Parametric Choreographies for Distributed Programming

ANKUSH DAS, Boston University, USA
JOHN LIAGOURIS, Boston University, USA

Existing choreographic languages have limited support for expressing protocols that are parametric in the number of parties. On the other hand, a broad class of distributed protocols, including protocols for consensus, atomicity, scheduling, and secure multiparty computation are parametric in nature. To close this gap, we introduce ParaChor, a language that allows implementation of choreographies that are parametric in the number of parties. ParaChor does so by introducing a simple yet expressive looping construct along with the notion of abstract party variables and sets. ParaChor's type system ensures that programs written are deadlock-free by ensuring that party sets do not overlap. ParaChor also features an efficient static endpoint projection algorithm that generates local executable code for each party. We also demonstrate that a standard substitution-based semantics is unsound for ParaChor and requires a sophisticated treatment of environments to be sound. To this end, we define a novel environment-based semantics for ParaChor that supports out-of-order executions, thus covering all possible thread interleavings. In addition to progress and preservation, our technical contributions include a simulation theorem that relates the choreography semantics with those of the projected language. We have also implemented a prototype for ParaChor that uses OCaml expressions as its base language for computation. To demonstrate its expressivity, we implement a wide range of standard distributed protocols in ParaChor including two-phase commit, MapReduce, the Mesos cluster manager, the BGW cryptographic protocol, and the log replication phase of the Raft consensus protocol.

1 Introduction

Distributed protocols are notoriously hard to implement, with concurrency being one of the biggest challenges faced by protocol developers. Accounting for all possible orderings under which a concurrent program can execute is cumbersome and error-prone. Choreographic programming [4–6, 14, 15, 18] provides a viable solution but has still not been widespread adoption in protocol development. One of the reasons for this limited applicability is that many protocols are parametric in the number of parties, meaning that they require looping over party sets, e.g., to express their intended computation and communication patterns. For instance, commitment protocols like two-phase [3, 9] and three-phase commit [19], consensus protocols like Raft [16] and Paxos [12, 13], or data processing models like MapReduce [7] are all defined and operate on a parametric number of parties. In contrast, support for writing choreographies that are parametric in the number of parties involved is fairly limited.

This paper addresses this limitation by introducing ParaChor, a core calculus that allows programmers to implement choreographies parametric over parties. ParaChor is equipped with a novel looping ability, syntactically expressed using the construct $\text{for } p \in P.\, C$. Abstractly, this construct relies on the notion of a party variable p and a party set P, informally describing a choreography C that must be executed by each party p in the set P. As a simple example, the expression $\text{for } p \in P.\, \text{coord}.e \rightsquigarrow p.x$ allows a coordinator party coord to broadcast the value of e to a set of participant parties in the set P. This simple construct enables us to express a variety of distributed protocols, while maintaining the aforementioned benefits of choreographies.

Although the looping construct itself is straightforward, it adds several complications both to the metatheory and implementation of ParaChor. First, preventing self-communication in the presence of party variables is no longer trivial. For instance, the choreography $\text{for } p \in P.\, \text{for } q \in Q.\, p.e \rightsquigarrow q.x$ will result in self-communication if party sets P and Q overlap. To resolve this issue, we require that

Authors' Contact Information: Ankush Das, Boston University, Boston, Massachusetts, USA; John Liagouris, Boston University, Boston, Massachusetts, USA.

party sets must be *disjoint* by construction, coupled with a type system enforcing and preserving this constraint, thereby guaranteeing that a well-typed choreography is deadlock-free.

The second complication is in defining the *endpoint projection* for ParaChor. The projection of a looping choreography onto a party crucially relies on whether that party is being iterated upon or not. For instance, given a concrete party c, the endpoint projection of for $p \in P$. $p.e \rightsquigarrow c.x$ on a party $p \in P$ is simply sending e to c. On the other hand, the projection of the same program on c is itself a for-loop that receives from each party $p \in P$. As programs become more complex with nested loops, defining the endpoint projection, which internally relies on tracking the membership of each party in party sets, becomes intractable.

The third and final complication is in the semantics of ParaChor. Prior works [11, 17] base their semantics on *substitution*, which becomes complicated in the presence of party variables. Due to the presence of two-level variables in ParaChor, e.g. $p.x$ that refers to a variable x at party variable p, a simple substitution of a variable with a value is unsound. We solve this problem by designing a novel *environment-based* semantics for ParaChor. To determine the value of $p.x$, we first look up the value of party variable p, and then look up p's environment to determine the value of x. This not only simplifies the semantics but leads to efficient evaluation for parametric programs.

To prove the soundness of ParaChor and its endpoint projection, we prove the standard type safety theorems of progress and preservation, and an additional *simulation* theorem that connects the semantics of ParaChor to that of the projected control programs.

We have implemented a prototype for ParaChor in OCaml using a subset of OCaml as the base language (often called the local language). To evaluate ParaChor, we implement 5 major protocols as choreographies: two-phase commit (2PC) [9], MapReduce [7], Mesos [10] cluster manager, BGW [2] protocol for secure multiparty computation, and Raft log replication [16]. Since each of these protocols is parametric in the number of parties, loops are crucial to their implementation. For each example, we describe the critical components of the protocol implementation, along with the efficiency of the endpoint projection. We also demonstrate that the generated projections are efficient and the semantics ensure maximal parallelism in execution.

2 ParaChor By Example

We highlight the features of ParaChor by demonstrating a simplified implementation of the two-phase commit protocol [9] (Figure 1). Line 1 iterates over an *abstract party set* denoted by Participants using a *party variable* p. Our first contribution is the introduction of this looping construct over a party set, which is *(i)* treated abstractly by the choreography, but is *(ii)* later concretized during endpoint projection to obtain a closed program. This is written as for $p \in$ Participants. C_1; C_2 to bind a party variable p in the choreography C_1. Semantically, this expresses executing choreography C_1 for all participants in an *arbitrary* order, followed by C_2. ParaChor allows both kinds of parties: party constants (e.g., coord in Fig. 1 to mean coordinator), and party variables (e.g., p) that iterate over party sets. Line 2 denotes the exchange of the Prepare message between the coordinator coord and participant p. The first loop then terminates with an explicit end (line 3) concluding the first part of the prepare phase. In the second loop (line 4), participant p then calls the prepare function, binding its result to variable *vote* (line 5), which is then communicated back to the coordinator (line 6). The coordinator stores the received votes in an *associative array* called *votes* indexed by the participant's id, followed by loop termination (line 7). After the second loop terminates, the coordinator arrives at a decision to commit or abort by counting the votes (line 8). The count_votes function returns true if all participants have voted yes and false otherwise. The third loop (line 9) iterates over the participants and *branches* on the value of *decision*. In the then branch, coordinator sends **L** to each participant (line 11), who then commits the transaction (line 12).

```
1  for p ∈ Participants.                                  # Prepare Phase I: Iterate over all participants
2    coord.Prepare ⇝ p._;                                 # Coordinator sends Prepare to each participant
3    end;                                                 # Prepare Phase I: End
4  for p ∈ Participants.                                  # Prepare Phase II: Iterate over all participants
5    let p.vote = p.(prepare ()) in                       # Each participant determines its vote
6    p.vote ⇝ coord.votes̃[p];                             # Each participant sends its vote to coordinator
7    end;                                                 # Prepare Phase II: End
8  let coord.decision = coord.(count_votes votes̃) in     # Coordinator counts all votes
9  for p ∈ Participants.                                  # Commit Phase: Iterate over all participants
10   let p.outcome = if coord.decision                    # Branch based on coordinator's decision
11                   then coord[L] ⇝ p;                   # If decision is true, send L to each participant
12                        p.(commit ())                   # Each participant commits
13                   else coord[R] ⇝ p;                   # If decision is false, send R to each participant
14                        p.(abort ())                    # Each participant aborts
15   in end;                                              # Commit Phase: End
16 coord.decision                                         # Choreography returns coordinator decision
```

```
17 for p ∈ Participants.
18   send Prepare to p;
19   end;
20 for p ∈ Participants.
21   receive votes̃[p] from p;
22   end;
23 let decision = ret(count_votes votes̃) in
24 for p ∈ Participants.
25   if decision
26   then choose L for p; end
27   else choose R for p; end
28 ret(decision)
```

```
29 receive _ from coord;
30 let vote = ret(prepare ()) in
31 send vote to coord;
32 let outcome = allow coord choice
33                | L ⇒ ret(commit ())
34                | R ⇒ ret(abort ())
35 in ()
```

Fig. 1. Implementation of Two-Phase Commit in ParaChor (top); (a) Control program for coordinator coord (bottom left) (b) Control program for $p \in$ Participants (bottom right)

Similarly, in the else branch, coordinator sends **R** to each participant (line 13), who then aborts (line 14). The result of committing or aborting is stored in the variable *outcome* (line 10) on each party, terminating the loop (line 15) with the end construct. Finally, the choreography returns the coordinator's decision (line 16) and terminates.

ParaChor is also *parametric* over the base language of expressions that are central to computation. We can choose any language as our base as long as it satisfies a few requirements. First, the base language needs to support booleans for choreographies to branch. Second, it should provide an associative array data structure (e.g., $\widetilde{votes}$ in Figure 1) along with a unique representation for parties (e.g., using integers or strings). This data structure should support the standard operations of read and write, a requirement necessary for parametric computation. Finally, the type safety of ParaChor relies on the base language itself being type safe, i.e., the standard theorems of progress and preservation must hold for the base language.

Two-phase commit has a clear communication protocol: coordinator interacts with participants and vice-versa. The only way this can create self-communication and a potential deadlock is if the coordinator is one of the participants. Although this is never the case in practice, ParaChor is not aware of this requirement. Fortunately, ParaChor guarantees deadlock-freedom by design using two requirements: *(i)* all abstract party sets must be *pairwise-disjoint*, and *(ii)* whenever we encounter $p.e \rightsquigarrow q.x$, we enforce that the party sets corresponding to p and q are distinct.

This avoids complicated party tracking in the type system and we have found all our examples to naturally satisfy this requirement.

Another noteworthy feature of our type system is the communication in line 6, where each participant sends its vote to the coordinator. What happens if the *votes* variable was not an array but a boolean or integer instead? In this case, the variable would get *repeatedly* overwritten in each iteration and we would not be able to collect the votes from each participant. To tackle this problem, ParaChor requires the programmer to use an associative array to store the votes. On encountering $p.e \rightsquigarrow q.x$, the type system checks if the sender is a concrete party or a variable. In the latter case, we require the variable x to be an array indexed by the party set that p belongs to.

Our endpoint projection relies on the key intuition that the choreography looping construct for $x_p \in P.\, C_1$; C_2 projects down to *(i)* a single loop iteration for any party $p \in P$, and *(ii)* a loop for any party $p \notin P$. This can be observed in Figures 1a and 1b, which describe the endpoint projection for the two-phase commit protocol onto the coordinator and participants, respectively. For the coordinator (Fig 1a), each choreography loop projects down to a loop in the control program (lines 17, 20, 24). Line 2 in the choreography projects to the coordinator sending Prepare to each participant (line 18). Similarly, line 6 translates to the coordinator receiving the votes from each participant (line 21). Line 8 in the choreography projects to line 23, where the coordinator counts the votes by calling the count_vote function and stores it in the variable *decision*. The ret() construct signifies that the enclosed code is written in the base language. Lastly, in the third loop (line 24), based on *decision* (line 25), the coordinator either chooses **L** (line 26) or **R** (line 27) for p. The coordinator finally terminates by returning *decision* (line 28).

In contrast, the control program for a participant is much simpler, as it belongs to the set Participants, and effectively executes a single iteration of each loop. Line 2 from Figure 1 projects to each participant receiving the Prepare message and binding it to _ (line 29). Next, choreography line 5 translates to the participant calling the prepare function, binding the result to *vote* (line 30). Line 31 sends the vote to the coordinator, which is the projection of line 6. Finally, each participant waits for the coordinator's choice (line 32) and commits in the **L** branch (line 33), or aborts in the **R** branch (line 34). Finally, the participant terminates by returning (). Formally, the programmer provides a *concretization* of the party sets for endpoint projection, at which point ParaChor checks that the party sets are pairwise-disjoint.

At first, it might seem that the choreography loops in Figure 1 impose an order in which the coordinator and participants communicate. This is not true, however. Our semantics have been carefully designed to allow the coordinator to communicate with the participants in an *arbitrary* order. Concretely, when the semantics encounters for $x_p \in P.\, C$, it *non-deterministically* chooses a concrete party $p \in P$, executes the loop body C (where x_p is mapped to p) and composes it with a loop that covers the remaining parties. Programmatically, this can be thought of as $C[p/x_p]$; for $x_p \in P \setminus \{p\}.\, C$ where $P \setminus \{p\}$ denotes the set P with p excluded. Applying this argument recursively and making the next non-deterministic choice for the set $P \setminus \{p\}$, we can see that this semantic approach covers all possible orders in which the loop can be unrolled. Following this approach for each loop of the protocol, the coordinator will send the Prepare message to participants, receive the votes, and send the commit/abort messages to participants in arbitrary (likely distinct) orders, covering all possible interleavings.

3 Metatheory of ParaChor

Formally, ParaChor is a first-order choreography language featuring loops, branching, and a restricted form of party variables and sets. Its syntax is described by Figure 2. *Parties*, written p, represent nodes in a distributed system. *Party sets*, written P, represent finite sets of concrete parties, which are used to generate the endpoint projection. Associative arrays are denoted by a

$$
\begin{array}{llll}
\text{Choreographies} & C & ::= & p.e \mid \text{let } p.x = C_1 \text{ in } C_2 \mid \text{let } p.\widehat{x} = \varnothing \text{ in } C \mid \text{if } p.e \text{ then } C_1 \text{ else } C_2 \\
& & \mid & p_1.e \rightsquigarrow p_2.x; C \mid p_1.e \rightsquigarrow p_2.\widehat{x}[p_1]; C \mid p_1[d] \rightsquigarrow p_2; C \\
& & \mid & \text{for } x_p \in P.\, C_1; C_2 \mid \text{end} \mid \mathbb{F}\, \overline{x_f} \\
\text{Parties} & p & ::= & \mathsf{c} \mid x_p \qquad\qquad \text{Sync Labels} \quad d \quad ::= \quad \mathbf{L} \mid \mathbf{R}
\end{array}
$$

Fig. 2. Formal Syntax of ParaChor

hat, e.g., $\widehat{x}$. A choreography C, in its simplest form, can be a *local expression* at a party, denoted by $p.e$. The expression $\text{let } p.x = C_1 \text{ in } C_2$ denotes executing choreography C_1 and binding the result to a local variable x at party p in choreography C_2. Similarly, $\text{let } p.\widehat{x} = \varnothing \text{ in } C$ initializes an empty associative array in choreography C, which is otherwise a regular local variable at party p. Choreographies can branch on the result of a local computation e using $\text{if } p.e \text{ then } C_1 \text{ else } C_2$. The communication primitives are standard, except $p_1.e \rightsquigarrow p_2.\widehat{x}[p_1]; C$ which binds the communicated value to the associative array $\widehat{x}$.

The novel primitive of ParaChor is the one that enables party parametricity in choreographies. A computation can be applied over a set of parties with a *for-expression*, written $\text{for } x_p \in P.\, C_1; C_2$. The *body expression* C_1 must terminate with an *end-expression*, written end, representing the end of the loop. The choreography C_2 denotes the continuation that executes after the loop body.

At first, it might be tempting to provide a traditional substitution-based semantics for ParaChor. Unfortunately, a naive substitution turns out to be unsound. Instead, ParaChor relies upon a novel environment-based semantics that supports out-of-order execution. The semantics are defined in the context of a store σ which contains two kinds of mappings:

(i) $x_p \mapsto c$ mapping a party variable x_p to a concrete party c, and
(ii) $c.x \mapsto v$ to map variable x at party c to value v.

Formally, choreography semantics are defined using the judgment $\langle \sigma_1, C_1 \rangle \overset{R\ B}{\Longrightarrow}_{\mathsf{c}} \langle \sigma_2, C_2 \rangle$. Here, C_1 and C_2 are the initial and resulting choreographies, while σ_1 and σ_2 are the corresponding stores. R is a redex label that describes the form of reduction; this is essential when we establish a simulation between choreographies and their projections. The blocked set B contains the parties that cannot step because they are blocked on a communication; this is essential in defining out-of-order execution as we will demonstrate with formal rules. These rules rely on a semantic judgment $\langle \mu_1, e_1 \rangle \Longrightarrow_{\mathsf{e}} \langle \mu_2, e_2 \rangle$ for the base language where μ_1 and μ_2 are the corresponding local stores for each party and are obtained by projecting the global store σ onto a party. Formally, for a mapping $c.x \mapsto v$ in σ, its projection μ on party c would contain the mapping $x \mapsto v$.

ParaChor features a standard type system whose main purpose is to prevent self-communication, and therefore deadlocks. Typing judgments have the form $\Gamma; \Phi \vdash C : \tau$ where Γ is a typing context and Φ contains the set of parties currently in scope. In addition, we track the party set that each party variable belongs to which is later used to prevent self-communication. The judgment states that under context Γ and set Φ, choreography C has type τ. Context Γ contains declarations of the form $p.x : t$ denoting a local variable x of type t at party p.

The main type safety theorems of ParaChor are the standard progress and preservation. These rely on a judgment $\Gamma; \Phi \vDash \sigma$ defining that a store σ is well-typed w.r.t. a context Γ and set Φ.

Theorem 3.1 (Progress). *If* $\Gamma; \Phi \vDash \sigma$ *and* $\Gamma; \Phi \vdash C : \tau$, *then either* C value *or there exist* R, B, σ', *and* C' *such that* $\langle \sigma, C \rangle \overset{R\ B}{\Longrightarrow}_{\mathsf{c}} \langle \sigma', C' \rangle$.

Theorem 3.2 (Preservation). *If* $\Gamma; \Phi \vDash \sigma, \Gamma; \Phi \vdash C : \tau$, *and* $\langle \sigma, C \rangle \overset{R\ B}{\Longrightarrow}_{\mathsf{c}} \langle \sigma', C' \rangle$, *then* $\exists \Gamma'.\, \Gamma \leq \Gamma'$, $\exists \Phi'.\, \Phi \leq \Phi', \Gamma'; \Phi' \vDash \sigma'$ *and* $\Gamma'; \Phi' \vdash C' : \tau$.

The most important contribution of ParaChor is its endpoint projection algorithm. This algorithm projects down a choreography into individual *sequential* programs that can then be deployed to individual parties who execute in parallel. This is a crucial requirement since there is no practical way to execute a global choreography on a central entity. We establish a correspondence between the semantic steps performed by the choreography and its projected control system.

Theorem 3.3 (Simulation). *If* $\Gamma; \Phi \vDash \sigma, \Gamma; \Phi \vdash C : \tau$, *and* $\langle \sigma, C \rangle \overset{R\ B}{\Longrightarrow}_{\mathrm{c}} \langle \sigma', C' \rangle$, *then there exist* ω' *and* Π' *such that* $[\![\sigma']\!] \leq \omega'$, $[\![C']\!]^{\Phi}_{\mathfrak{L}} \leq_{\mathrm{nd}} \Pi'$, *and* $\langle \omega, \Pi \rangle \overset{L}{\Longrightarrow}_{\mathrm{S}} \langle \omega', \Pi' \rangle$ *where* $\omega = [\![\sigma]\!]$, $\Pi = [\![C]\!]^{\Phi}_{\mathfrak{L}}$, *and* $L = [\![R]\!]$.

Intuitively, this theorem states that if a choreography C can step, then the corresponding system $[\![C]\!]^{\Phi}_{\mathfrak{L}}$ can step. We achieve this by projecting the choreography store σ (under which C steps) into the system store ω (under which Π steps). Furthermore, the redex label R that is used to step the choreography is also related to the projected label $[\![R]\!]$ that is used to step the system (the primary purpose of introducing redices).

4 Implementation and Evaluation

We have developed a prototype of ParaChor in OCaml. Our endpoint projection algorithm is efficient and *linear-time* in the size of the choreography. ParaChor comes with a control language interpreter that simulates distributed execution locally. We evaluate ParaChor using five representative distributed protocols. In addition to two-phase commit, we implement MapReduce [7], the Mesos cluster manager [10], the BGW protocol [2] for secure multiparty computation, and the Raft log replication phase [16]. Figure 3 summarizes the results from our evaluation.

Protocol	LoC	EPP	Local	EPP Time
2PC	32	66	5	0.495
MapReduce	28	70	6	0.403
BGW	79	138	41	1.501
Mesos	52	109	48	0.823
Raft	49	71	36	0.647

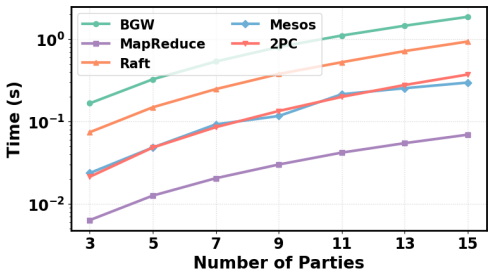


Fig. 3. Total lines of code (left) in ParaChor (LoC) compared to the projected programs (EPP), including lines required for party initialization. Lines of code for local functions are common in both ParaChor and EPP and are reported separately. The rightmost table column shows the time (in milliseconds) that the endpoint projection algorithm takes with 15 parties. The plot (right) shows end-to-end execution times (in seconds) for each choreography while varying the number of parties from 3 to 15.

5 Related Work and Conclusion

Most closely related to ParaChor are works that support *multiply-located values* [1, 8, 17], i.e., values that are known to a set of parties. Instead of abstracting via loops, party concretization in PolyChorλ [8] happens via function application, i.e., calling the function $\Lambda P.C$ with a concrete party. λ_{QC} [17] generalizes this by introducing a new construct AmI $\in P$ then E_1 else E_2 meaning "if the party is in set P, then execute E_1 else E_2". This design choice has trade-offs: while λ_{QC}'s symbolic projection enables deployment with unknown participant identities, it produces more complex projected code with runtime party identity checks. ParaChor's concrete projection requires deployment-time configuration but generates simpler code without runtime branching. The type

systems also differ: ParaChor employs party set disjointness at its core, while λ_{QC} uses a four-kind System F-like system with structural subset judgments ($\rho_1 \subseteq \rho_2 \subseteq \rho_3$) supporting recursive types and richer algebraic types.

In summary, this paper introduces ParaChor to implement choreographies with a parametric number of parties. The main novelty is a simple syntactic construct that allows us to iterate over an abstract set of parties and execute a choreography for each party in that set.

References

[1] Mako Bates, Shun Kashiwa, Syed Jafri, Gan Shen, Lindsey Kuper, and Joseph P. Near. 2025. Efficient, Portable, Census-Polymorphic Choreographic Programming. *Proc. ACM Program. Lang.* 9, PLDI, Article 193 (June 2025), 24 pages. doi:10.1145/3729296

[2] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing* (Chicago, Illinois, USA) *(STOC '88)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/62212.62213

[3] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

[4] Luís Cruz-Filipe, Eva Graversen, Lovro Lugović, Fabrizio Montesi, and Marco Peressotti. 2022. Functional Choreographic Programming. In *Theoretical Aspects of Computing – ICTAC 2022*, Helmut Seidl, Zhiming Liu, and Corina S. Pasareanu (Eds.). Springer International Publishing, Cham, 212–237.

[5] Luís Cruz-Filipe and Fabrizio Montesi. 2020. A core model for choreographic programming. *Theoretical Computer Science* 802 (2020), 38–66. doi:10.1016/j.tcs.2019.07.005

[6] Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2023. A Formal Theory of Choreographic Programming. *Journal of Automated Reasoning* 67, 2 (2023), 21. doi:10.1007/s10817-023-09665-3

[7] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 137–150.

[8] Eva Graversen, Andrew K. Hirsch, and Fabrizio Montesi. 2024. Alice or Bob?: Process polymorphism in choreographies. *Journal of Functional Programming* 34 (2024), e1. doi:10.1017/S0956796823000114

[9] J. N. Gray. 1978. *Notes on data base operating systems.* Springer Berlin Heidelberg, Berlin, Heidelberg, 393–481. doi:10.1007/3-540-08755-9_9

[10] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 295–308.

[11] Andrew K. Hirsch and Deepak Garg. 2022. Pirouette: higher-order typed functional choreographies. *Proc. ACM Program. Lang.* 6, POPL, Article 23 (Jan. 2022), 27 pages. doi:10.1145/3498684

[12] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (December 2001), 51–58. https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[13] Leslie Lamport. 2019. *The part-time parliament.* Association for Computing Machinery, New York, NY, USA, 277–317. https://doi.org/10.1145/3335772.3335939

[14] Fabrizio Montesi. 2013. *Choreographic Programming.* Ph.D. Thesis. IT University of Copenhagen. https://www.fabriziomontesi.com/files/choreographic-programming.pdf.

[15] Fabrizio Montesi. 2014. *Choreographic programming.* IT-Universitetet i København.

[16] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[17] Ashley Samuelson, Andrew K. Hirsch, and Ethan Cecchetti. 2025. Choreographic Quick Changes: First-Class Location (Set) Polymorphism. arXiv:2506.10913 [cs.PL] https://arxiv.org/abs/2506.10913

[18] Gan Shen, Shun Kashiwa, and Lindsey Kuper. 2023. HasChor: Functional Choreographic Programming for All (Functional Pearl). *Proc. ACM Program. Lang.* 7, ICFP, Article 207 (Aug. 2023), 25 pages. doi:10.1145/3607849

[19] D. Skeen and M. Stonebraker. 1983. A Formal Model of Crash Recovery in a Distributed System. *IEEE Transactions on Software Engineering* SE-9, 3 (1983), 219–228. doi:10.1109/TSE.1983.236608