When One Becomes n: Realising Distributed Concurrency with ARBOC

Samarth Bhat Y.[1⋆[0009−0003−0530−1283]], Varun Shenoy[1⋆[0009−0000−1087−4033]], Vibhav Tiwari[1⋆[0009−0009−8381−0714]], Samar Garg[1⋆[0009−0009−9898−7192]], and Gowri Srinivasa[1[0000−0002−3568−6749]]

[1]PES Center for Pattern Recognition, Dept. of Comp. Sc. and Engg., PES University, Bengaluru, India
samarthbhaty@gmail.com, shenoy.varun@protonmail.com, vibhavtiwari.official@gmail.com, aries.samar@gmail.com, gsrinivasa@pes.edu

Abstract. Parallel programming remains notoriously difficult, primarily due to the cognitive load of managing synchronisation and nondeterministic state across distributed nodes. While Behaviour-Oriented Concurrency (BoC) simplifies this on single-system architectures by unifying parallelism and coordination, its reliance on shared memory limits its horizontal scalability. This paper presents ARBOC (Arbitrated Runtime for Behaviour Oriented Concurrency), a distributed runtime that extends BoC across multiple systems while preserving its core semantic guarantees. ARBOC separates coordination from execution through centralised arbitration, scheduling behaviours via a dependency-aware directed acyclic graph and executing them on distributed workers without exposing programmers to message passing, distributed locks, or explicit synchronisation. We describe the execution semantics, architecture, and implementation of ARBOC, and evaluate it on a range of parallel and coordination-intensive benchmarks. Results show that ARBOC scales effectively for parallelisable compute-heavy workloads while retaining predictable behaviour under contention, demonstrating that BoC's abstraction naturally generalises to distributed environments.

Keywords: Parallel Programming · Distributed Systems · Behaviour Oriented Concurrency · Concurrency

1 Introduction

Parallel programming is inherently complex, particularly when it requires expressing the coordination of concurrent, atomic tasks. BoC addresses this complexity by unifying parallelism and coordination into a single abstraction. This enables asynchronous, atomic, and ordered work units over independent resources. However, the current single-system design limits scalability and precludes the performance benefits of distributed environments.

⋆ All authors have equal contributions.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_43

This paper presents ARBOC (Arbitrated Runtime for Behaviour Oriented Concurrency)[1], which extends Behaviour-Oriented Concurrency [5, 4] to support execution across multiple systems. The original guarantees of exclusive access, atomicity, and ordering are preserved while facilitating the coordination of shared dependencies for isolated data objects. It is demonstrated that the model naturally generalises to distributed settings without exposing programming complexities such as explicit message passing, distributed locking, or conflict resolution. The extended model is presented herein to address distribution challenges and outline how simplicity and expressiveness are retained at scale.

Prior work approaches parts of this problem but makes different trade-offs. Ownership-based languages such as Rust [7] and capability-based runtimes like Pony [6] enforce safety via type systems, but push synchronisation concerns onto the programmer and do not directly address distributed execution. Actor-based systems [9, 1] like Erlang [2] scale well across nodes, but inherently expose non-determinism in message ordering and state evolution. DAG-based schedulers provide deterministic execution for dataflow and batch workloads, but are not designed for fine-grained, contended mutable state with exclusive access requirements. Related concurrency models include isolation-typed actors [10], capability-based type systems [8], and composable integration of actors with other paradigms [11].

ARBOC occupies a distinct point in this design space. By retaining BoC's cown-based exclusivity and deterministic scheduling, while introducing centralised arbitration and distributed execution, ARBOC preserves the simplicity and reasoning model of BoC even when scaled across a cluster.

2 Background

2.1 Scope and Assumptions

(1) **Reliable network:** the cluster networking must guarantee reliable and in-order data delivery. (2) **Fail-stop processes:** processes may fail by stopping execution but do not exhibit Byzantine behaviour. (3) **Closed-world resource definition:** the complete set of resource objects and tasks constituting a workload must be statically defined in advance. These assumptions allow us to focus on the core challenge addressed in this work—extending BoC's coordination semantics to distributed systems while preserving the simplicity of its programming model. Fault tolerance, failure detection, and network partition handling, while important, are prospects of future work.

2.2 Core Concepts

ARBOC defines its fundamental primitives and execution semantics based on the principles of Project Verona [3], extending its concepts of concurrent ownership and ordered execution of discrete work units to a distributed environment. The

[1] The source code for ARBOC: https://github.com/CPR-research/ARBOC

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_43

following concepts, defined by BoC serve as guidelines for the execution model. A *cown (concurrent owner)* serves as an abstraction used to manage an isolated piece of data involved in concurrent tasks and provides the only point of entry to mutate and read the resource. This ensures mutual exclusion across concurrent behaviours. A cown may exist either in the *available* or *acquired* state. The *behaviour* forms a unit of concurrent execution, and is defined by the `When()` construct, which takes cowns as parameters.

Execution Semantics Two primary semantics are enforced to ensure safe concurrent operation. **(i) Atomicity of Ownership:** A behaviour begins execution only after it has acquired exclusive access to its complete set of required cowns. This ensures that the behaviour's operations are isolated and data-race free, and **(ii) The Happens-Before Relation:** ARBOC maintains deterministic execution through the *happens-before* relationship - *a behaviour b will happen before another behaviour b' iff b and b' require overlapping sets of cowns, and b is spawned before b'.*

2.3 Overview

The system follows a centralised-scheduling, distributed-execution pattern and is organised into four primary stages:

1. **Definition Phase:** An ARBOC program defines all behaviours and cowns prior to the top-level `When()` call which schedules the initial behaviour. At this stage, behaviours are added to a node-local registry and each cown is statically assigned a home node.
2. **Scheduling Phase:** When execution encounters a `When()` call, the corresponding behaviour is submitted to the centralised Dispatcher that maintains a Directed Acyclic Graph (DAG) which tracks cown dependencies. The DAG ensures that a behaviour gets dispatched with exclusive access to its required set of cowns. Resource contention is thus converted into a well-defined execution order.
3. **Distribution Phase:** When a behaviour becomes unblocked in the DAG, i.e., it reaches the root position for all cown dependencies, it is dispatched to a message broker. The broker delivers the behaviour to one of the available Executors in the distributed cluster.
4. **Resolution Phase:** The assigned Executor executes the behaviour and returns the updated state of the associated cowns to the master process. Upon completion, the Dispatcher updates the DAG, unblocking and triggering subsequent behaviours waiting on the same cowns. (see Fig. 1).

3 System Architecture

3.1 The Dispatcher

Script Execution and Behaviour Capture The Dispatcher handles workload scheduling throughout the system. The user script runs on the Dispatcher,

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_43

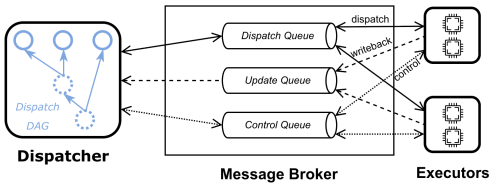

Fig. 1: ARBOC architecture showing a single Dispatcher with a message broker and multiple Executors (each Executor can receive a dispatch, control and write-back message).

which creates behaviour instances upon encountering a `When()` call and enqueues it into the DAG. **The DAG** Scheduling in ARBOC is built around a DAG where behaviours are represented as nodes, and edges encode dependencies on required cowns. When a new behaviour is scheduled, it is added to the DAG with edges pointing either to the cown itself or to the terminal node of that cown's current dependency chain, ensuring that required access ordering is preserved. Fig. 2 shows a bank transfer behaviour using nested `When()` constructs. The Dispatcher translates this code into the dependency graph visualised in Fig. 2. A behaviour becomes ready for execution on reaching the root level of all its cown dependencies, i.e. when no other behaviours requiring the same cown precede it. At this point, the behaviour and its required cowns are sent to the message broker. For this proof of concept, ARBOC utilises Neural Autonomic Transport System (NATS)[12] – a lightweight, high-performance publish-subscribe messaging system. No explicit load-balancing algorithm was implemented, as NATS defaults to a native random selection approach for message distribution. **Post-Execution Resolution** Once the execution is complete, the Dispatcher updates the modified cown values to reflect the changes in its local records. The behaviour is then purged from the DAG, potentially triggering a cascading effect where newly freed up resources allow a behaviour to be immediately dispatched. **Behaviour Registry** To avoid the overhead of serialising function bytecode to send across the network, ARBOC initialises a node-local *Behaviour Registry* that maps behaviour names to their function definitions. During the initialisation phase, this registry is populated with the complete set of behaviour definitions required for the computation. This architecture enables the Dispatcher to transmit a lightweight execution payload that only consists of the behaviour name and the cowns.

3.2 The Executor

Distributed Execution and Behaviour An instance of ARBOC can host multiple Executors. It is limited only by the capacity of the underlying message broker and network topology. The Executor is responsible for the life-cycle of behaviour execution. Upon receiving a set of cowns and a behaviour identifier, the

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_43

```
type Acc struct {bal int32}
func transfer(src Cown[Acc], dst Cown[Acc], amt Cown[int32]) {
    When(src, amt) { // check
    amount = amt.val
        if src.val.bal >= amt.val {
            When(src) { src.val.bal -= amount } // debit
            When(dst) { dst.val.bal += amount } // credit
        }
    }
}
```

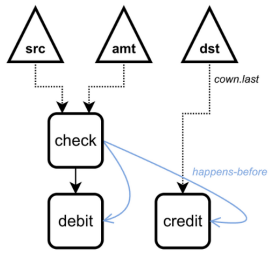

Fig. 2: Bank transfer using nested behaviours and its associated DAG

Executor instantiates a local copy of the behaviour from the Behaviour Registry. It then spawns a dedicated worker thread on which the behaviour runs.

Result Transmission and Modular Deployment Upon completion, the Executor constructs a mapping of the updated cown values which is then sent to the Master via the message queue to synchronize state across the system. The separation between the Master and Executor enables a modular architecture that supports heterogeneous deployments, allowing Executors to operate across diverse hardware platforms and execution environments.

3.3 Communication Infrastructure

The interaction between the Dispatcher and the Executors is mediated by a message broker, which serves as an asynchronous transport layer. This decouples task scheduling from task execution. When a behaviour is ready for execution, its metadata is serialised into a message and published to a work queue. This infrastructure facilitates two critical system properties:

1. **Dynamic Work Distribution:** The broker can act as a load balancer, ensuring that dispatched behaviours are distributed among available Executors. This prevents the Dispatcher from needing to maintain the state or health of individual worker nodes.
2. **Backpressure and Buffering:** By using a queue-based model, the system can handle bursts in behaviour spawns. The broker buffers pending behaviours, allowing Executors to pull work at varying processing rates.

4 Results and Evaluation

ARBOC is evaluated across a suite of six benchmarks **shared by BoC**–representing diverse computational patterns: Parallelisable tasks (*Monte Carlo Pi*, *MatMul*, *Convolution*), recursive workloads (*Fib*), and coordination-heavy scenarios (*Bank*, *Dining Phil*). All benchmarks were executed on dedicated CPU Linode instances (AMD EPYC 7713, 2 core 4GB RAM / 4 core 8GB RAM) within a Virtual Pri-

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_43

Table 1: Benchmark Execution Times (seconds): Comparing Single-Node Baseline (BoC) vs. Distributed Scalability (ARBOC)

Benchmark	**Single-Node Baseline**		**Distributed ARBOC (Cluster)**			
Configuration	**2 Cores** (Local)	**4 Cores** (Local)	**2 Cores** (1 Worker)	**4 Cores** (2 Workers)	**6 Cores** (3 Workers)	**8 Cores** (4 Workers)
Monte Carlo Pi	342.343	301.212	608.490	340.917	225.333	**179.439**
MatMul	100.246	82.926	196.617	111.643	81.349	**68.951**
Fib	**1.325**	1.376	3.221	3.153	3.025	3.227
Bank	**6.340**	5.970	42.025	35.240	33.230	32.080
Dining Phil	27.543	**26.501**	149.008	137.886	133.276	133.175
Convolution	12.743	11.739	22.444	14.722	10.468	**8.206**

vate Cloud. To establish the baseline (BoC), we executed the workloads[2] on two standalone configurations: a 2-core instance and a 4-core instance. For the ARBOC benchmarks, a 4-core dedicated node served as the centralised Dispatcher, managing a cluster of 2-core worker nodes. Performance is measured in total execution time (seconds).

Table 1 presents the execution times for both the single-system baseline (BoC within the ARBOC framework) and the distributed cluster (ARBOC) across various core and worker configurations.

4.1 Baseline and Comparative Analysis

The single-node results establish the performance floor for our architecture. When comparing BoC and ARBOC at the 4-core mark, we observe a performance delta where single-system execution is generally faster (e.g., *MatMul* at 82.9s vs 111.6s). This overhead is attributed to communication costs. However, for compute-heavy workloads like *Monte Carlo Pi*, *Convolution* and *MatMul*, ARBOC begins to outperform the single-node baseline as the cluster scales to 6 and 8 cores. Despite the distribution overhead, the system scales horizontally. Conversely, in highly sequential or small-scale computations like *Dining Phil*, *Fib* and *Bank*, the distribution overhead is more pronounced, since the behaviours are too small to offset the communication overhead.

4.2 Scalability in a Distributed Environment

The distributed results demonstrate varying degrees of speedup tied to the compute-to-coordination ratio of the workload:

[2] Scale: MCPI $\sim 10^{11}$ samples; MatMul/Conv $\sim 4K \times 4K$; Bank used thousands of transactions; Dining Phil/Fib used small fixed setups.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_43

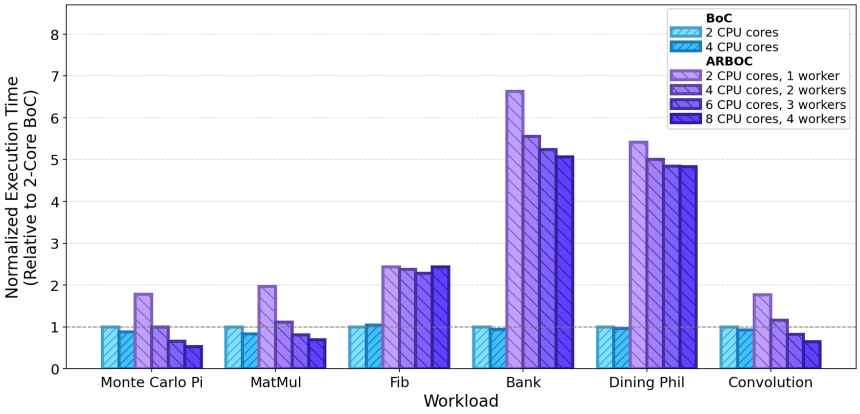

Fig. 3: Normalised execution time relative to 2-core BoC baseline (lower is better) across all configurations.

1. **Performance Leads:** *Monte Carlo Pi* showed the largest speedup, running **67.9% faster** than the best local single-node result, *Convolution* and, *MatMul* achieved a speed up of **43.1%** and **20.3%** respectively.
2. **Coordination Bottlenecks:** Workloads with high resource contention, such as *Bank*, showed minimal improvement. Performance suffers when many behaviours contend over the same cowns, because the Dispatcher only permits exclusive access.
3. **Overhead Analysis:** Small, rapid-fire and inherently sequential workloads (*Fib*) show nearly flat performance across core counts. *Dining Phil* ran slower on our system, an expected result given the sequential nature of the problem; the latency introduced by the message broker and DAG scheduling logic outweighs the benefits of parallel execution. As for *Bank*, its nested behaviour spawning pattern means the Dispatcher repeatedly serialises and re-dispatches behaviours across the cluster, making serialisation and dispatch overhead the dominant cost.

5 Conclusion

This paper introduced ARBOC, a distributed runtime for the Behaviour-Oriented Concurrency (BoC) model implemented in Go. By utilising a centralised arbitrator and distributed executors, the system achieves a total decoupling of coordination from execution. It proves that BoC abstractions can be preserved in a distributed setting without resorting to manual locking. Further, ARBOC shows near-linear scaling on compute-bound, highly parallel workloads, maintaining efficiency as the system scales and demonstrating the practicality of the approach for large deployments. While ARBOC successfully extends BoC, there are still several areas for improvement. The centralised Dispatcher poses as a single point of failure and a possible bottleneck for behaviour spawning at scale. Fault tolerance for node failures during atomic behaviour execution remains a requirement

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_43

for production systems. ARBOC demonstrates that Behaviour-Oriented Concurrency can scale to distributed settings without losing its simplicity, enabling safe and powerful parallel programming.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Agha, G.A.: Actors: a model of concurrent computation in distributed systems (parallel processing, semantics, open, programming languages, artificial intelligence). University of Michigan (1985)
2. Armstrong, J.: A history of erlang. In: Proceedings of the third ACM SIGPLAN conference on History of programming languages. pp. 6–1 (2007)
3. Arvidsson, E., Castegren, E., Clebsch, S., Drossopoulou, S., Noble, J., Parkinson, M.J., Wrigstad, T.: Reference capabilities for flexible memory management. Proceedings of the ACM on Programming Languages **7**(OOPSLA2), 1363–1393 (2023)
4. Cheeseman, L., Castegren, E., Drossopoulou, S., Wrigstad, T., Clebsch, S., Parkinson, M.: Decoupling isolation and concurrency: An actor-centric view of behaviour-oriented concurrency. In: Concurrent Programming, Open Systems and Formal Methods: Essays Dedicated to Gul Agha to Celebrate His Scientific Career, pp. 165–186. Springer (2025)
5. Cheeseman, L., Parkinson, M.J., Clebsch, S., Kogias, M., Drossopoulou, S., Chisnall, D., Wrigstad, T., Liétar, P.: When concurrency matters: Behaviour-oriented concurrency. Proceedings of the ACM on Programming Languages **7**(OOPSLA2), 1531–1560 (2023)
6. Clebsch, S., Franco, J., Drossopoulou, S., Yang, A.M., Wrigstad, T., Vitek, J.: Orca: Gc and type system co-design for actor languages. Proceedings of the ACM on Programming Languages **1**(OOPSLA), 1–28 (2017)
7. Crichton, W., Gray, G., Krishnamurthi, S.: A grounded conceptual model for ownership types in rust. Proceedings of the ACM on Programming Languages **7**(OOPSLA2), 1224–1252 (2023)
8. Haller, P., Loiko, A.: Lacasa: lightweight affinity and object capabilities in scala. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 272–291 (2016)
9. Hewitt, C., Bishop, P., Steiger, R.: Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In: Advance papers of the conference. vol. 3, p. 235. Stanford Research Institute Menlo Park, CA (1973)
10. Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for java: (a million actors, safe zero-copy communication). In: European Conference on Object-Oriented Programming. pp. 104–128. Springer (2008)
11. Swalens, J., De Koster, J., De Meuter, W.: Chocola: integrating futures, actors, and transactions. In: Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control. pp. 33–43 (2018)
12. Synadia Communications: NATS: The cloud native messaging system. https://nats.io (2024), accessed: 2025

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_43