An Implementation of the Archipelago algorithms for Leaderless Consensus

IN4391 - Distributed Systems - Group 12

Rohan Madhwal (5568412)
Miruna Betianu (5632927)
Aditya Shankar (5454360)
Theodoros Veneti (5527805)

***Abstract*—Consensus algorithms involve multiple processes that are required to reach a decision (or consensus) on a certain value. In this paper, we describe our Scala implementation of the Archipelago algorithms [1] which consider indulgent consensus algorithms. These algorithms tolerate an adversary that can delay processes from participating in the algorithm for an arbitrary long period of time. Traditionally, such algorithms rely on a leader, this is a weakness because a slow leader acts as a bottleneck in the algorithm. The paper presents a series of leaderless consensus algorithms, that tolerate three different scenarios: shared memory model with omissions, message passing with omissions, and Byzantine failures. We cover the first two scenarios.**
Our code is available at
https://github.com/adis98/DS/tree/Aditya.

I. Introduction

The problem of consensus is usually summarised as that of agreeing on a piece of data among multiple processes. It is fundamental to most Distributed Systems because without all processes being on the same page about what to do, cooperation between them is impossible. However, in systems with unreliable communication and faulty nodes, it is not a trivial problem to solve. Hence, for reliability of operation in these systems, intelligent consensus protocols are required to ensure that the whole system can continue to function in a collaborative manner. It is important to note that for the purpose of these algorithms, it is not as important what the processes agree on, but instead that they all come to the same conclusion.

Traditionally, popular consensus algorithms such as Paxos and Raft rely on the concept of a "leader" which helps all processes converge towards a decision. However, there are certain issues involved with the approach, the most obvious of which is that it's possible for the leader to go down or be unreachable. In this scenario, algorithms usually attempt to elect a new leader after a certain timeout, this creates a significant slowdown in the process if this occurs frequently. Further, an adversary can easily target the leader to bring it down and hence create arbitrary long periods of delays in the system.

Archipelago is a series of three novel algorithms presented in the Leaderless Consensus paper which generate consensus in a distributed system of nodes without electing a leader node. A "leaderless algorithm" is presented intuitively as one that should be resistant to repeated slow downs of an individual nodes. This is reasonable since in an algorithm that relies on a leader, repeated slow downs of the leader node would lead to non-functionality inside the system.

The three algorithms presented in the paper are:

1) Archipelago: The first algorithm relies on shared memory and a new variant of the the classical adopt-commit object [2] that returns maximum values to help different processes converge to the same output
2) OFT-Archipelago: The second algorithm is a generalisation of the first in a message passing system with omission failures
3) BFT-Archipelago: The third algorithm also tolerates byzantine failures

Our major contributions with this project are:

1) Adaptation of the Archipelago and OFT-Archipelagos to use the actor model
2) Implementation of this adaptation inside Scala using Akka
3) Experiments on both of the algorithms which help us compare them and evaluate their performance

While we only implemented the first two algorithms due to the limited scope of the course project, implementing the third algorithm should simply be a matter of extending the second algorithm.

Archipelago and its variants work in a *synchronous-k* setting. This definition is inspired by the notion of synchrony in a message passing model where there is a bound on the time needed for a message to propagate from one process to another and for the receiver to process this message. In a message passing model, we can divide time into rounds such that, in each round, every process p: (i) sends a message to every other process in the system, and (ii) delivers any message that was sent to p and performs local computation.

To adapt synchrony to the shared memory model, we also assume that processes take steps in rounds. Specifically, in each round, every process P_i : (i) performs a write in some

	1	2	3	4	5	6	7	8	9	10	11
p_1	$step(\mathcal{R}_5)_{p_1}$	X	$step(\mathcal{R}_2)_{p_1}$	$step(\mathcal{R}_6)_{p_1}$	$step(\mathcal{R}_3)_{p_1}$	X	$step(\mathcal{R}_3)_{p_1}$	$step(\mathcal{R}_2)_{p_1}$	$step(\mathcal{R}_1)_{p_1}$	$step(\mathcal{R}_4)_{p_1}$	$step(\mathcal{R}_1)_{p_1}$
p_2	$step(\mathcal{R}_2)_{p_2}$	$step(\mathcal{R}_4)_{p_2}$	X	X	X	$step(\mathcal{R}_2)_{p_2}$	$step(\mathcal{R}_1)_{p_2}$	X	X	X	X

Fig. 1: Interaction in a synchronous-1 setup with 2 processes. X indicates the disabled process

R_j[i] and (ii) collects all the values written in array R_j . In one round, different processes can read from different arrays. Figure 1 shows a synchronous-1 execution with two processes.

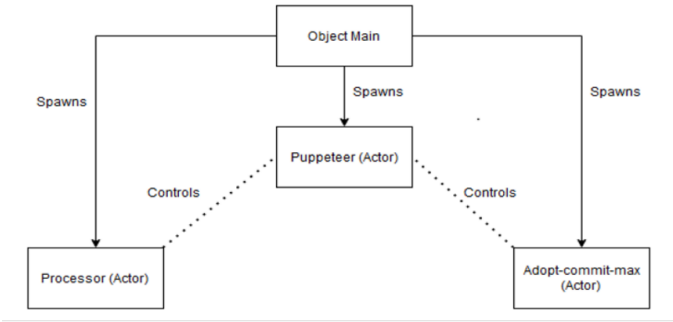

Fig. 2: High-level design for Archipelago

II. Archipelago

We will now explain our implementation of Archipelago for the shared memory model by first giving a high-level view of the system architecture, and then present the algorithm mentioned in the paper.

A. Architecture

Figure 2 shows the design for the algorithm. We have a Main object that creates the Processors, the Puppeteer, and the Adopt-Commit-Max object. All three of them are Actors because of Archipelago's need to implement single-writer access. Actors sequentially process messages, making the actor model a convenient design choice. Actors also have easy-to-use mechanisms for message passing.

Implementing synchronization between rounds is done with the Puppeteer. It sends "enable" or "disable" messages to all the nodes every round to indicate whether a process is enabled or not. At the end of every round, a process sends an acknowledge message back to the Puppeteer. The Puppeteer aggregates all the replies before starting a new round to ensure that the algorithm proceeds in an organized manner. The Puppeteer also decides on the adversary selection strategy, i.e., which nodes to disable every round.

Nodes/Processors run the Archipelago algorithm. Each one can propose values, write to or read from shared storage, and communicate with the adopt-commit-max objects. The adopt-commit-max objects accept proposals from different nodes and return a pair of values (D,P). D can be either an "adopt" or "commit" message and P is the corresponding value. The object satisfies the properties of CA-Validity, CA-Agreement, CA-Commitment, and CA-Termination. See figure 3

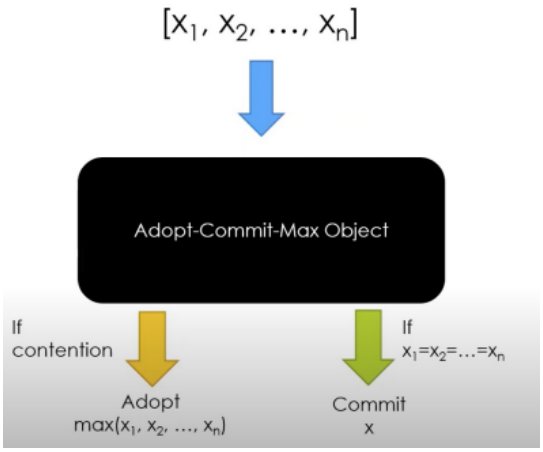

Fig. 3: Adopt-commit-max object functioning

B. Algorithm

The adopt-commit-max algorithm is shown in figure 4a. Figure 4b depicts Archipelago where all processes share an infinite sequence of adopt-commit-max objects (C) to ensure safety and a max register m (lines 17 to 20) to help with convergence. A max register r is a wait-free register that provides a write operation, as well as a readmax operation that retrieves back the largest value that was previously written to r. Its write can be implemented by letting each process write to a single-writer multi-reader register whereas its readmax can be implemented by collecting all values written by all processes and taking the maximum. In a synchronous minus 1 execution, the processes converge towards one value and there is an adopt-commit-max object where all processes propose this exact single value. Then, due to CA-commitment property of the adopt-commit-max object, the adopt-commit-max outputs (commit, value) and Archipelago decides in finite time.

Algorithm The adopt-commit-max algorithm

Shared state:
  A and B, two arrays of n single-writer multi-reader registers, all initially $\perp$

procedure propose(v): ▷ taken by a process p_i
  $A[i] \leftarrow v$ ▷ step A starts
  $S_A \leftarrow$ collect(A) ▷ step A ends
  if $(S_A \setminus \{\perp\} = \{v'\})$ **then** ▷ step B starts
    $B[i] \leftarrow \langle$commit, $v'\rangle$
  else $B[i] \leftarrow \langle$adopt, $\max(S_A)\rangle$ ▷ or step B starts
  $S_B \leftarrow$ collect(B) ▷ step B ends
  if $S_B \setminus \{\perp\} = \{\langle$commit, $v'\rangle\}$ **then**
    return $\langle$commit, $v'\rangle$
  else if $\langle$commit, $v'\rangle \in S_B$ **then return** $\langle$adopt, $v'\rangle$
  else return $\langle$adopt, $max(S_B)\rangle$

(a) Adopt-Commit-Max algorithm

Algorithm Archipelago leaderless consensus

Shared state:
  $C[0, \ldots, +\infty]$, an infinite array of adopt-commit-max objects in their initial state
  m, a max register object that initially contains $\langle 0, \perp\rangle$.
  Note that $\langle x, y\rangle > \langle x', y'\rangle$ if $x > x'$ or $(x = x'$ and $y > y')$

Local state:
  c ▷ index of adopt-commit-max object, initially 0

procedure propose(v):
  while *true* **do**
    m.write($\langle c, v\rangle$) ▷ step R starts
    $\langle c', v'\rangle \leftarrow m$.readmax() ▷ step R ends
    $\langle control, v''\rangle \leftarrow C[c']$.propose($v'$)
    $c \leftarrow c' + 1$
    if $control =$ adopt **then** $v \leftarrow v''$
    else return v''

(b) Archipelago for shared memory

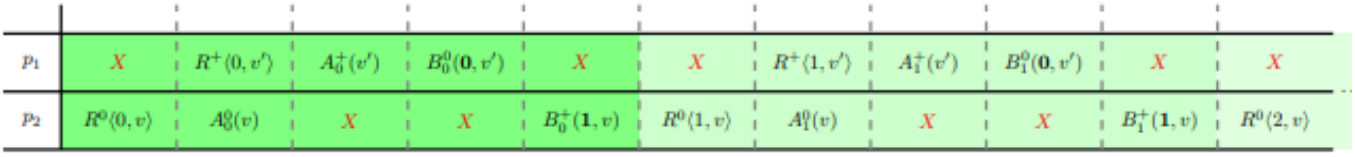

Fig. 5: Adversary selection strategy for synchronous-1 with two nodes

III. Archipelago in message passing: Oft Archipelago

Oft-Archipelago has been implemented in a similar fashion to the shared memory version. The Puppeteer's function does not change too much, since it is only used to synchronize rounds. The change however is that there is no shared memory between processes, so the nodes now also send messages among themselves. The algorithm also requires more than 50% of the nodes to be active in a round for proper functioning. Oft-Archipelago's proof hinges on each step taking exactly one round. This is why, Algorithm 8 combines the write and collect in a single round: the broadcasts in lines 14, 20 and 26 act as both the write and read invocations whereas the responses in lines 36, 39 and 42 confirm the write, and return all values written so far. Although this combination of writes and reads can break atomicity, it does not violate safety during asynchronous periods.

IV. Adversary strategy

So far, we have not focused on the process for selecting disabled nodes. However, the theoretical results in the paper indicate that it is possible to devise an adversary that is able to prevent archipelago from ever converging if the number of disabled processors exceeds a threshold (greater than 50%). One such adversary strategy is shown in figure 5. This execution has a pattern that repeats every 5 rounds (light-green boxes). Processes p1 and p2 propose values v' and v respectively, with v' greater than v. In the first round, process p1 is suspended, so process p2 performs an R step, writes (0,v), and retrieves (0,v) from m. Then, in the second round both processes p1 and p2 take steps. Process p1 writes (0,v') and retrieves (0,v') since (0,v') is greater than (0,v). In the same round, p2 writes v to C[0].A[2]. Then, in the third round, when process p1 takes an A step it writes value v' in C[0].A[1] and when p1 collects the values written in array A (line 6), p1 sees that there are two different values (v and v') in C[0].A. Therefore, in the fourth round, when process p1 performs a B step, it retrieves back (adopt, v'). Process p2 takes a B step in the fifth round after being suspended in the third and fourth rounds, p2 writes (commit, v) in C[0].B[2], and then during the collect of B, p2 sees that (adopt, v') is written in C[0].B[1] and p2 returns (commit, v) (line 13). Afterwards, starting from the sixth round the processes behave in the exact same way. This adversary strategy has also been implemented for the shared memory version of archipelago (can be enabled by setting "adversary_mode" variable to true).

V. Experiments & Results

After completing the implementation of the two algorithms, we proceeded to evaluate their performance and behaviour. To achieve this, we did multiple runs (100 samples) for each selection of number of processes and percentage of them being blocked (randomly selected nodes) (abiding to what the initial authors define as acceptable values). The goal here is to get an overview about the number of rounds before converging to a single value happens (since convergence itself is guaranteed), so we chose to report mean round and variance numbers. The results are presented below:

The results paint an interesting picture about the two algorithms. In the shared-memory (table I and figure 7) algorithm,

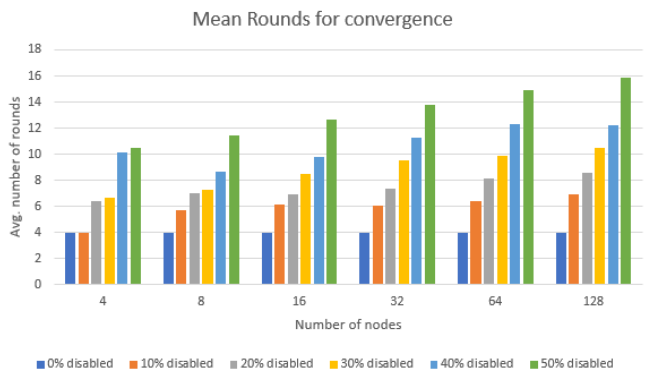

Fig. 6: Rounds for convergence in Archipelago with message passing

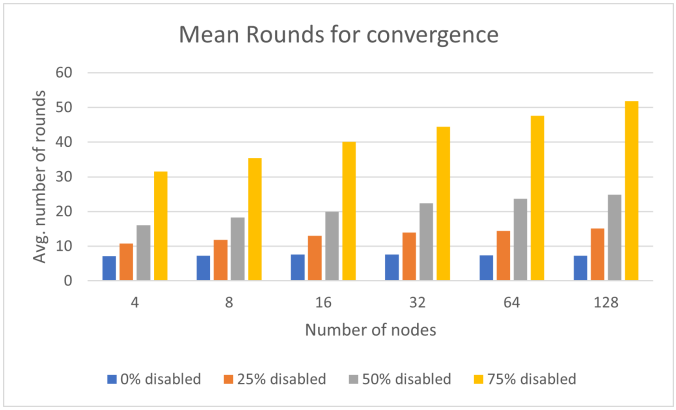

Fig. 7: Rounds for convergence in Archipelago (shared memory)

	Disabled Nodes			
	0%	25%	50%	75%
Nodes	Mean *(Variance)*			
4	7.09 *(0.26)*	10.83 *(3.80)*	16.05 *(14.95)*	31.51 *(80.25)*
8	7.27 *(1.11)*	11.8 *(3.80)*	18.33 *(13.43)*	35.38 *(71.31)*
16	7.57 *(1.76)*	13.03 *(4.25)*	19.92 *(9.10)*	41.12 *(60.00)*
32	7.63 *(2.23)*	13.90 *(5.24)*	22.35 *(8.27)*	44.44 *(42.87)*
64	7.36 *(1.14)*	14.45 *(3.91)*	23.69 *(10.84)*	47.61 *(50.19)*
128	7.21 *(0.59)*	15.1 *(4.07)*	24.91 *(10.77)*	51.78 *(67.62)*

TABLE I: Results for experiments on shared memory algorithm.

we can see that the number of rounds are only affected by the percentage of nodes being blocked, rather than the nodes running the algorithm itself. Both variance and mean metrics seem to follow this pattern.

However, in the case of the message-passing (table II and figure 6) algorithm, the behaviour is different. Here, both the number of nodes running the algorithm and the percentage of them being blocked, affect the average number of rounds before convergence is reached. Moreover, variance seems to generally be quite low for a smaller percentage of nodes being blocked, but increases rapidly while getting closer to the limit of blocked nodes for which convergence is still guaranteed ($f = \frac{n-1}{2}$).

Algorithm Archipelago in message passing

Local State:
  i, the current adopt-commit-max object, initially 0
  R, a set of tuples, initially empty
  $A[0, 1, \dots]$, a sequence of sets, all initially empty
  $B[0, 1, \dots]$, a sequence of sets, all initially empty

procedure propose(v):
  while true **do**
    $\langle i, v' \rangle \leftarrow$ R-Step(v)
    $\langle flag, v'' \rangle \leftarrow$ A-Step(v')
    $\langle control, val \rangle \leftarrow$ B-Step($flag, v''$)
    if $control =$ commit **then return** val
    else $i \leftarrow i + 1$

procedure R-Step(v):
  broadcast(R, i, v)
  wait until receive (R-response, i, R) from $f + 1$ proc.
  $R \leftarrow R \cup \{$ union of all Rs received in previous line$\}$
  $\langle i', v' \rangle \leftarrow \max(R)$
  return $\langle i', v' \rangle$

procedure A-Step(v):
  broadcast(A, i, v)
  wait until receive (A-response, $i, A[i]$) from $f + 1$ proc.
  $\mathcal{S} \leftarrow$ union of all $A[i]$s received
  if $\mathcal{S}$ contains only one value val **then return** $\langle$true, $val\rangle$
  else return $\langle$false, $\max(\mathcal{S})\rangle$

procedure B-Step($flag, v$):
  broadcast($B, i, flag, v$)
  wait until receive (B-response, $i, B[i]$) from $f + 1$ proc.
  $\mathcal{S} \leftarrow$ union of all $B[i]$s received
  if $\mathcal{S}$ contains only $\langle$true, $val\rangle$ for some val **then**
    return $\langle$commit, $val\rangle$
  else if $\mathcal{S}$ contains some entry $\langle$true, $val\rangle$ **then**
    return $\langle$adopt, $val\rangle$
  else return $\langle$adopt, $max(\mathcal{S})\rangle$

upon reception of (R, j, v) **from** p:
  Add $\langle j, v\rangle$ to R
  send(R-response, j, R) to p

upon reception of (A, j, v) **from** p:
  Add v to $A[j]$
  send(A-response, $j, A[j]$) to p

upon reception of $(B, j, flag, v)$ **from** p:
  Add $\langle flag, v\rangle$ to $B[j]$
  send(B-response, $j, B[j]$) to p

Fig. 8: Archipelago for message passing

VI. Challenges

The paper that we implemented primarily presents theoretical results, hence it was up to us to come up with an implementation from scratch. The main challenge we faced was in designing the puppeteer for round-synchronization. The other challenge was that the descriptions given in the paper were sometimes open to interpretation. Coming up with a suitable experiment for evaluation was also up to us because the original paper does not present any empirical results. Hence there was no baseline for us to compare our results with. We felt that the number of rounds to achieve consensus

	Disabled Nodes					
	0%	10%	20%	30%	40%	50%
Nodes	Mean *(Variance)*					
4	4.00 *(0.00)*	4.00 *(0.00)*	6.4 *(1.24)*	6.68 *(1.06)*	10.10 *(6.43)*	10.51 *(6.80)*
8	4.00 *(0.00)*	5.74 *(0.61)*	7.00 *(0.98)*	7.24 *(1.16)*	8.67 *(2.24)*	11.40 *(5.64)*
16	4.00 *(0.00)*	6.16 *(0.70)*	6.92 *(0.92)*	8.47 *(1.53)*	9.78 *(2.17)*	12.62 *(4.43)*
32	4.00 *(0.00)*	6.10 *(0.67)*	7.39 *(0.90)*	9.54 *(1.69)*	11.29 *(3.17)*	13.77 *(5.14)*
64	4.00 *(0.00)*	6.43 *(0.54)*	8.15 *(1.06)*	9.92 *(1.70)*	12.27 *(2.61)*	14.90 *(6.35)*
128	4.00 *(0.00)*	6.94 *(0.59)*	8.54 *(0.77)*	10.49 *(1.82)*	12.25 *(1.84)*	15.85 *(3.96)*

TABLE II: Results for experiments on message-passing algorithm.

is a reasonable metric because it gives a measure of how long the algorithm takes to complete under different scenarios.

REFERENCES

[1] K. Antoniadis, A. Desjardins, V. Gramoli, R. Guerraoui, and I. Zablotchi, "Leaderless consensus," 07 2021.

[2] E. Gafni, "Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract)." *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pp. 143–152, 01 1998.