. This is because the presence of a large number of clients makes it unlikely that a client will be able to obtain the majority of server members. Finally, if ordering is achieved at the client by relying on the serialisation property of atomic actions, it requires a responsibility on that client to be aware of server member failures and, if the protocol requires so, to maintain the collection of server members in the server group. In the protocol to be described in section 4, a mechanism is presented to solve this problem.

2.2 Support in passive replication

In passive replication, only the primary replica processes the input messages and it provides the output messages. The states of the other replicas are regularly updated by means of checkpoints of the state of the primary replica. If the primary replica fails, one of the other replicas will become the new primary replica and it resumes execution from the most recent checkpoint. Passive replication uses a mechanism similar to atomic actions: inserting checkpoints and rolling back to the most previous checkpoint in the presence of a failure. Therefore, there is no need to integrate an atomic actions infrastructure into a passive replication scheme.

3 Failure model

The protocol described in section 4 tolerates the following failures that are assumed to be rare [2]:

3
Crash failures A group member suffering a crash failure behaves correctly until it crashes after which it is forever silent.

Omission failures Group members suffering omission failures occasionally omit to respond to an invocation. The limiting case is when an object fails to respond to all invocations in which case it is suffering a crash failure.

Timing failures A timing failure occurs when a group member either omits to respond to an invocation or responds too early or too late.

Management of failures can be specified by two policies. These policies give the application programmer more flexibility in controlling the group execution protocol:

Replication policy This policy specifies whether or not a new group member should be included after a crash failure has occurred.

Retry policy In case of timing failures, this policy specifies how many times a invocation request should be retransmitted before giving up and assuming a crash failure.

The protocol in section 4 is based on obtaining the majority of server members, therefore it can cope with network partitions: only the partition with the majority of server members of the original server group will continue to operate. During the network partition (it is assumed that only one network partition occurs at a time), the partitioned group is divided into two new subgroups. After repairing the partition, the largest subgroup of the two will be chosen to be the new server group. This mechanism is chosen to deliver a deterministic and consistent manner of determining a new group after repairing a network partition. It also has the advantage that the new groups (the client group can also be partitioned) are still able to communicate with each other.

4 Group execution protocol

The group execution protocol (GEX) is based on the lightweight group protocol described in [10]. The structure of the GEX protocol is as follows:
Figure 2: Group execution protocol structure

The group members form a logical ring and a distinguished group member, the sequencer, has the responsibility of multicasting the invocation and determining the order in which the invocations should be processed. To avoid the sequencer being a single point of failure, the responsibility is shared among all group members. Essentially, it is the protocol described by Chang and Maxemchuk [3].

Each group member is separated into an application member for processing invocation results and a GEX-member for handling the inter-group and intra-group communication.

The GEX protocol is extended with atomic actions to simplify the replication implementation.

4.1 Basic functionality

When the client group wants to invoke a request on the server group, each client application member first starts an atomic action. Then the request is passed on to the client GEX-member and it qualifies the request with a distinguishing mark. The mark is a combination of the client group name, the incarnation number and a client specific request identifier. The request needs to be marked to distinguish it from other invocations from other client groups. The client group name is used to distinguish requests from other client groups, the incarnation number has a double function. If the client group has changed, the server group is able to detect this as the incarnation number is increased. The incarnation number can also be used to detect and to configure out slow running group members. Finally, the client specific request identifier is used to distinguish requests, invoked by the same client group, from each other requests.

After qualifying the request with the mark, all client GEX-members simply queue this request, except for the client sequencer (i.e. the client GEX-member with enabled status).
The client sequencer multicasts the invocation to the GEX-members of the server group, while the other client GEX-members simply wait for a reply to be returned.

Each server member processes incoming requests in FIFO-order. At each server member, the request is passed on to the server application member where a nested atomic action is started. The next step is to obtain the appropriate lock to schedule interfering actions. If the lock is granted, the invocation will be processed. Otherwise the request is blocked and a lock-conflict message is generated.

The server application member passes the result on to the server GEX member that returns the result to the server sequencer rather than the client sequencer. Otherwise this would introduce the problem that the server group depends on the availability of the client group. The server sequencer collects all the results sent by the other server members and it determines which request has been processed at the majority of server group members.

An interesting issue arises when concurrent read operations are processed at the server members. In that case, the server sequencer has multiple majority sets: for each client's request there exists a majority set. This can be handled by enabling the server sequencer to associate activities in the multi-threaded server members (a special mechanism is required to name the different activities)

For the client’s request that was processed at the majority of server members, the server sequencer returns the result. Server members that have returned a lock-conflict for that client’s request will receive an abort message from the server sequencer. The respective client sequencers are not notified of this event as only this single request at those particular server members is aborted. It is possible that server sequencer is able to obtain the majority for that client’s request later on.

For instance, considering the situation as depicted in figure 2 it means that only request w₁ at server member A is aborted. The server sequencer is still able to obtain the majority for request w₁ after commitment of request w₂.

The server sequencer then tries to collect any remaining results. For reasons of performance, returning the result to the client sequencer and collecting remaining results can be done concurrently. If none of the incoming requests have been processed at the majority of server members, the server sequencer aborts all requests at all server members. Only in this case does it notify the respective client sequencers that their requests have been aborted. After returning the result to the client sequencer(s), the sequencer responsibility is passed on to the next server member.

If the client sequencer has received a normal termination result, it is allowed to commit the action. If the client sequencer has received an abort message, it can handle the abort any way it chooses. The client sequencer uses an atomic multicast to ensure that all the other client members receive the termination result and to ensure that the server group is
informed of its decision to commit or abort. Finally, the client sequencer passes sequencer responsibility to its successor. If a sequencer are not processing, it will eventually time out and pass responsibility on to its successor.

Finally, it is easy to extend our model for multi-threaded client groups. If the client sequencer multicasts the result to the other client members, a sequence notification can be added to impose ordering at the client members. If the client group is multi-threaded, it must be ensured that activities in the client group are scheduled deterministically.

The protocol still works if the server group switches role to client and invokes another server group [2].

4.2 Failure handling

Message failures, or timing failures, at the client group are detected when the server group fails to respond within a certain time frame. The client sequencer multicasts the request periodically until a result has been returned. A maximum number of multicasts is introduced in case the server group has failed. The use of the atomic multicast ensures that all client members receive the termination result, even in the presence of client sequencer failure.

Besides multicasting on behalf of the client’s group, the client sequencer also has a second function: it detects client group member failures. If the client sequencer tries to pass control to its successor and fails, it will fail to receive an acknowledgement. Client sequencer failures are detected in a similar way: if the other client members time out, they poll the client sequencer, after which it is detected that the client sequencer has failed. Client member failures result in the creation of a new client group and the election of a new client sequencer. The client sequencer multicasts the original request to the server group so that the server sequencer knows it must return results to the new client sequencer.

As stated, each server member processes requests in FIFO-order and, therefore it will not detect any message failures. It will only detect duplicated requests that will be discarded. An exception for detecting message failures is the server sequencer. The server sequencer is able to detect message failures because some server members fail to respond: they return neither a termination result nor a lock-conflict. The message failures managed by the server sequencer are request failures (the message from the client has been lost) and reply failures (the reply sent by the server member has been lost). Both are detected and managed in a similar way.

Suppose that the server sequencer has been able to determine which client’s request has been processed at the majority of server group members and suppose that some message failures have occurred. Depending on the policy, the server sequencer aborts those server members and retransmits the request on the client’s behalf and, if necessary, it does so several times. It collects the remaining results and excludes server members that are still failing or responding
with different termination results. If the server sequencer itself has missed the request being processed at the majority of the server group, it will abort the request which it is currently processing and it asks the client sequencer to multicast the request again. Now, the server sequencer is also able to collect replies from server members not in the majority set.

The server sequencer persists in returning the termination result until the client sequencer has sent a commit or an abort message. There is an upper limit of retries in case the client group has failed.

Server member failures are detected by the server sequencer as they fail to respond (even after possible retries). If the server member fails after sending its reply to the server sequencer, this failure will be detected when the next request is processed or, if it is next in line to be sequencer, when server sequencer responsibility is passed on. If the server sequencer fails before it can return the reply, other server members will detect this failure because they time out, similar to the way that client sequencer failures are detected.

5 Related work

The ARJUNA programming system [8, 7] uses replication to increase the availability of persistent objects. Ordering in the server group is also imposed by relying on the serialisation property of atomic actions [4]. The main differences with our protocol can be summarised as follows:

1. In ARJUNA the client is responsible for maintenance of the server group. During an invocation, the client assembles an exclude list which is a list of those server replicas that have been detected to have failed. This list is used to update the group view. In our protocol, both the server group and the client group themselves are responsible for maintaining their own collection of group members.

2. If the client is a group with \( N \) members, then all these client group members will invoke the server group and are able to obtain the locks concurrently. In our protocol, only one client member performs the actual invocation.

3. In our protocol clients can be multi-threaded. When the client sequencer commits (aborts) the action, a sequence notification can be added to establish an ordering at the client. This sequence notification is ignored by the server group, because at the server group ordering is imposed by relying on the serialisation property of atomic actions. A multi-threaded client group requires a deterministic scheduling mechanism.

6 Conclusions and future work

In this paper, we have demonstrated that it is possible to use an atomic actions infrastructure in active replica groups. The use of atomic actions has simplified the protocols used for
implementing ordering and delivery guarantees since these requirements are now imposed by relying on the serialisation property of atomic actions.

We have been able to integrate the atomic actions into groups in such a way that a group itself is responsible for maintaining the group membership. A side effect is that group members can be multi-threaded. Multi-threading, however, requires deterministic scheduling in the client. It also requires a mechanism to distinguish the activities executed by the threads. These problems need further investigation. Considering multi-threaded client groups, we also need to investigate whether there should be one GEX-member per client member or one GEX-member per thread.

If only a few client groups are invoking the server group and failures are rare, then high performance can be expected as the server group members start processing requests without communicating with each other. Afterwards, the result is returned to the client group. However, more evidence is needed to sustain this. Another side effect is a minimal use of the network: the protocol only requires one multicast to invoke the server group; if $K$ is the resilience of the group, $2K$ unicast are required to determine ordering and one unicast is required to return the result to the client group; finally, one atomic multicast is required to commit or abort the action. An atomic multicast can be implemented as two plain multicasts [9].

7 Acknowledgements

I would like to thank all the referees providing me with comments on the draft versions of this paper. Special thanks to Robert Cole, Michael Olsen and Nigel Edwards.

8 References


