Leaderless Consensus

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

March 30, 2022

Introduction

- Objective is to simulate a leaderless consensus algorithm: Archipelago and its variants

Introduction

- Objective is to simulate a leaderless consensus algorithm: Archipelago and its variants
- Divided into rounds, processors/nodes propose values per-round

Introduction

- Objective is to simulate a leaderless consensus algorithm: Archipelago and its variants
- Divided into rounds, processors/nodes propose values per-round
- Some nodes are disabled in a round

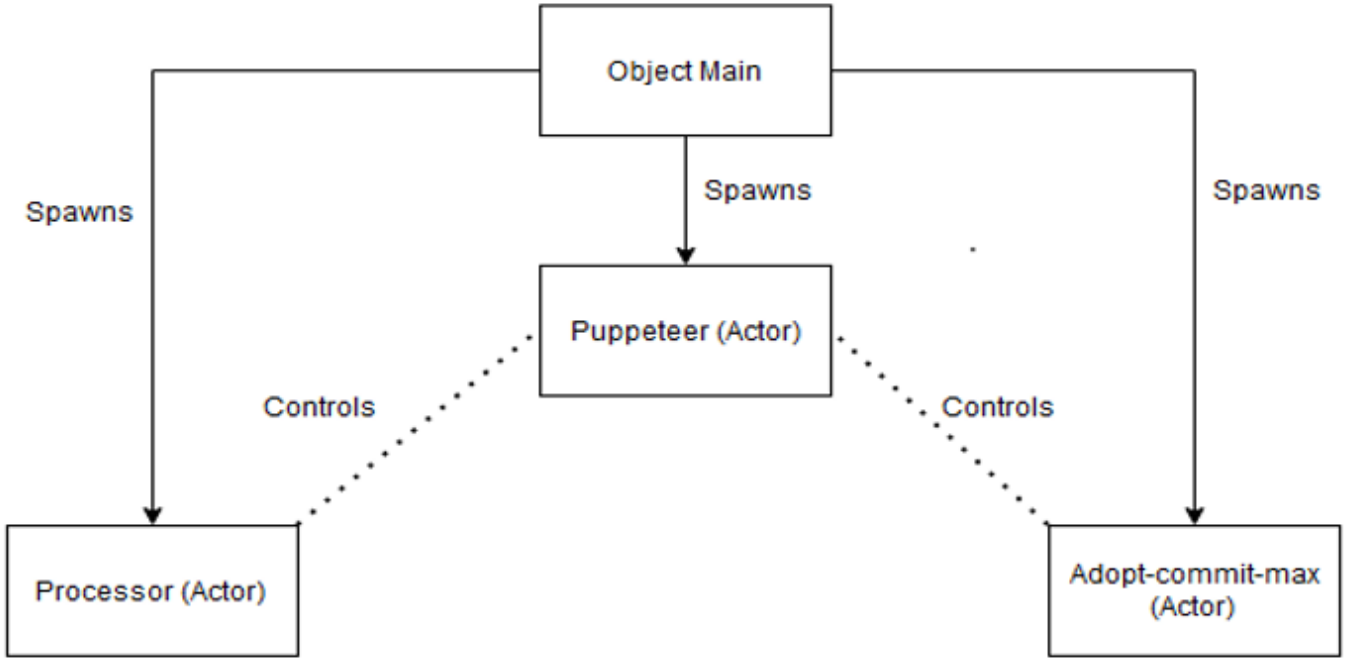
Introduction

- Objective is to simulate a leaderless consensus algorithm: Archipelago and its variants
- Divided into rounds, processors/nodes propose values per-round
- Some nodes are disabled in a round
- We've also implemented a tougher adversary selection strategy for 2 processors

	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

Figure: Interaction in a synchronous-1 situation

High level view of components



Adopt-commit-max algorithm

- Every process p proposes a value

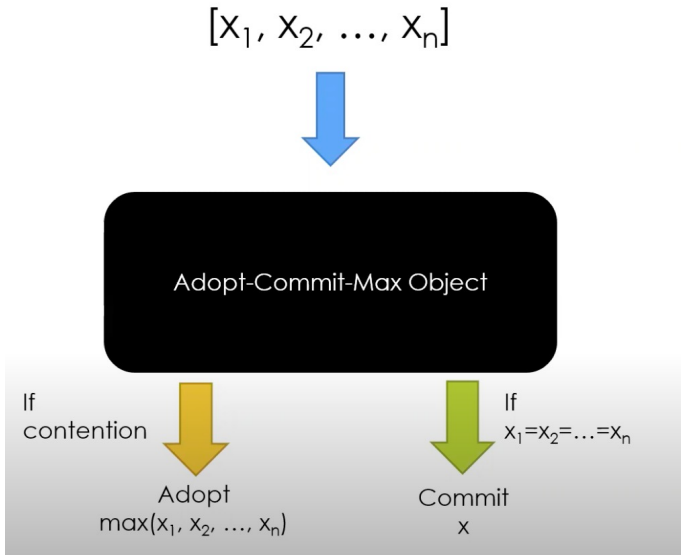


Figure: The manner in which the adopt-commit-max object behaves

Adopt-commit-max algorithm

- Every process p proposes a value
- The output is a pair $<d, p>$, where d is either *commit* or *adapt*

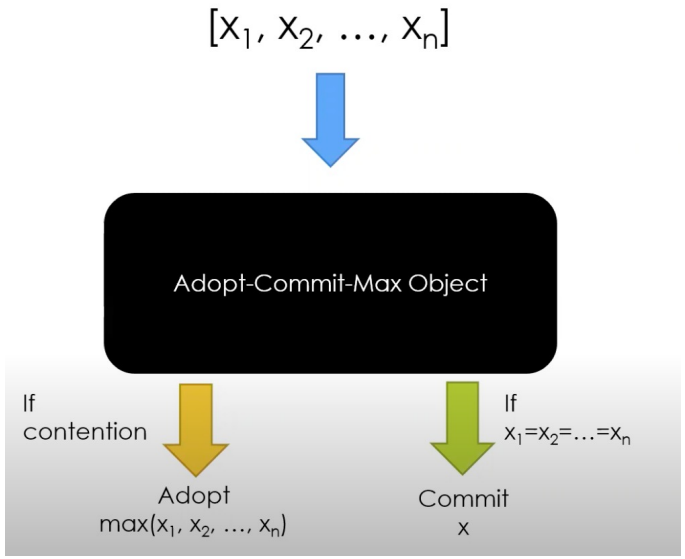


Figure: The manner in which the adopt-commit-max object behaves

Adopt-commit-max algorithm

- Every process p proposes a value
- The output is a pair $<d, p>$, where d is either *commit* or *adapt*
- Satisfies the following properties
  - CA-Validity
  - CA-Agreement
  - CA-Commitment
  - CA-Termination

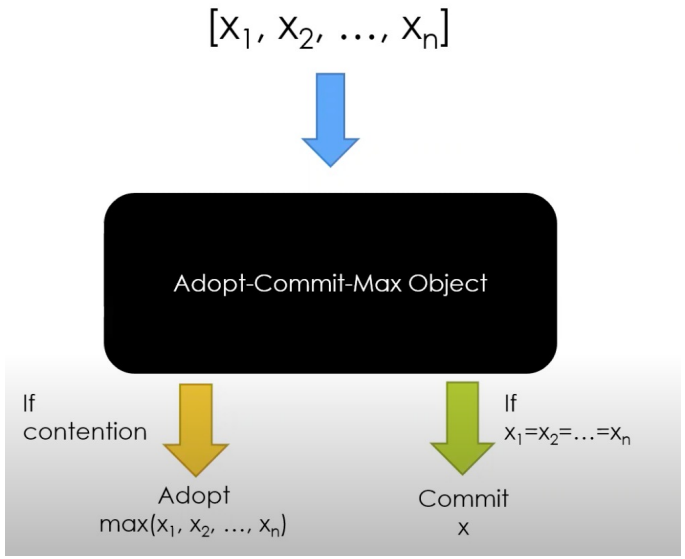


Figure: The manner in which the adopt-commit-max object behaves

Demo

Initial algorithm

Algorithm 4 Archipelago in message passing

```
1: Local State:
2:    i, the current adopt-commit-max object, initially 0
3:    R, a set of tuples, initially empty
4:    A[0, 1, ...], a sequence of sets, all initially empty
5:    B[0, 1, ...], a sequence of sets, all initially empty

6: procedure propose(v):
7:    while true do
8:       ⟨i, v'⟩ ← R-Step(v)
9:       ⟨flag, v''⟩ ← A-Step(v')
10:      ⟨control, val⟩ ← B-Step(flag, v'')
11:      if control = commit then return val
12:      else i ← i + 1

13: procedure R-Step(v):
14:   broadcast(R, i, v)
15:   wait until receive (R-response, i, R) from f + 1 proc.
16:   R ← R ∪ { union of all Rs received in previous line}
17:   ⟨i', v'⟩ ← max(R)
18:   return ⟨i', v'⟩

19: procedure A-Step(v):
20:   broadcast(A, i, v)
21:   wait until receive (A-response, i, A[i]) from f + 1 proc.
22:   S ← union of all A[i]s received
23:   if S contains only one value val then return ⟨true, val⟩
24:   else return ⟨false, max(S)⟩

25: procedure B-Step(flag, v):
26:   broadcast(B, i, flag, v)
27:   wait until receive (B-response, i, B[i]) from f + 1 proc.
28:   S ← union of all B[i]s received
29:   if S contains only ⟨true, val⟩ for some val then
30:      return ⟨commit, val⟩
31:   else if S contains some entry ⟨true, val⟩ then
32:      return ⟨adopt, val⟩
33:   else return ⟨adopt, max(S)⟩

34: upon reception of (R, j, v) from p:
35:   Add ⟨j, v⟩ to R
36:   send(R-response, j, R) to p

37: upon reception of (A, j, v) from p:
38:   Add v to A[j]
39:   send(A-response, j, A[j]) to p

40: upon reception of (B, j, flag, v) from p:
41:   Add ⟨flag, v⟩ to B[j]
42:   send(B-response, j, B[j]) to p
```

- Good but not well suited for message passing

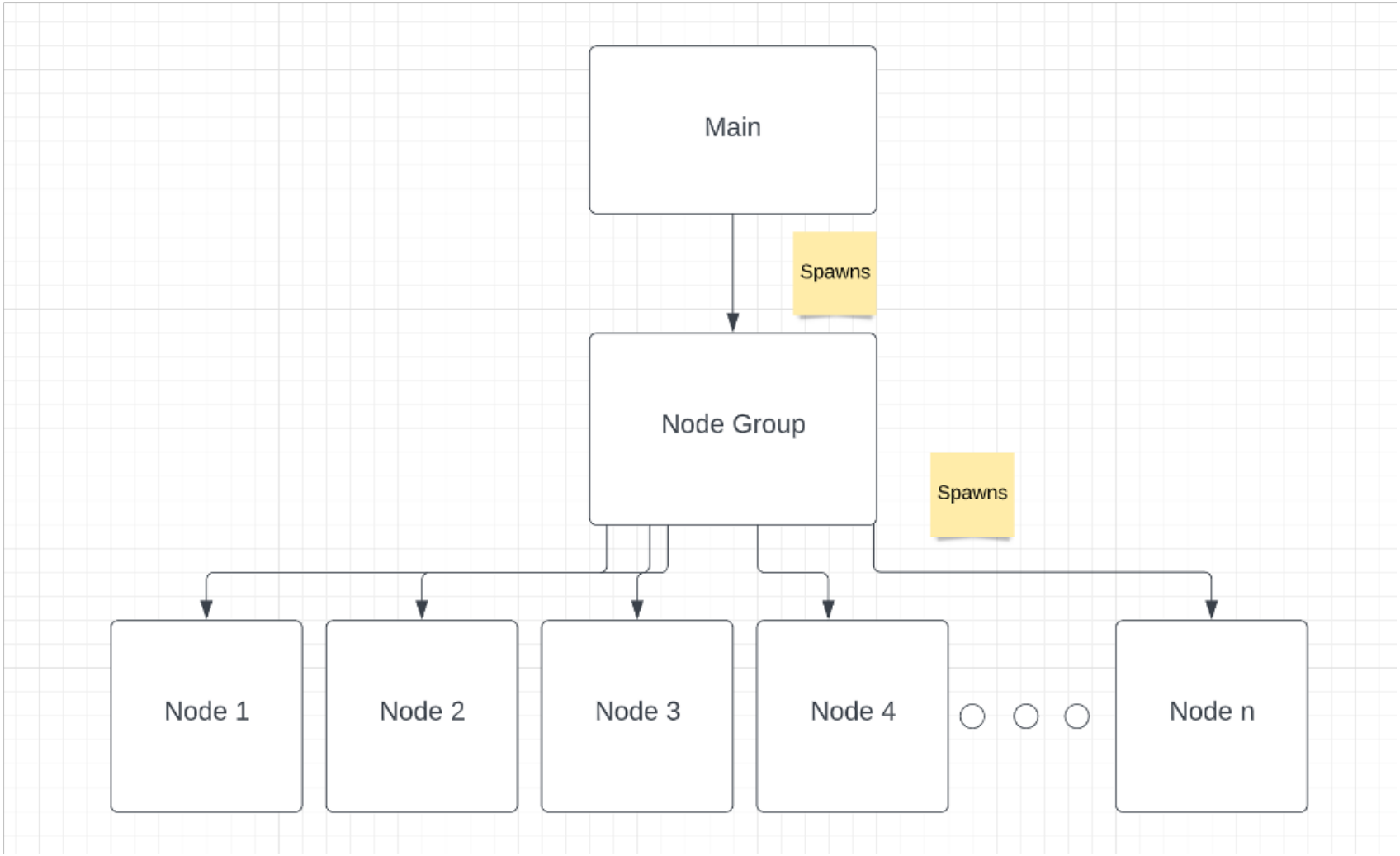
Initial algorithm

Algorithm 4 Archipelago in message passing

```
1: Local State:
2:    i, the current adopt-commit-max object, initially 0
3:    R, a set of tuples, initially empty
4:    A[0, 1, ...], a sequence of sets, all initially empty
5:    B[0, 1, ...], a sequence of sets, all initially empty

6: procedure propose(v):
7:    while true do
8:       ⟨i, v'⟩ ← R-Step(v)
9:       ⟨flag, v''⟩ ← A-Step(v')
10:      ⟨control, val⟩ ← B-Step(flag, v'')
11:      if control = commit then return val
12:      else i ← i + 1

13: procedure R-Step(v):
14:   broadcast(R, i, v)
15:   wait until receive (R-response, i, R) from f + 1 proc.
16:   R ← R ∪ { union of all Rs received in previous line}
17:   ⟨i', v'⟩ ← max(R)
18:   return ⟨i', v'⟩

19: procedure A-Step(v):
20:   broadcast(A, i, v)
21:   wait until receive (A-response, i, A[i]) from f + 1 proc.
22:   S ← union of all A[i]s received
23:   if S contains only one value val then return ⟨true, val⟩
24:   else return ⟨false, max(S)⟩

25: procedure B-Step(flag, v):
26:   broadcast(B, i, flag, v)
27:   wait until receive (B-response, i, B[i]) from f + 1 proc.
28:   S ← union of all B[i]s received
29:   if S contains only ⟨true, val⟩ for some val then
30:      return ⟨commit, val⟩
31:   else if S contains some entry ⟨true, val⟩ then
32:      return ⟨adopt, val⟩
33:   else return ⟨adopt, max(S)⟩

34: upon reception of (R, j, v) from p:
35:   Add ⟨j, v⟩ to R
36:   send(R-response, j, R) to p

37: upon reception of (A, j, v) from p:
38:   Add v to A[j]
39:   send(A-response, j, A[j]) to p

40: upon reception of (B, j, flag, v) from p:
41:   Add ⟨flag, v⟩ to B[j]
42:   send(B-response, j, B[j]) to p
```

- Good but not well suited for message passing
- Waiting for responses seemed antithetical to the reactive, async style of doing thins

Archipelago with message passing



Main Actor

- Manages the Node Group

Main Actor

- Manages the Node Group
- Asks it to spawn nodes and start Archipelago

Node Group

```
sealed trait Command

final case class RequestTrackDevice(nodeId: String, replyTo: ActorRef[Main.Command])
final case class Start() extends Command
final case class Commit(value: Int, nodeId: String) extends Command
final case class BroadcastR(rBroadcast: Node.RBroadcast) extends Command
final case class BroadcastA(aBroadcast: Node.ABroadcast) extends Command
final case class BroadcastB(bBroadcast: Node.BBroadcast) extends Command
```

- Keeps track of all nodes

Node Group

```
sealed trait Command

final case class RequestTrackDevice(nodeId: String, replyTo: ActorRef[Main.Command])
final case class Start() extends Command
final case class Commit(value: Int, nodeId: String) extends Command
final case class BroadcastR(rBroadcast: Node.RBroadcast) extends Command
final case class BroadcastA(aBroadcast: Node.ABroadcast) extends Command
final case class BroadcastB(bBroadcast: Node.BBroadcast) extends Command
```

- Keeps track of all nodes
- Asks each node to start Archipelago inside them

Node Group

```
sealed trait Command

final case class RequestTrackDevice(nodeId: String, replyTo: ActorRef[Main.Command])
final case class Start() extends Command
final case class Commit(value: Int, nodeId: String) extends Command
final case class BroadcastR(rBroadcast: Node.RBroadcast) extends Command
final case class BroadcastA(aBroadcast: Node.ABroadcast) extends Command
final case class BroadcastB(bBroadcast: Node.BBroadcast) extends Command
```

- Keeps track of all nodes
- Asks each node to start Archipelago inside them
- Accepts final *commit*

Node Group

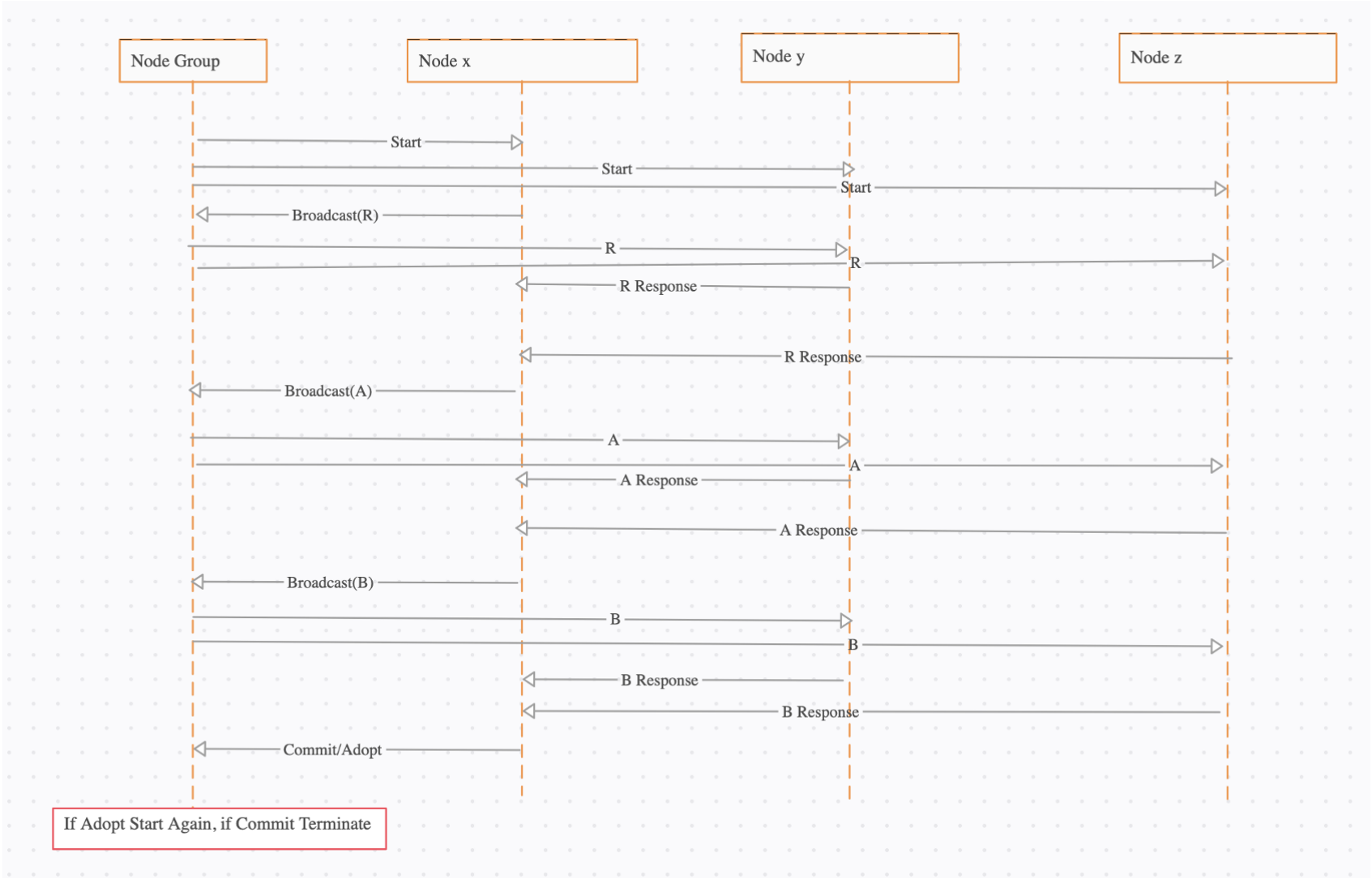
```
sealed trait Command

final case class RequestTrackDevice(nodeId: String, replyTo: ActorRef[Main.Command])
final case class Start() extends Command
final case class Commit(value: Int, nodeId: String) extends Command
final case class BroadcastR(rBroadcast: Node.RBroadcast) extends Command
final case class BroadcastA(aBroadcast: Node.ABroadcast) extends Command
final case class BroadcastB(bBroadcast: Node.BBroadcast) extends Command
```

- Keeps track of all nodes
- Asks each node to start Archipelago inside them
- Accepts final *commit*
- Allows nodes to broadcast messages

Node

```
sealed trait Command

final case class RBroadcast(i: Int, v: Int, replyTo: ActorRef[Node.Command]) extends Command
private final case class RResponse(i: Int, R: Set[(Int, Int)]) extends Command
final case class ABroadcast(i: Int, v: Int, replyTo: ActorRef[Node.Command]) extends Command
private final case class AResponse(i: Int, aJ: Set[Int]) extends Command
final case class BBroadcast(i: Int, flag: Boolean, v: Int, replyTo: ActorRef[Node.Command]) extends Command
private final case class BResponse(i: Int, bJ: Set[(Boolean, Int)]) extends Command
final case class Start(v: Int) extends Command
final case class Stop() extends Command
```

Sequential Diagram

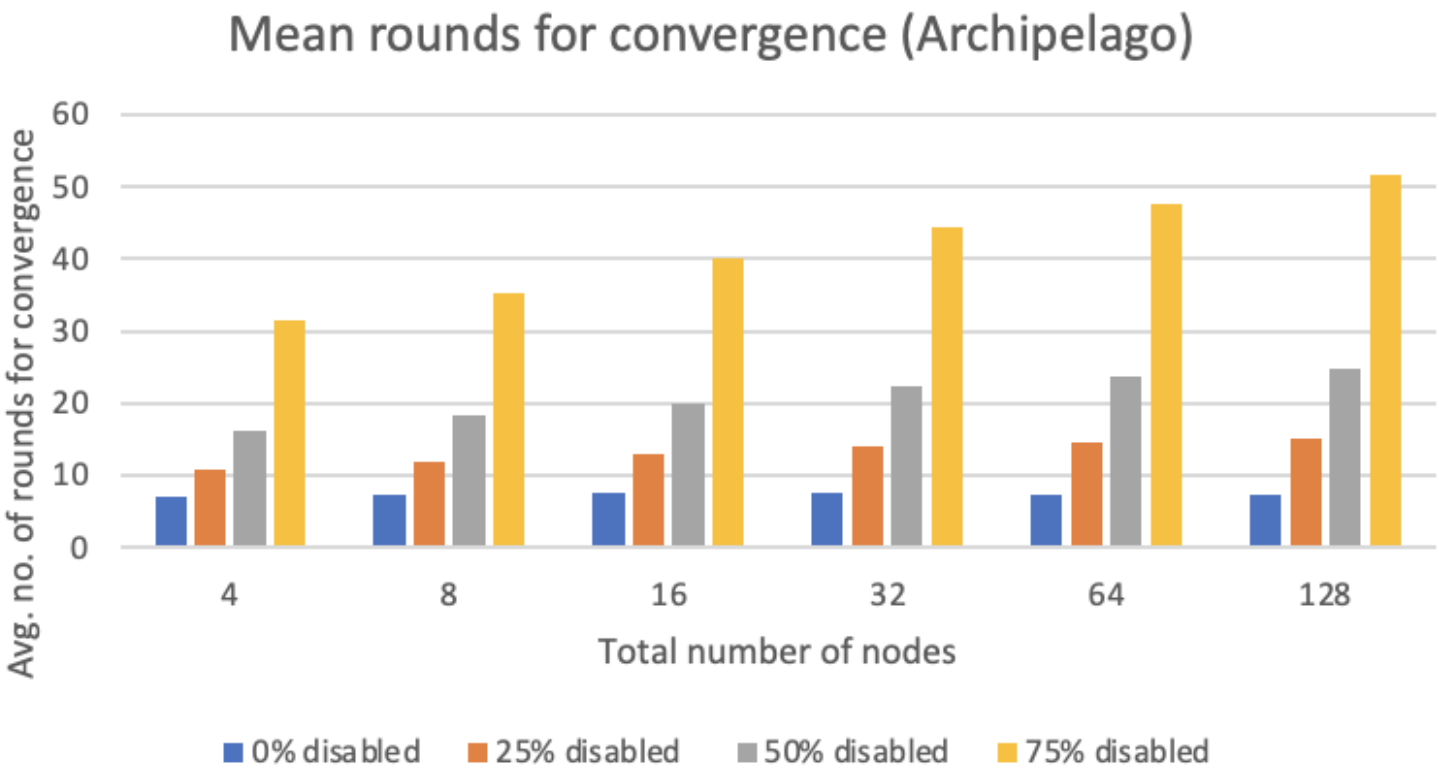


Archipelago with message passing

Demo

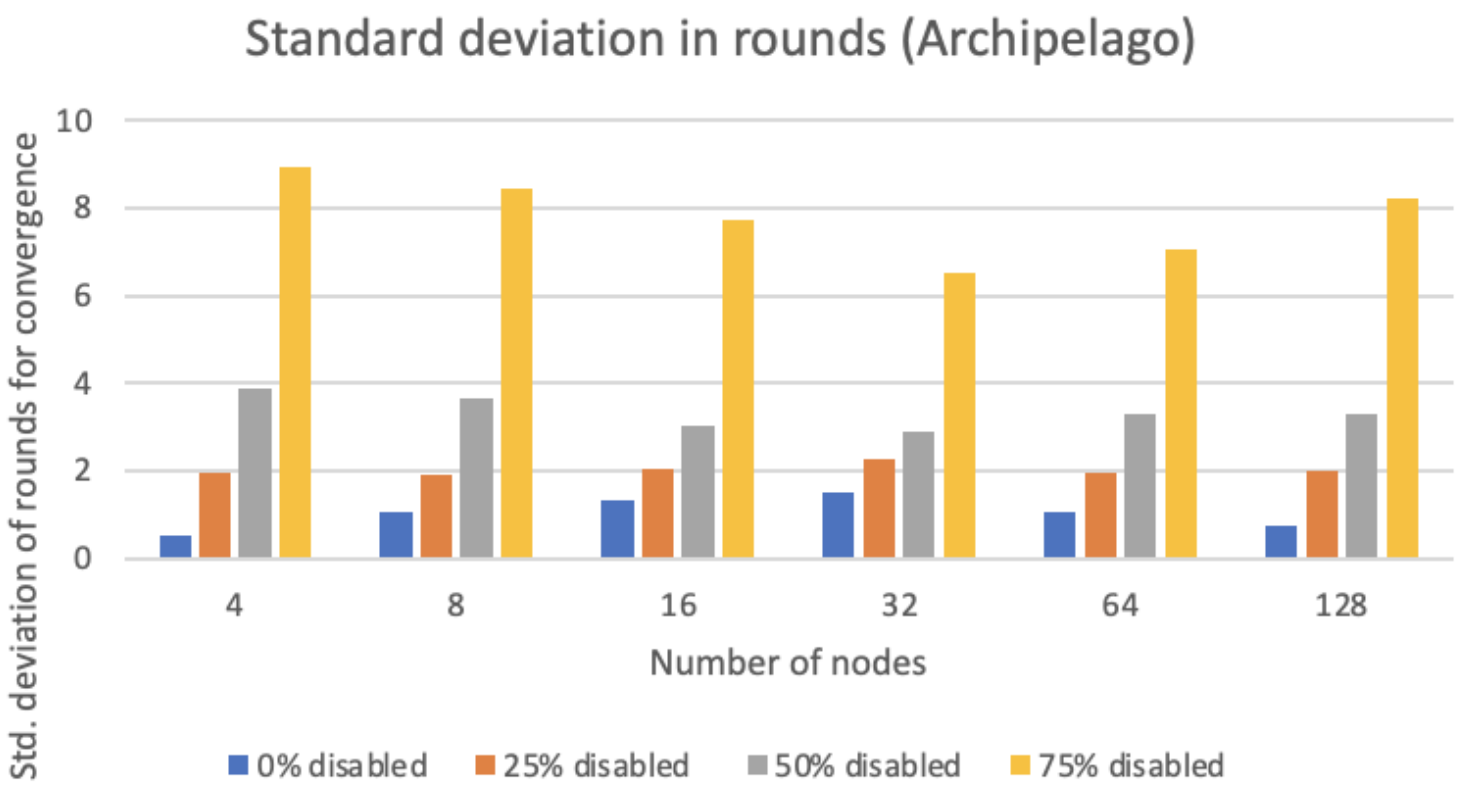
Experiments - Archipelago

Node Count	0% disabled	25% disabled	50% disabled	75% disabled
4	7.09	10.83	16.05	31.51
8	7.27	11.8	18.33	35.38
16	7.57	13.03	19.92	40.12
32	7.63	13.9	22.35	44.44
64	7.36	14.45	23.69	47.61
128	7.21	15.1	24.91	51.78



Experiments - Archipelago

Node Count	0% disabled	25% disabled	50% disabled	75% disabled
4	0.509901951	1.949358869	3.866522986	8.958236434
8	1.053565375	1.892088793	3.66469644	8.444524853
16	1.326649916	2.061552813	3.016620626	7.745966692
32	1.493318452	2.289104628	2.875760769	6.547518614
64	1.067707825	1.977371993	3.292415527	7.084490102
128	0.768114575	2.0174241	3.281767816	8.223138087

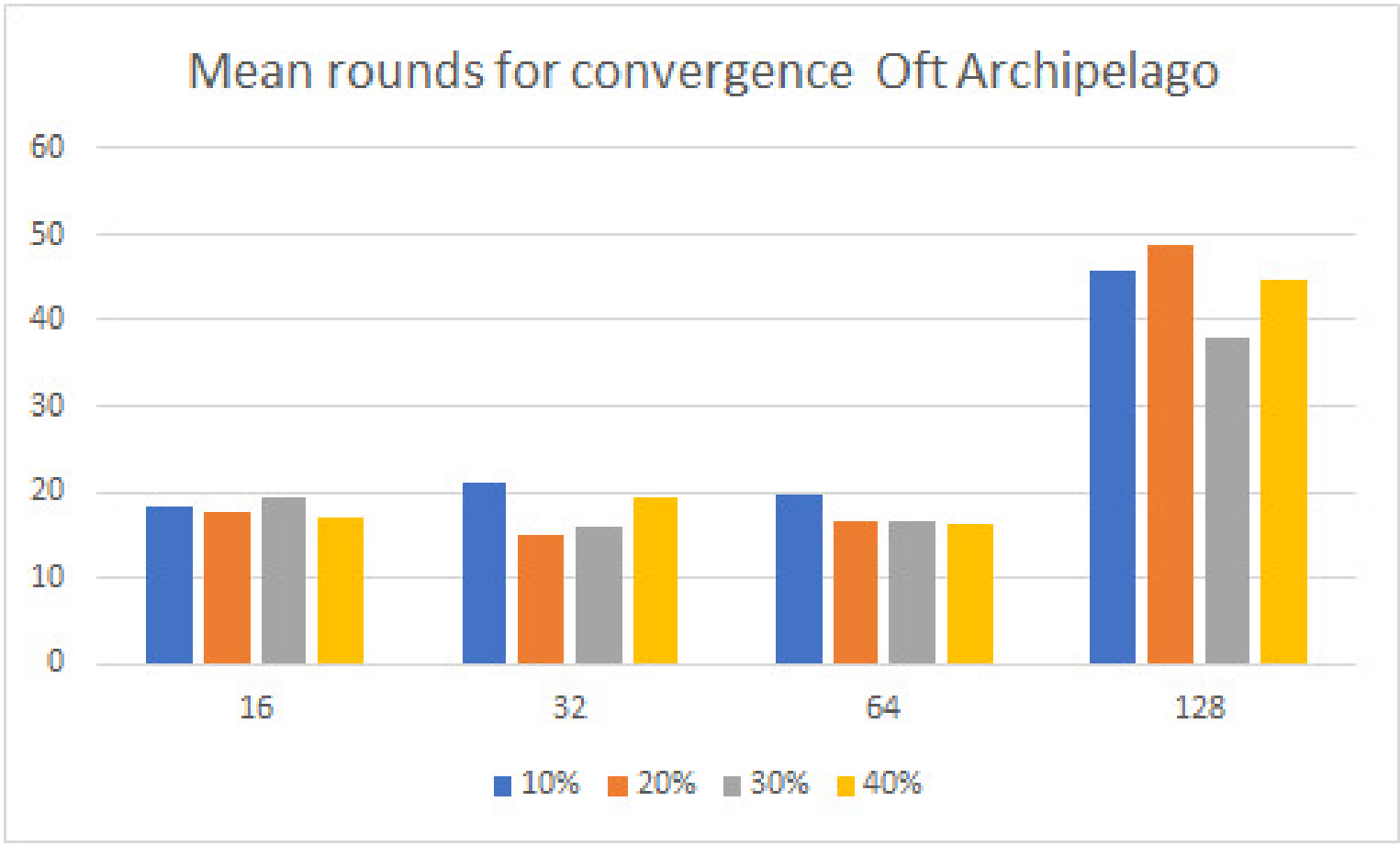


Experiments - Failure rate Archipelago adversary mode

Round limit	Convergence(Y/N)
10	N
100	N
1000	N

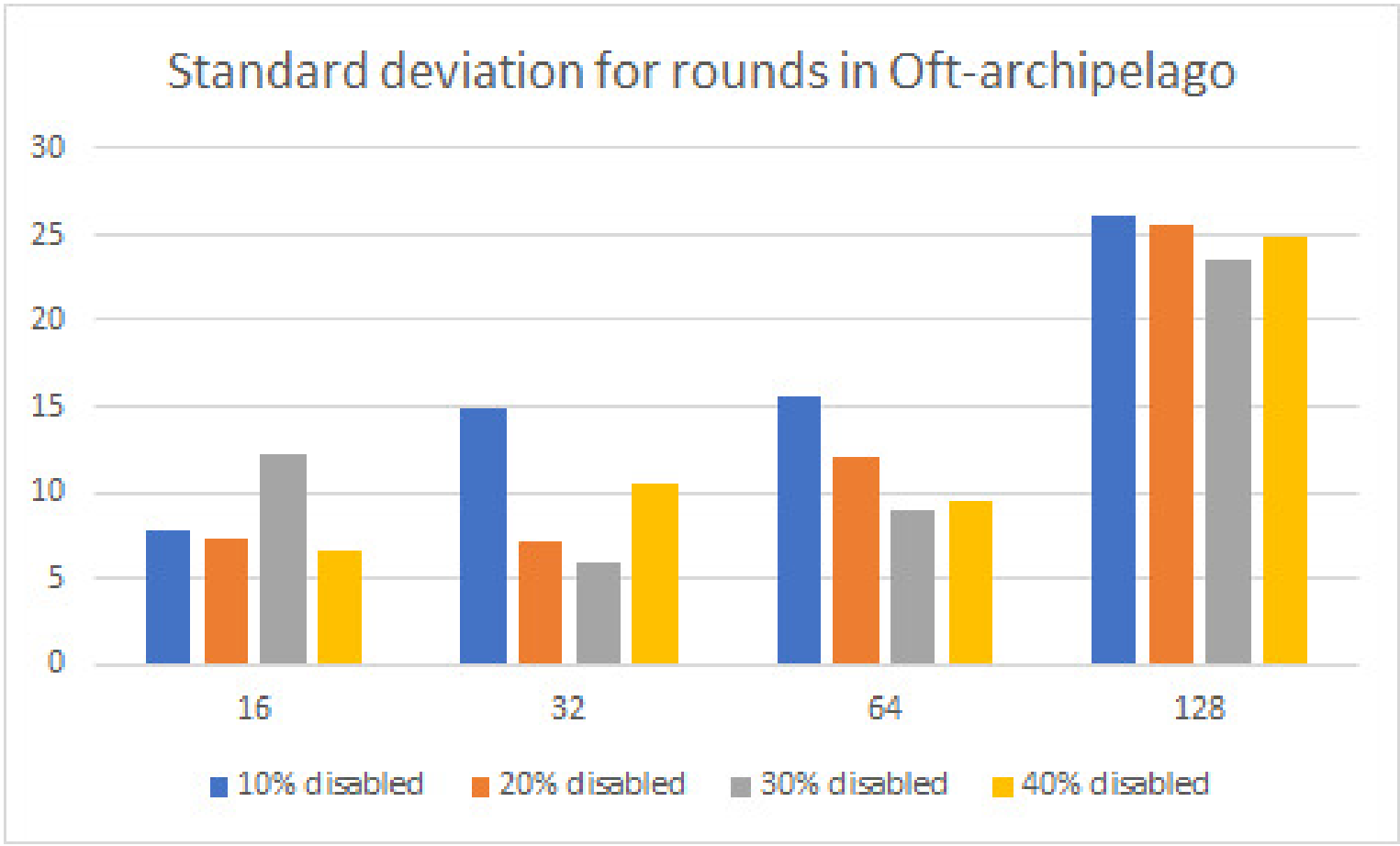
Experiments - Archipelago OFT

Node Count	10% disabled	20% disabled	30% disabled	40% disabled
16	18.24	17.8	19.34	16.8
32	21.05	14.91	15.78	19.5
64	19.69	16.52	16.48	16.16
128	45.92	48.75	37.82	44.69



Experiments - Archipelago OFT

Node Count	10% disabled	20% disabled	30% disabled	40% disabled
16	7.78	7.36	12.2	6.69
32	14.93	7.08	5.99	10.43
64	15.66	11.96	8.97	9.47
128	26.05	25.59	23.56	24.9



Conclusions

- Archipelago works well in practice for shared memory case despite theoretical limitations (for random disabling)

Conclusions

- Archipelago works well in practice for shared memory case despite theoretical limitations (for random disabling)
- Once proposed cap is reached, no more guarantees as it can be countered with a "stronger" adversary strategy

Conclusions

- Archipelago works well in practice for shared memory case despite theoretical limitations (for random disabling)
- Once proposed cap is reached, no more guarantees as it can be countered with a "stronger" adversary strategy
- Difficult to implement for all scenarios because of the need to perform round synchronization (expensive)

Conclusions

- Archipelago works well in practice for shared memory case despite theoretical limitations (for random disabling)
- Once proposed cap is reached, no more guarantees as it can be countered with a "stronger" adversary strategy
- Difficult to implement for all scenarios because of the need to perform round synchronization (expensive)
- Archipelago with message passing remains consistent in terms of performance regardless of the percentage of nodes disabled

Conclusions

- Archipelago works well in practice for shared memory case despite theoretical limitations (for random disabling)
- Once proposed cap is reached, no more guarantees as it can be countered with a "stronger" adversary strategy
- Difficult to implement for all scenarios because of the need to perform round synchronization (expensive)
- Archipelago with message passing remains consistent in terms of performance regardless of the percentage of nodes disabled
- However, for large number of nodes (128) the mean and standard deviation increases significantly. Indicating that it takes longer to converge and that there is more volatility in rounds for convergence

Conclusions

- Archipelago works well in practice for shared memory case despite theoretical limitations (for random disabling)
- Once proposed cap is reached, no more guarantees as it can be countered with a "stronger" adversary strategy
- Difficult to implement for all scenarios because of the need to perform round synchronization (expensive)
- Archipelago with message passing remains consistent in terms of performance regardless of the percentage of nodes disabled
- However, for large number of nodes (128) the mean and standard deviation increases significantly. Indicating that it takes longer to converge and that there is more volatility in rounds for convergence
- Archipelago OFT obeys the O(n) theoretical limit for total rounds

Conclusions

- Archipelago works well in practice for shared memory case despite theoretical limitations (for random disabling)
- Once proposed cap is reached, no more guarantees as it can be countered with a "stronger" adversary strategy
- Difficult to implement for all scenarios because of the need to perform round synchronization (expensive)
- Archipelago with message passing remains consistent in terms of performance regardless of the percentage of nodes disabled
- However, for large number of nodes (128) the mean and standard deviation increases significantly. Indicating that it takes longer to converge and that there is more volatility in rounds for convergence
- Archipelago OFT obeys the O(n) theoretical limit for total rounds
- Once proposed limit

$$f = \frac{n-1}{2} \tag{1}$$

for disabled processes is reached, converging is not guaranteed.