UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

PhD thesis

Scalable and Transactional Actor-Oriented Databases

Yijian Liu

Advisor: Yongluan Zhou

Submitted: October 22, 2024

This thesis has been submitted to the PhD School of The Faculty of Science, University of Copenhagen

Acknowledgements

My PhD work was supported by Independent Research Fund Denmark under Grant 9041-00368B. This PhD journey has been a profound learning experience, and I am grateful to all the people who made it possible. I am deeply thankful for their support, encouragement, and guidance throughout this challenging yet rewarding path.

First and foremost, I express my sincere gratitude to Yongluan, my supervisor. Your expertise and continuous support have guided me through the complexities of this work. You also gave me a lot of freedom to work at my own pace, never imposing undue pressure. This allowed me to independently explore and develop my ideas, knowing I had your guidance whenever needed. I am also especially grateful to the co-authors of my first research work, Li, Yongluan, Vivek, and Marcos. You patiently provided me with a wealth of solid knowledge, from the fundamentals of research methodology to the intricacies of developing and evaluating systems. Your willingness to share your insights and offer constant advice has been a cornerstone of this project, and I am truly grateful for your mentorship.

I would also like to thank my fellow PhD colleagues, Rodrigo, Tilman, and Li, for your accompany and support. Our discussions, whether academic or otherwise, have been a source of motivation and inspiration. You have made the research process less isolating and much more enjoyable.

I would like to give my heartfelt thanks to my partner and my friends, whose delicious meals and cozy gatherings never failed to lift my spirits. The time spent with you brought much-needed balance to my life, making work more enjoyable and the long winter months easier to endure. Your warmth and care truly brightened this journey, and I am deeply grateful for your generous hospitality and mental support.

To my parents and my little sister, words cannot express my gratitude for your unconditional love and encouragement. You have always believed in my ability to accomplish anything, given me the freedom to pursue my passions, and offered your support and understanding. Despite the thousands of miles between us, whenever I return home, I feel like a carefree child once again. You are my greatest source of strength.

Abstract

In modern system design, the "separation of concerns" principle has led to two significant architectural trends: a shift from stateless to stateful middle tiers in the three-tier client-server architecture and the growing adoption of modular, loosely coupled architectures for complex applications. These trends enhance system responsiveness, scalability, and flexibility but also present challenges. The rising popularity of stateful middle-tier architectures, which maintain application states close to application logic in the middle-tier servers, offers numerous benefits but transfers the complexity of state management from backend databases to the application layer. In addition, it is particularly challenging to manage distributed states across loosely coupled components.

The actor model [4] has experienced a resurgence as a tool for building stateful middle tiers and modular systems. By breaking down application logic into independent actors that communicate asynchronously, the actor model simplifies concurrency, scalability, and system resource management. However, its adoption has been limited due to the absence of critical database features such as transaction management, data replication, and data constraint enforcement. In response, the Actor-Oriented Database (AODB) [30] concept has emerged, proposing integrating these database capabilities into actor-based systems.

This dissertation emphasizes the need for further research and development in AODB. Specifically, we propose a scalable and transactional AODB to address the state management challenges for actor-based stateful middle tiers of modern applications. This dissertation achieves the goal through three steps: developing a transaction library for multi-actor transactions, designing a distributed system architecture, and introducing a data model to enable finer-grained state management. The outcome of this dissertation is a fully developed AODB featuring a clear and expressive programming model, a scalable actor-oriented architecture, efficient transaction processing techniques, a prototype implementation built on the Orleans framework, and comprehensive cloud-based evaluation.

Resumé

I moderne systemdesign har "separation of concerns"-princippet ført til to væsentlige arkitekturtendenser: et skift fra stateless til stateful mellemlag i den trelagede klient-server-arkitektur og den voksende anvendelse af modulære, løst koblede arkitekturer til komplekse applikationer . Disse tendenser forbedrer systemets reaktionsevne, skalerbarhed og fleksibilitet, men giver også udfordringer. Den stigende popularitet af stateful middle-tier-arkitekturer, som opretholder applikationstilstande tæt på applikationslogikken i middle-tier-serverne, byder på adskillige fordele, men overfører kompleksiteten af state management fra backend-databaser til applikationslaget. Derudover er det særligt udfordrende at styre distribuerede states på tværs af løst koblede komponenter.

Actor-modellen [4] har oplevet en genopblussen som et værktøj til at bygge stateful mellemlag og modulære systemer. Ved at nedbryde applikationslogikken i uafhængige aktører, der kommunikerer asynkront, forenkler aktørmodellen samtidighed, skalerbarhed og systemressourcestyring. Dets vedtagelse har dog været begrænset på grund af fraværet af kritiske databasefunktioner såsom transaktionsstyring, datareplikering og håndhævelse af databegrænsninger. Som reaktion herpå er konceptet Actor-Oriented Database (AODB) [30] dukket op, der foreslår at integrere disse databasefunktioner i aktørbaserede systemer.

Denne afhandling understreger behovet for yderligere forskning og udvikling i AODB. Specifikt foreslår vi en skalerbar og transaktionsbaseret AODB for at imødekomme de udfordringer ved håndtering af state for aktørbaserede stateful middle tiers i moderne applikationer. Denne afhandling opnår målet gennem tre trin: udvikling af et transaktionsbibliotek til multi-aktør transaktioner, design af en distribueret systemarkitektur og introduktion af en datamodel for at muliggøre en mere finkornet state-styring. Resultatet af denne afhandling er en fuldt udviklet AODB med en klar og udtryksfuld programmeringsmodel, en skalerbar aktørorienteret arkitektur, effektive transaktionsbehandlingsteknikker, en prototypeimplementering bygget på Orleans rammeværket og omfattende cloud-baseret evaluering.

Contents

Chapter 1

Introduction

Modern applications adhere to the principle of "separation of concerns" as a key guideline for designing system architecture. Over the past few decades, two major trends have shaped architectural design. Horizontally, the widely adopted layered architecture, particularly in web applications, has evolved from a traditional stateless middle tier to a stateful middle tier [30, 26, 155]. Vertically, complex applications that once followed a monolithic design now increasingly adopt a modular architecture, where components are loosely coupled [89, 86]. These two trends, while offering benefits, also introduce new challenges.

The three-tier architecture is a stalwart in client-server architectures. Its framework divides functionality into presentation, logic, and data storage layers. Traditionally, the middle tier has hosted application logic, bridging the communication between user interfaces and backend data storage. However, a significant shift is underway. An increasing number of applications are adopting a stateful middle-tier architecture, where the application state is managed within the middle tier and asynchronously saved to backend storage. This approach offers many advantages: it improves responsiveness, supports compute-intensive tasks, gives developers greater flexibility in implementing application logic, reduces reliance on backend storage, and provides a cost-effective way to scale by adding CPUs and memory to the middle-tier servers. However, keeping the application state in the middle tier also means deviating from the inherent state management capabilities of the backend database system. These capabilities are now the responsibility of developers.

Simultaneously, there has also been a shift from monolithic systems to modular, loosely coupled architectures, such as microservices. A loosely coupled system architecture breaks down functionalities into separate components that interact with minimal dependencies, enabling them to operate independently and be easily modified or replaced without affecting the entire system. This architecture brings systems with easier maintenance, independent deployment, scalability, reliability, and flexibility in technology choices. However, it

exacerbates the challenge of managing the state across components, exposing the complexities of distributed systems that developers must navigate.

The actor model [78] proposed more than 50 years ago has now gained increasing popularity for building stateful middle-tiers and developing loosely coupled systems. An actor-based application decomposes application states and functionalities into a large number of actors that communicate with each other via asynchronous messages. The actor model serves as a programming paradigm that speeds up the development of interactive, distributed, and large-scale applications. Key features of the actor model, such as the encapsulated un-shared state, the single-thread abstraction, and the asynchronous communication, simplify concurrency management, facilitate distributed system deployment, optimize resource usage, and enhance system scalability. Moreover, the rapid development of commercial actor frameworks, e.g., Akka, Orleans, brings the actor model to a broader range of applications.

While the actor model offers numerous advantages, its limitations have hindered it from achieving wider adoption. One major drawback is its lack of built-in support for features commonly offered by database systems. In recent years, a novel concept, Actor-Oriented Database (AODB) [30], has been proposed, which promotes the idea of integrating database features into actor systems, such as transaction management, indexing and queries, persistence, data replication, etc. This dissertation highlights the urgent need to advance AODB, an emerging research topic that warrants further exploration and discussion.

Inspired by the few recent research works that contribute to this area, the main objective of this dissertation is to build a scalable and transactional AODB. To achieve this goal, we first propose an efficient transaction management solution for multi-actor transactions, which serves as a fundamental technique for building up other critical features. In the first work, a programming model is provided for developers to write multi-actor transactions and plug-in system-level support for transaction processing. Afterward, an extended system architecture is deployed based on the previous single-node design, which enhances the system with horizontal scalability. Moreover, to deal with various requirements needed in a distributed system, such as elastic scaling, load balancing, and server failures, we also develop an actor migration mechanism that works efficiently under the context of transaction processing. In the last work, we further exploit the capability of the actor model by introducing a more expressive data model for the actor state, which enables the declaration and management of data constraints across actors, facilitates fine-grained transactional actor state management, and enhances the overall system performance.

In summary, this dissertation establishes a state management layer on top of the actor system, which integrates a transaction programming abstraction, an expressive data model for fine-grained actor state management, novel transaction processing techniques, a data dependency management mechanism, and

an efficient actor migration method. Achieving strong transactional guarantees, efficiency, and scalability while enabling these features on the actor system is the main contribution of this dissertation.

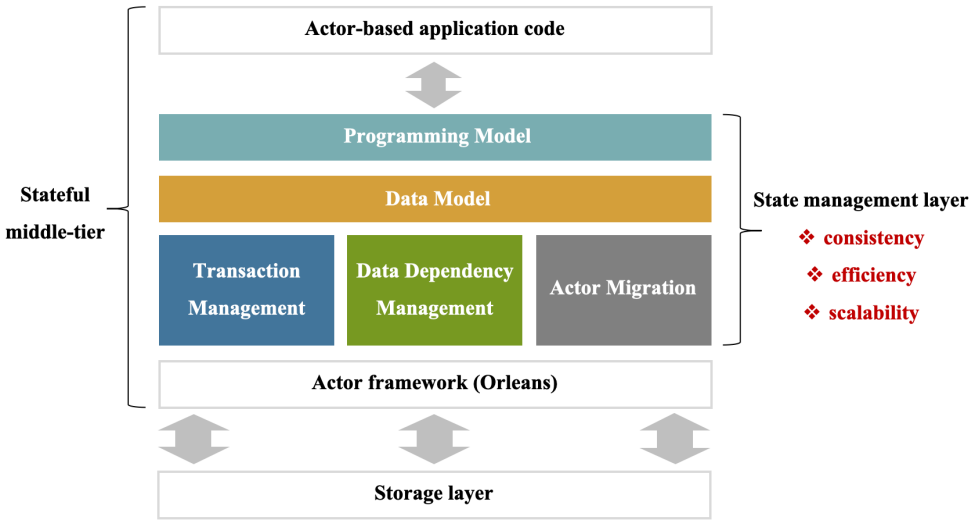


Figure 1.1: Scalable and transactional AODB architecture

1.1 Snapper: A Transaction Library On Actor Systems

The actor model emerges as a promising programming abstraction for building stateful middle tiers for interactive, highly concurrent, and large-scale applications. Though it provides a simplified concurrency model for individual actors, it does not address the concurrency issue for requests that span multiple actors. In this case, cross-actor coordination is needed to guarantee application consistency and correctness. Several research efforts have aimed to ensure ACID properties for multi-actor transactions. However, their performance has been suboptimal, and they fail to fully leverage the unique characteristics of actor-based applications.

In this part of the thesis, we create `Snapper`, a transaction library built on actor systems. In this work, we adopt deterministic transaction execution techniques for multi-actor transactions, which largely reduce the overhead of 2PC, increase the system concurrency level, and improve transaction throughput. In addition, we preserve the traditional transaction processing pattern, 2PL + 2PC, to support transactions that can not adopt deterministic concurrency control. Moreover, `Snapper` can accommodate these two distinct transaction processing modes simultaneously on one system, namely hybrid concurrency control. This is a novel feature that benefits from both the good

performance of deterministic execution and the flexibility of nondeterministic execution. We also design the architecture for `Snapper`, which uses different types of actors to facilitate system functionalities. This architecture is evaluated in a single-node setup and achieves vertical scalability.

1.2 SnapperD: A Scalable Transactional Actor System

The previous work `Snapper` investigates transaction processing techniques that work efficiently on a single-node setup. However, for modern applications, it is insufficient to only achieve vertical scalability, which has limited capability and faces the risk of a single point of failure. Therefore, we find it essential to explore a multi-node architecture for `Snapper`. However, it is non-trivial to support a workload mixing with both distributed and non-distributed transactions while achieving good performance and scalability.

In this part of the thesis, we develop `SnapperD`, a distributed and scalable transactional actor system. In this work, we extend the original design of `Snapper` into a distributed environment. More specifically, we adopt a hierarchical architecture and discover two necessary optimizations (tuned batching and optimized commit protocol) that should be applied to achieve high throughput. Additionally, an actor migration method is developed to enhance the functionality of a distributed actor system. This method seamlessly integrates with the actor transaction processing and significantly reduces transaction aborts caused by data and message loss.

1.3 SnapperX: Fine-Grained Actor State Management

In the actor model, the actor state is typically treated as an opaque object, offering no insight into its structure or access patterns. This limitation leaves complex scenarios, such as referential integrity, data replication, and functional dependencies, as full responsibilities of developers. To alleviate the burden on developers, a dedicated data model for actors is needed, which reveals more information about the actor state and helps the system monitor and control operations on the actor state, therefore enabling system-level support for finer-grained state management tasks.

In this part of the thesis, we propose `SnapperX`, a transactional actor system that enables fine-grained actor state management. In `SnapperX`, each actor's state is modeled as a collection of key-value pairs, and the relations cut across keys on different actors are modeled as update or delete dependencies. With this data model, `SnapperX` provides a more expressive programming abstraction for developers to specify data constraints on top of the actor model.

This data model is integrated with `Snapper` to enable transactional and automatic data constraint enforcement across actors. Furthermore, with the extra information revealed by the data model, `SnapperX` advances the transaction processing techniques by enabling incremental logging and finer-grained key-level concurrency control.

1.4 Publications

This thesis is based on the following three manuscripts:

- **Chapter 2** contains the background information from the below-listed three manuscripts.
- **Chapter 3** is reproduced from a published conference paper: **Liu, Y.**, Su, L., Shah, V., Zhou, Y., Salles, M. A. V. Hybrid Deterministic and Nondeterministic Execution of Transactions in Actor Systems. In *SIGMOD '22: Proceedings of the 2022 International Conference on Management of Data* (June 2022), https://doi.org/10.1145/3514221.3526172. The source code of this work is available on GitHub: https://github.com/diku-dk/Snapper-Orleans.
- **Chapter 4** and **Appendix A** are based on a manuscript: **Liu, Y.**, Su, L., Shah, V., Zhou, Y., Salles, M. A. V. Distributed Snapper: Scalable and Efficient Transaction Execution in Actor Systems. Manuscript in submission (September 2024).
- **Chapter 5** is modified from a published conference paper: **Liu, Y.**, Laigner, R., Zhou, Y. Rethinking State Management in Actor Systems for Cloud-Native Applications. In *SoCC '24: Proceedings of the 2024 ACM Symposium on Cloud Computing* (November 2024), https://doi.org/10.1145/3698038.3698540. The source code of this work is available on GitHub: https://github.com/diku-dk/SnapperX-Orleans.

1.5 Structure of Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 provides an in-depth discussion of the background and the current state-of-the-art. Chapter 3, 4, and 5 present the three main research studies summarized above, respectively. Each of them covers motivation, system architecture design, key techniques, and performance evaluation. Finally, Chapter 6 offers a summary of the dissertation and outlines ongoing and future work.

Chapter 2

Background and State-of-the-Art

2.1 Three-Tier Architecture

The three-tier architecture [143] has long been a cornerstone in software development, particularly since its prominence in the 1990s [68, 37]. This architecture organizes applications into three distinct layers: the presentation tier (user interface), the middle tier (application logic), and the database tier (data storage). Its separation of concerns and modularity quickly made it the architecture of choice for a variety of applications, from enterprise resource planning (ERP) systems [18] to web applications [97].

In a three-tier architecture, a user's request first goes to the presentation tier, which passes it to the middle tier. The middle tier, or application server, processes the request using business logic and may interact with the database to retrieve or update data. After processing, the response is sent back to the presentation tier for display. This middle tier is especially crucial as it manages application logic and coordinates interactions between the user interface and the database.

In recent decades, more and more applications have tended to adopt a stateful middle tier rather than a traditional stateless design. This evolution is driven by the changing characteristics and requirements of modern applications. In the following part of this section, three different implementations of the middle tier are discussed.

2.1.1 Stateless Middle Tier

The classic implementation of the three-tier architecture is with a stateless middle tier. The stateless middle tier does not maintain any application state or session information between requests. Each client request is treated independently by the middle tier, and any required state is stored externally, such

as in a database or client-side storage. The interactions between the middle-tier server and the database usually require a layer of abstraction to map between the middle-tier programming paradigm and the database scheme. For example, ORM (Object-Relational Mapping) [173] is a technique that converts data between an object-oriented programming language and a relational database. ORM also automatically generates basic SQL queries for CRUD (Create, Read, Update, Delete) database operations [104].

One of the most significant advantages of a stateless middle tier is its inherent scalability. Given that each request is independent, multiple application servers can be deployed in a load-balanced configuration, where any server can handle any request without needing access to prior session data. This makes it easier to scale horizontally by simply adding more servers to accommodate increased demand. Besides, individual server failures have a minimal impact on the overall system. Since no session information or application data is stored on the middle-tier server, requests can easily be rerouted to another server in case of a failure, enhancing the system's resilience.

However, while this design makes the system easier to manage and scale, it also introduces significant latency when interacting with the database, especially when handling large datasets or when the database is located remotely. The stateless middle tier either relies on the database to perform all the complex queries directly or retrieves data from the database, processes them in the middle-tier server, and then pushes the updates back to the storage layer. This process depends heavily on the computational power of the database and can be time-consuming due to the network round trips [154]. Besides, the database can become a bottleneck when there are a large number of concurrent requests during peak times. In addition, the stateless middle tier can lead to inefficient data transfer, as the same data may be repeatedly sent from the database to the middle tier for different requests. This **data shipping** paradigm [183] can consume excessive bandwidth, particularly in applications with high data retrieval demands.

2.1.2 Stateless Middle Tier with Caching

Caching is often introduced to mitigate the drawbacks of the stateless middle tier [103, 125, 142, 108]. The caching layer temporarily stores frequently accessed data, reducing the need to fetch it from the database for every request. It can also cache the results of expensive computations, avoiding duplicated calculations for the same task. This setup significantly reduces request latency and database load.

However, caching works as a supplementary layer on top of the primary database or persistent storage if data durability is required; therefore, it does not eliminate the dependency on the storage and the need to scale the storage layer. In addition, the effectiveness of the cache depends heavily on the predictability of data access patterns and data consistency. When requests

require diverse or frequently updated data, cache hit ratios can drop, leading to increased reliance on the database. Moreover, maintaining cache consistency can be challenging in the case of highly concurrent and write-intensive requests. It potentially results in outdated or incorrect data being served and compromising application safety and correctness. Meanwhile, managing cache expiration, invalidation, and synchronization adds complexity to the system.

2.1.3 Stateful Middle Tier

Alternatively, adopting a stateful middle tier becomes a more popular approach. In this model, the middle tier not only processes application logic but also stores and manages the application state directly within its computing nodes. Instead of a data shipping paradigm, where data is constantly moved between the database and the middle tier, the stateful middle tier adopts a **function shipping** paradigm [183]. Here, client requests are sent to the nodes holding the relevant data, allowing the business logic to execute locally. To enhance durability, the updated state can be flushed periodically and asynchronously to the persistent storage.

Holding state in the middle tier differs from caching data. For caching, (1) data are primarily stored in the backend database, which may receive update requests from different sites; (2) various mechanisms are explored to ensure that the cache remains up-to-date and consistent with the database, meaning the database serves as the source of truth in case of conflicts; and (3) the cached data is mainly used for serving read requests. Differently, for the stateful middle tier, (1) the most recent state is kept in the middle tier, accepting requests over the network; (2) the data in storage is mainly for persistence and might be outdated if the updates are not flushed immediately, and (3) the middle tier state serves both read and write-intensive requests within real-time.

The stateful middle tier offers numerous advantages. First, it facilitates interactive and latency-sensitive requests that require real-time response, given that the application state is served directly from the middle tier. Second, it is more cost-effective to scale the middle-tier servers by adding more CPUs and memory space than to scale a database server [15, 16]. Third, instead of encoding transactions as stored procedures to adapt to the interfaces and abstractions limited by the database systems, developers are offered the flexibility to program transactions in any programming language and abstraction in the middle tier. Technologies like ORM allow developers to program business logic on an abstract database using the languages and frameworks they are familiar with. However, these technologies still require the definition or co-design of schemes, structures, or models in both the application layer and the database layer, and they have many limitations, such as the expensive overhead and the limited ability to generate and optimize complex queries [173].

The stateful middle tier encounters various challenges, too. The primary challenge is to conduct state management tasks in the middle tier, such as transaction management, backup and recovery, data integrity enforcement, replication management, index management, etc. Unlike the previous two models, where such tasks are shipped to the database, a stateful middle tier is supposed to conduct these tasks in the application layer.

A straightforward solution is to deploy an in-memory database, which exposes complex data structures such as relational tables (H2 [74], VoltDB [165], SQLite [161]) and key-value collections (Redis [130], Aerospike [163]), provides rich database features, and allows fast access to data from the application codes. However, we still need to face the limitations of a database system, including: (1) Transactions are written as stored procedures. (2) Databases typically lack built-in abstractions for asynchrony [155, 66, 141]. In most databases, such as MySQL, PostgreSQL, and Redis, the transaction model is synchronous, meaning that once a transaction starts, each statement within the transaction is executed one after the other in a blocking fashion. In a database system, asynchronous behavior is usually achieved through special techniques like application-level asynchronous programming [110], background jobs [107, 144], and message queues [137]. (3) A transaction processing middleware [24, 152, 26] is often needed to manage distributed transactions across database servers. And (4) the mapping between objects in the application layer and the data records in the database can be complex and slow (e.g., object-relational impedance mismatch [80, 153]).

The idea of migrating database functionalities to the middle tier by implementing similar algorithms, protocols, or strategies is intuitive and has been explored by plenty of research works [30, 117, 166]. However, a commonly encountered challenge is to integrate the database-oriented features with the application's programming model, data model, architecture, and performance requirements. This dissertation investigates state management challenges and explores solutions for applications that are built with the actor-based (Section 2.3.1) stateful middle tier.

2.2 Loosely-Coupled Architecture

The tiered architecture separates a system **horizontally** by organizing it into layers. Under the principle of separation of concerns, another trend of architectural design is to separate a system **vertically** by dividing it into independent, self-contained components or services, each responsible for a specific business capability from the user interface to data management. Within each component, the layered architecture can be adopted. This loosely coupled architecture has become popular for building applications that need to deliver multiple functionalities and accommodate diverse workloads (e.g., e-commerce [76]).

This architecture is often discussed in contrast to the monolithic sys-

tem [86, 71], where all components are tightly integrated and deployed as a single, unified application. Nowadays, plenty of system design principles and techniques have emerged for developing loosely-coupled systems, including microservices [86, 176], event-driven architecture (EDA) [48], publish-subscribe architecture, actor systems [122, 5], and serverless architecture.

2.2.1 Cross-Component State Management

The loosely coupled architecture provides many promising features, including independent scalability, failure isolation, flexibility to adopt various technologies, fast development and deployment, optimized resource utilization, and easy maintenance. To maximize these advantages, a best practice when building such a system is to reduce inter-dependencies between different components or modules [178, 86]. However, this is non-trivial for many real-life applications. It requires developers to have a high level of expertise and careful planning to design clear, well-defined interfaces and communication protocols. In addition, for scenarios where cross-component coordination and data sharing are inevitable [89], it is significantly challenging to maintain state consistency across multiple components.

Supporting efficient and scalable state management for a distributed system has been a mainstream topic discussed in a variety of research works [77, 185]. Most of them rely on the fundamental trade-offs among consistency, availability, and partition tolerance (CAP theorem [67]). A series of database management systems have emerged to fit different scenarios. For example, strong consistency [22], causal consistency [20], and eventual consistency [19]. Additionally, there are also database management systems that achieve all CAP guarantees by making certain assumptions or imposing specific constraints. For example, deterministic databases (Section 2.5.2.2) utilize deterministic execution of pre-ordered transactions.

Moreover, a loosely coupled system usually relies on indirect communications between components, such as messages, events, and asynchronous network protocols. Ensuring the correctness of the application state in the presence of communication delays, redundancy, and failures, as well as under various delivery guarantees and persistence models, becomes even more complex.

2.2.2 Application-Layer State Management

In a loosely coupled system, an interesting observation is that developers often need to manage states in the application layer [89]. In addition to the benefits of adopting the stateful middle tier, there are many cases where developers find it necessary to manage the state in the application layer because it is insufficient to barely rely on the backend database. For example, there are two popular design patterns applied in loosely coupled systems, particularly mi-

croservices, which also require extra control or maintenance in the application code to ensure application correctness.

One is the Saga pattern [69, 164], which breaks down a complex and long-running distributed transaction into a series of smaller local transactions, each handled by a different service. When a service successfully completes a local transaction, the next service is notified via an event to start another local transaction. If any local transaction fails, the corresponding service invokes a compensation transaction, which will undo changes made by the previous steps, therefore aborting the transaction atomically. The Saga pattern is an alternative method of processing distributed transactions under an eventual consistency guarantee. Though it provides a distributed transaction processing mechanism that naturally matches the loosely coupled system architecture, it imposes a lot of burden on developers to implement correct compensation transactions. In the end, it is the developers' responsibility to maintain the state consistency.

The other one is the CQRS pattern (Command and Query Responsibility Segregation [33]), where the write operations (commands) and read operations (queries) are handled by different services. In this pattern, the write service focuses on maintaining data integrity and consistency under frequent state changes, while the read service is optimized for efficient querying by organizing data specifically for fast retrieval. The read service stays up-to-date by subscribing to the update events from the write services. This pattern is commonly used to manage materialized views that aggregate data from multiple sources. However, a well-known challenge for this pattern is to maintain a consistent view of the read state with asynchronous events sent from different sites. Again, it heavily relies on developers to maintain consistency.

To wrap up, these architectural design trends push developers to implement state management functionalities in the application code due to the lack of system-level support. Their solutions are often tailored to specific cases and can be error-prone. To alleviate the burden on developers, this dissertation identifies an urgent need for a middle-tier solution that effectively and efficiently manages the application state across components in loosely coupled systems.

2.3 Actor Model and Actor Systems

This section discusses the actor model, a programming abstraction that is particularly suitable for building stateful middle tiers of interactive, highly concurrent, and large-scale applications. The actor model also aligns closely with the principles of loosely coupled systems that the system functionalities and application states of an actor-based application are decomposed into hundreds and thousands of actors. In recent years, with the growing popularity of the actor model, the need arose to develop a new type of database — the Actor-

Oriented Database (AODB, Section 2.4) — specifically designed to enhance system-level state management for actor-based applications. This dissertation focuses on providing efficient and scalable transactional state management solutions for AODB.

2.3.1 Actor Model

The actor model was invented by Carl Hewitt, Peter Bishop, and Richard Steiger in 1973 [78] as an efficient and highly parallel architecture for artificial intelligence languages. Later, in 1986, Gul Agha [4] formalized the foundations of the actor model, defined the basic constructs of an actor system, and particularly highlighted actors' potential to achieve extremely high concurrency. In the actor model, actors are the fundamental units of computation, embodying state, behavior, and communication through message passing. The features of the actor model can be summarized as follows:

- **Identification**: Each actor has a unique identifier (e.g., a name, an address/path, or a process ID [59, 5, 122]), allowing it to be explicitly accessed, located, and messaged within the system.

- **Encapsulation**: Similar to objects in OOP (Object-Oriented Programming), each actor encapsulates its own state and behaviors.

- **Asynchronous Communication**: Actors communicate with each other by passing asynchronous messages – an actor delivers a message to another without waiting for the message to be processed and the result to be returned by the receiver actor. In other words, actors are allowed to perform other tasks while waiting for responses. This non-blocking communication between actors reduces idle time, improves the concurrency level, and enhances system scalability. Moreover, it can isolate the failure of one actor from others, embracing the system's resilience.

- **Isolation**: Different from OOP, each actor's state is private, preventing direct access from an actor to another actor's state. An actor requests data from another by passing a message, and an actor only accesses its own state when processing messages. Therefore, a group of actors in one server logically forms a distributed environment.

- **Single-Thread Abstraction**: Each actor processes incoming messages one after the other. While one message is processed, the other received messages are buffered in the actor's "mailbox". The sequential processing of messages builds a single-thread abstraction within each actor such that no state is ever shared between multiple threads. Therefore, no locking or other synchronization mechanisms are needed for an actor.

In an actor system, computations are driven by message passing and processing. Upon receiving a message, an actor processes it according to its defined behavior, which may involve updating its state, sending messages to other actors (or itself), creating new actors, or changing its behavior. An actor system is supposed to dynamically allocate resources for pending computation tasks and release them once tasks are completed. In contrast to a sequential program, which is limited to running on a single core and executing one instruction at a time, the actor model allows for concurrent execution across multiple cores. The creation of new actors naturally increases the distribution or the parallelism level of the computation and effectively drives higher usage of resources. Meanwhile, idle actors, those with no messages to process, consume no computing power and do not hinder the system's efficiency and scalability. Under the rapid development of multi-threaded and many-core systems [182, 179, 22], the actor model becomes more appealing as a programming abstraction that can efficiently leverage computing resources.

2.3.2 Actor-Based Stateful Middle Tier

This section summarizes the advantages of the actor model that make it an ideal choice for constructing stateful middle tiers [26, 30, 155] for modern applications.

From the perspective of developers, the actor model resembles the OOP programming paradigm in terms of encapsulation and modularity, offering an intuitive way to structure applications by modeling them as actors. Besides, the actor model naturally partitions the application state across many actors, eliminating the need for developers to explicitly design the application for a distributed architecture. In addition, each actor's state can be flexibly organized by using different data types and structures without the constraints of conforming to database schemes. Meanwhile, transactions can be coded as workflows that pass through one or multiple actors, enabling the implementation of complex transaction logic without the need for stored procedures. Moreover, the actor model simplifies concurrency management, alleviating the burden on developers.

From the perspective of system performance, actors are able to serve interactive requests by operating directly on the private states residing in the middle tier. Besides, the actor model can efficiently utilize the computing resources of the middle-tier servers, therefore achieving high concurrency and throughput. In addition, the system is easy to scale by adding more actors and distributing them across multiple servers. Furthermore, the failure of one actor does not directly affect others, benefiting from the loosely coupled messaging mechanism.

2.3.3 Actor Runtime and Framework

The adoption of the actor model in Erlang [59] marked a significant milestone in the actor model's history. Erlang was developed in the 1980s by Ericsson [12] to build highly reliable, concurrent, and distributed systems, particularly for telecommunications. In such applications, signals like voice, data, and video generated by thousands or even millions of concurrent users must be transmitted swiftly and reliably over long distances across networks. The actor model provided a solid conceptual foundation for building applications alike to easily handle high concurrency, fault tolerance, and large-scale distributed computing. Since Erlang successfully demonstrated the power of the actor model, plenty of actor-based programming languages, libraries, runtimes, and frameworks have emerged (listed below) in the past decades, making the actor model available for a broader range of use cases [160, 132, 60, 169, 40, 126, 2, 70].

- **programming languages**: Erlang [59], Pony [41], Elixir [58]
- **libraries**: Kilim [162], Pykka [135], Pyactor [134], Thespian [170]
- **runtimes and frameworks**: Erlang/OTP [62], Akka [5], Orleans [122], CAF [39], Pulsar [133], Orbit [118], Actor4J [23], Proto.Actor [131], Ray [112], Actix [2], Riker [147], Dapr [46]

An actor runtime manages the low-level infrastructure and focuses on system-level tasks such as executing actor code while ensuring isolation between actors, scheduling tasks according to the actor's concurrency model, and delivering messages that are immutable and asynchronous. An actor framework offers tools and patterns (e.g., libraries) to help developers structure and build actor-based applications. Actor frameworks often rely on the underlying runtimes to achieve all the critical features of the actor model. Existing actor runtimes and frameworks make it easier for developers to focus on developing the application logic rather than low-level system management.

2.3.4 Orleans: Virtual Actor Framework

Conventionally, actor frameworks, such as Erlang/OTP [62] and Akka [5], provide developers with constructs to explicitly manage the actor lifecycle, actor placement, and actors' failure recovery. While these abstractions offer greater flexibility for customizing the management of actors, they also introduce additional complexity into the development process. To free developers from these complexities, Microsoft Research introduced a new actor framework, Orleans [38, 29, 28], that invented virtual actors, namely **grains**, to automate many actor management tasks and offers a simpler and more programmable abstraction for developers. The novel features of grains are summarized as follows:

- **Automatic Instantiation**: In Orleans, a grain is automatically instantiated by the runtime when it first time receives a message. Besides, failed or garbage-collected grains are also automatically re-activated upon receiving messages. Orleans implements this functionality by retrieving the physical location of a given grain reference from the internal grain placement directory. If no instance of this grain is found, a new instance is created on an available server; then, the message is forwarded to the grain's real location. By doing so, grains perpetually exist in the system, and developers do not need to explicitly create, recover, or restart grains in the application code.

- **Automatic Garbage Collection**: Orleans supports automatic garbage collection of grains that have been idle for a long time by periodically scanning all the grain instances in the whole system. If a grain does not receive messages within a configurable time period, its memory is reclaimed, and its entry is removed from the grain placement directory. This process helps Orleans optimize memory usage and reduces the overhead associated with maintaining and retrieving the grain placement directory. In addition, it also indicates that developers do not need to explicitly destroy or terminate grains.

- **Location Transparency**: In Orleans, each grain instance is identified with a globally unique grain ID, which usually consists of the grain type and an associated ID. This identifier is determined by developers and contains no information about the grain's physical location. When sending a message, a grain reference should be obtained from the runtime using the grain ID. The grain reference acts as a proxy object that assists the runtime in routing the message to the actual location. Orleans exposes a simple API for developers to acquire the grain reference. By hiding the grain locations and the details of looking up grain locations from developers, Orleans is able to manage grain distribution, load balancing, and cluster scaling in the background. In addition, Orleans supports both built-in and customized grain placement strategies, and once these strategies are configured, they work transparently from the developer's perspective.

- **Parallelizing Worker Grains**: Another notable feature of Orleans is its ability to automatically create multiple instances of one grain, enabling parallel execution to enhance throughput. These grains function as stateless or immutable workers, each capable of independently processing incoming messages without requiring coordination.

This dissertation developed a prototype system using Orleans, leveraging its high-level, easy-to-use actor-oriented programming model and the rich feature set provided by its virtual actors. The following section discusses more

details about message passing, task scheduling, and exception handling in Orleans, which are essential components for designing and implementing state management solutions proposed in the dissertation [1].

- **Asynchronous Communication**: In Orleans, grains communicate exclusively through strongly-typed asynchronous messaging, presented to developers as asynchronous RPCs [38]. These asynchronous RPCs enable grains to interact without blocking by overlapping method invocations, a key factor in achieving the high concurrency characteristic of actor systems. Orleans also allows users to explicitly wait for the result of an asynchronous call by using the keyword `await`, in a style reminiscent of promises [96]. Specifically, sending a message to a grain is programmed as a method invocation on the grain reference. Whenever a grain method is invoked, a *promise* object is created and returned to the caller grain immediately, allowing the execution of subsequent instructions to continue without blocking. If the method call is explicitly awaited, the caller grain's execution pauses until the result of this call is assigned to the *promise*, after which the caller grain resumes execution.

- **Message Delivery**: In Orleans, message delivery timing and order are non-deterministic, i.e., messages can arrive at a destination grain at any time and in a different order than the sending order. In addition, the default message delivery guarantee is at-most-once, meaning each message being delivered once or not at all. Other than that, Orleans allows configuration of retry policies to achieve at-most-once delivery.

- **Grain Task Scheduling**: Orleans implements turn-based scheduling [38] on each grain, which processes a request as discrete units of work called *turns*. Turns are sequentially executed. By default, turns from different requests are not allowed to interleave on the same grain activation. However, Orleans provides a mechanism called *reentrancy* [123], where a grain can switch to process the turn of another request while one request is blocked by an asynchronous operation. Note that reentrancy does not compromise the grain's single-threaded execution model, as only one request can actively execute at a time even with reentrancy enabled. Reentrancy allows multiple requests to be interleaved on the same grain, which brings even higher concurrency with asynchronous RPCs. Reentracy also helps reduce deadlocks when multiple grains forward messages to each other while awaiting responses. However, reentrancy introduces additional complexity and requires careful management.

- **Exception Handling**: In Orleans, grain failures are reported as exceptions thrown from grain codes or the runtime. Orleans propagates an

[1] The discussion extracts some content of a previously published paper [99].

exception along the call chain to the caller grain. The caller grain can handle the exception using the standard `try-catch` block around the `await` statement.

2.4 Actor-Oriented Databases

The concept, Actor-Oriented Database (AODB) was first introduced by Bernstein et al. in 2017 [30], advocating the integration of database features into actor systems. It is motivated by the characteristics of actor-based applications, including: (1) Each actor maintains its own in-memory state, which can be queried and updated, functioning similarly to a small, isolated database. (2) Many actor frameworks offer the ability to persist actor state to external storage, creating the effect of seamless state persistence across failures. (3) The simplified concurrency model of actors mirrors the concurrency control mechanisms adopted in database systems. (4) Actors can be distributed across multiple servers, effectively partitioning both computation and state, much like data partitioning in distributed database systems.

Although actor systems provide developers with the illusion of interacting with database systems, they lack many essential features typically supported by traditional databases. To enhance the capability of actor systems and liberate developers from doing state management with middle-tier ad-hoc solutions, the introduction of AODB is crucial. Stet-of-the-art research works have put efforts into introducing ACID transactions (Section 2.5.1), querying functionalities, built-in indexing, actor replication, and migration to actor systems.

2.4.1 Multi-Actor Transactions

The actor model simplifies concurrency by relieving developers from manually handling it within individual actors. However, when processing concurrent requests, each spanning multiple actors, coordination among these actors becomes necessary, similar to handling distributed transactions. Moreover, the complexity increases with the introduction of reentrant actors (Section 2.3.4), where different transactions enter one actor concurrently. In these cases, supporting ACID transactional properties at the system level for multi-actor transactions is essential [149].

Though transaction processing is one of the fundamental features that has been widely adopted in many commercial database systems and investigated in plenty of research works, migrating it to an actor system remains a topic that needs further discussion and exploration. Among existing actor frameworks, only few of them have limited support for multi-actor transactions. Here we look into the three most representative ones, Erlang/OTP, Akka, and Orleans.

In Erlang, there is no native concept of ensuring that a request spanning multiple processes (or actors) is consistently committed across all participants.

Instead, Erlang performs multi-node operations using a backend database system, Mnesia [106]. Mnesia synchronously propagates an update made by a process on one node to all other replica nodes and ensures that changes are committed across these nodes atomically. However, this approach does not comply with the idea of maintaining the actor state in the middle tier and it does not address the scenarios of multi-actor transactions, where different data modifications could happen on different actors.

Akka initially adopted transactors [7], a type of actor that relies on STM (Software Transactional Memory [166]) to atomically perform a series of operations within one actor. However, this mechanism does not support operations distributing across different actors. In addition, this feature was deprecated around 2017 [6] because it introduced high performance overhead and it was counter-intuitive to share the memory space in an actor system. In recent years, researchers gradually move their attention back to the support of multi-actor transactions on Akka. For example, [153] introduced `Functor`s to Akka, which acts as a coordinator to orchestrate operations on multiple actors. Although, in principle, distributed transaction protocols can be implemented using `Functor`s, no concrete implementation was done in this work. Another work [158] adopted 2PC across Akka actors with an optimization to release lock earlier by checking some application-level information (domain knowledge).

Orleans introduced OrleansTransaction in 2016 [55], and launched a new version later in 2024 [56], which includes an improved programming model and a new transaction execution technique to enhance performance. In OrleansTransaction, each actor specifies the state object that could be involved in transactions and perform transactional operations. Each actor should also mark the methods that initiate a new transaction or join an existing one when invoked. With this programming abstraction, developers are allowed to selectively enable transactional capabilities as needed.

Besides, from the technical aspect, OrleansTransaction adopts the traditional 2PL + 2PC pattern, where 2PL guarantees the serializable isolation level and 2PC ensures atomic commit. In addition, deadlocks are detected by a timeout mechanism. To increase concurrency and alleviate the increased contention caused by the slow 2PC process, OrleansTransaction applies an optimization, ELR (Early Lock Release [13]), where each actor releases write locks right after processing the `Prepare` message, instead of waiting for the final `Commit` message. With ELR, uncommitted actor states become visible to other transactions once locks are released; therefore, a transaction can only be committed after all its dependent transactions, those that expose their dirty data to the current one, have been committed. This method requires Orleans to track a chain of transaction dependencies and is vulnerable to cascading abort.

Since the original version of OrleansTransaction has been reported with poor performance, especially under high contention [99, 34, 36], the new ver-

sion introduces reconnaissance queries [172]. This approach first runs a transaction under the read-only mode to pre-activate the involved actors; afterward, the transaction is run again by applying ordered locking based on the conjectured read/write set to reduce the occurrence of deadlocks. This approach is highly inspired by OLLP (Optimistic Lock Location Prediction), a technique introduced in [171] for addressing non-determinism for deterministic database transactions.

2.4.2 Indexing and Querying Actors

Indexing is another fundamental feature of database systems, enabling efficient querying based on specific predicates and ensuring global uniqueness when necessary. Due to the un-shared states, the original actor model presents its unsuitability for performing "bulk operations on large sets of entities" or querying "a different combination of entities" [28]. Nonetheless, it might be a potential need to query actors when taking the actor system as an AODB.

Orleans is a pioneer in exploring the possibility of building index for actors [30]. In this work, each actor is supposed to specify indexable attributes in its private state, and an index handler is responsible for propagating the updates on these attributes to the actual index structure. The particular challenges it encounters include: (1) designing a comprehensible programming model that can easily define indexable attributes and write queries, (2) providing a stronger guarantee than eventual consistency when updating index, e.g. leveraging transactions, (3) efficiently maintaining index in the distributed environment, and (4) gracefully combining the index functionality with the virtual actor model, e.g. avoid triggering the activation of actors while retrieving them.

Another work [177], inspired by Orleans, built R-tree index for spatially distributed and IoT-oriented actors. Similarly, an important trade-off in this work is to select between freshness (fast but inconsistent) or snapshot (consistent but slow) semantics for index updating.

In contrast to these two approaches, ReactDB [155] proposed a novel concept, reactor, an actor that encapsulates one or multiple partitions of relational tables as its state. Under this model, each reactor supports declarative queries on its own state. Although ReactDB introduces relational database schemes and queries into the actor model, its query ability is limited on a single actor.

2.4.3 Actor Replication

Data replication [101, 45, 102, 91] in database systems is often adopted for improving performance and enhancing availability. Data replicated across sites are able to serve highly concurrent read requests with lower latency, and the replica can be used as a backup in the presence of failures. Similarly, replicating actors can be adopted for the same purposes. However, a com-

mon challenge for doing replication is to serve write requests while ensuring consistency of replicas.

Orleans proposed GEO [27], a geo-distributed actor system that supports replicating actors across regions where round-trips between regions are significantly slow. GEO exposes APIs for developers to define different types of actors, volatile or persistent, and interact with versioned actor state. Specifically, read and write operations are performed under linearizable consistency model. GEO particularly focuses on the protocols for performing write operations on the local copy and propagating updates to replicas. The outcome of this research work has been deployed in a recent release of Orleans, where JournaledGrain is introduced. JournaledGrain is a special type of grain whose state is formed via event sourcing [121]. Additionally, a JournaledGrain can have multiple instances, and Orleans provides built-in consensus protocol to ensure replicas agree on the same sequence of events.

Similarly, Akka also adopts event sourcing to persist actor state. In Akka, an actor can be replicated by replicating events under the eventual consistency guarantee. Different from Orleans, Akka resolves conflicts between events raised from different replicas by performing commutative operations on a special data structure [9], Conflict-free Replicated Data Type (CRDT [156]).

2.4.4 Actor Migration

Migrating actors between servers is a critical feature for handling load balancing, data locality, elastic scaling, and sever failures. Relevant techniques mostly explore actor partitioning strategies, i.e. determining when and where to migrate actors.

For example, ActOp [115] extends Orleans by introducing a distributed graph partitioning algorithm that dynamically moves actors between servers in order to minimize cross-server communications. AEON [150] adopts a simple approach to monitor resource utilization of each server and migrate actors from an overloaded server to under-loaded ones. In addition, existing actor frameworks have built-in algorithms to determine actor locations. Akka partition actors into several shards, each allocated to a node by a re-balance algorithm [8]. Orleans supports several actor placement strategies such as activation-count-based placement and resource-optimized placement [120].

2.4.5 Conclusion

This section summarizes common challenges of integrating database features into actor systems. (1) A carefully designed programming abstraction is necessary to expose more information of the application to the actor system, which is critical for enabling system-level support for transactions, indices, replications, etc. (2) A well defined data model is required for each actor to enable controlled operations on its private state, such as simple read/write op-

erations and SQL-like queries. (3) It is non-trivial to combine these database features with the actor model without compromising its encapsulation, efficiency, scalability, and fault tolerance. (4) The initial proposal of AODB on Orleans [30, 25] emphasizes the independent support of different features which can be plugged in and out dynamically. However, this overlooks the potential benefits of combining different features to achieve better performance or stronger guarantees. A simple example is to enable actor replication with transactional properties which enhances consistency. This dissertation particularly investigate these aspects, and presents a comprehensive solution for scalable and transactional state management within AODB.

2.5 Transactional Database Management Systems

This section reviews some key techniques in transactional database systems that inspire us while building scalable and transactional AODB.

2.5.1 Transaction Management

In a database system, a transaction is a group of read/write operations that are executed as a single logical unit of work. In a well-managed transactional database, the ACID (Atomicity, Consistency, Isolation, Durability) properties [31] are upheld. Concurrency control [75] is a method to execute transactions concurrently and guarantee a certain level of isolation. A strict isolation level, conflict serializability [31], is fulfilled if the transaction execution schedule is equivalent to some serial schedule with the same precedence relations of respective conflict operations. A pair of operations conflict if they belong to different transactions, access the same data item, and one of them performs write.

2.5.1.1 Concurrency Control

2PL (Two-Phase Locking) is one of the widely adopted locking-based concurrency control protocols. By definition, 2PL consists of a growing phase where a transaction can acquire locks and a shrinking phase where a transaction starts to release locks, and no more locks should be acquired during this phase. There are four variants of 2PL [31], including whether the growing phase is conservative and whether the shrinking phase is strict. The conservative 2PL acquires all needed locks as an atomic operation at the beginning of the transaction, which eliminates deadlocks. The strict 2PL releases all locks at the end of the transaction, which prevents cascading rollbacks. Among these four variants, the non-conservative and strict 2PL, or, in short, S2PL, is the most commonly adopted one. A promising property of S2PL is that it guarantees a conflict serializability [31]. To deal with the deadlock issue of S2PL, deadlock prevention [148] or detection [116] mechanisms can be applied.

2.5.1.2 Tow-Phase Commit

2PC (Tow-Phase Commit) is a distributed algorithm for atomically and durably committing a distributed transaction. It ensures that a transaction involving multiple distributed components either commits (makes all changes permanent) or aborts (rolls back changes) consistently across all participants. 2PC consists of a prepare phase and a commit/abort phase. The prepare phase conducts three tasks: (1) It notifies each participant about the completion of the transaction. (2) It collects votes from all participants about whether they tend to commit or abort the transaction. (3) It ensures the participant who votes "commit" persists the changes made by the transaction; therefore, this vote stays valid even after node failure. The commit phase does three things: (1) It gathers all the votes to make the final decision. (2) It logs the final decision. (3) It informs all the participants. The two phases of communications guarantee the correctness of the commit protocol even in the presence of network or node failures.

However, 2PC is expensive because it takes a long time to conduct the two round-trips and two blocking IOs. It has been identified as one of the primary bottlenecks for scaling a distributed system [72, 77, 93, 102]. Many research works have explored various optimizations for 2PC. For example, [102, 51] commit transactions in epochs or batches to amortize overhead, [13, 73, 57, 85] shorten the lock holding time to allow higher concurrency, [184, 22] take advantage of Remote Direct Memory Access (RDMA) to reduce network latency. In addition to optimizing 2PC, there are also plenty of works putting effort into reducing the number of distributed transactions by data replication [82], data re-partitioning [93, 47, 95], or eliminating the need to do 2PC by performing deterministic transaction execution (Section 2.5.2) or compromising to weaker semantics [21, 20].

2.5.2 Deterministic Databases

A deterministic database [171] is a specialized type of database that leverages determinism in transactions to achieve several desirable properties. Here, we list three key techniques in a deterministic database and their corresponding assumptions:

- **Deterministic concurrency control**: This method guarantees that the concurrent execution of a group of transactions is equivalent to a pre-determined serial execution of these transactions. Therefore, no transaction will be aborted because of nondeterministic conflicts. To fulfill the pre-determined order, instead of processing these transactions one after the other, which allows no concurrency, the system adds an assumption to maximize concurrency.

  A1: The read/write set of each transaction is known in advance.

With this detailed data access information of each transaction, the system can apply ordered locking where locks are granted to transactions in a pre-determined order or form a dependency graph that guides the progress of each transaction. These methods are referred to as deterministic concurrency control.

- **Simplified commit protocol**: The assumption **A1** avoids deadlocks and transaction aborts caused by conflicts. The deterministic database also introduces another assumption to eliminate the consequence of node failures.

  A2: The transaction logic is deterministic.

  This assumption describes a type of transaction that can be replayed to generate the same results given the same input. In this sense, the database state can always be recovered by replaying the same ordered sequence of transactions. In other words, a failed node can always restore to the state that is right before the failure happens. This is equivalent to the situation in which no node failure exists. Therefore, 2PC can be simplified as one phase because it only needs to be determined if each participant decides to commit or abort the transaction according to the transaction logic.

- **State machine replication**: Combining **A1** and **A2**, a deterministic database can be efficiently replicated by forming each replica as a state machine that performs deterministic concurrency control for pre-ordered transactions. This method requires replication of the input sequence of transactions rather than the data stored in the database. This method can largely reduce the amount of data transferred across networks, easily maintain the consistency of each replica, and quickly build up a replica to replace a failed node immediately.

In summary, a deterministic database adopts deterministic concurrency control, simplifies distributed commit protocol, and enables state machine replication. With these techniques, a deterministic database can achieve serializability for concurrent transactions, high throughput and scalability for distributed transactions, and high availability in case of failures.

2.5.2.1 Calvin

Calvin [172] is a classic deterministic database that provides a concrete design of the system architecture and implementation. In Calvin, the system has multiple replicas. A replica consists of multiple nodes, each hosting a partition of data. Within each node, there is a sequencer responsible for receiving and ordering transaction requests from the network, a scheduler that collects schedules from all sequencers within the replica and conducts deterministic concurrency control for all transactions, and a storage layer for persistence.

Every scheduler in Calvin can consistently rebuild the global order of all transactions received by the whole system by running a consensus algorithm among all nodes or by applying a deterministic ordering strategy such as round-robin. During the execution phase, each node first fetches necessary data from other partitions within the replica and then executes the same sequence of transactions locally and deterministically. With the data prefetching phase enabled by the assumption **A1**, all transactions in Calvin are executed on a single node. With the assumption **A2**, each node can converge to a consistent state without cross-node coordination. In Calvin, each node performs like a state machine, except that only data hosted in that partition are persisted.

In addition, to deal with transactions that can not provide their read/write sets, Calvin adopts the OLLP [171] scheme, where a transaction is first run as a read-only reconnaissance query for the system to discover its read/write set. Afterward, the transaction is run again as a regular read-write transaction. However, if the conjectured read/write set does not match the actual one, the transaction will be aborted and re-tried.

SLOG [145] is a geo-replicated deterministic database that extends from Calvin. In SLOG, each replica is located in a different region, and each data item selects one region as its home region. Therefore, transactions in SLOG can access data mastered in one or multiple regions, namely single-home and multi-home transactions, respectively. SLOG adopts the hierarchical ordering strategy to order multi-home transactions using a global ordering service and single-home transactions within the corresponding region. SLOG mainly focuses on addressing the challenges of replicating transaction schedules across regions and ensuring each region converges to a consistent order for all transactions. After the order is determined, each node proceeds the same way as in Calvin.

Detock [116] is a geo-replicated deterministic database that shares a similar architectural approach as SLOG, where each data item is mastered at a home region. The primary difference between Detock and SLOG is that Detock provides a novel approach to ordering multi-home transactions. In Detock, each region independently generates an order for all multi-home transactions, and this order is replicated to other regions. To resolve the inconsistency of the different orders, Detock adopts a graph analysis method to ensure each region deterministically resolves the inconsistency and results in the same final transaction dependency graph.

2.5.2.2 Other Deterministic Databases

Existing deterministic databases have adopted different implementations of deterministic transaction processing. First of all, they make different assumptions about the determinism of transactions. Most assume a transaction's read/write set is known before its execution, and its transaction logic is de-

terministic so that the transaction can be replayed to obtain the same effect [172, 181, 137, 43]. Some only require the write set [139, 63], and others require a more detailed transaction workflow [64]. Second, based on different assumptions, they vary in concurrency control methods [146], `Calvin` [172] applies ordered locking, `Bohm` [63] and `Caracal` [139] employs multi-versioning, `Granola` [43] adopts a timestamp-based method, `Aria` [100], `DOCC` [52] and `Sparkle` [91] use optimistic concurrency control, `PWV` [64] and `T-Part` [181] rely on analyzing transaction dependency graphs. Third, they vary in thread models. `Calvin` and `T-Part` use a single thread to execute a transaction, while `Granola`, `VoltDB` [165], `QueCC` [138] and `Q-Store` [137] decompose each transaction into transaction pieces, one per partition of data, and then use one thread to execute ordered transaction pieces on each partition. Fourth, they also vary in data-sharing methods, `Bohm`, `Caracal` and `VoltDB` assume threads on different partitions have shared memory, which requires extra efforts to enable a distributed deployment. `Calvin` sends data from one node to another through a data-sharing phase before executing the transaction. `Q-Store` triggers data being sent to another node when finishing the read operation on a node [136]. `T-Part` voluntarily delivers data across nodes from one transaction to another based on the pre-generated transaction dependency graph.

Chapter 3

Snapper: A Transaction Library On Actor Systems

THE actor model has been widely adopted in building stateful middle-tiers for large-scale interactive applications, where ACID transactions are useful to ensure application correctness. In this paper, we present `Snapper`, a new transaction library on top of Orleans, a popular actor system. `Snapper` exploits the characteristics of actor-oriented programming to improve the performance of multi-actor transactions by employing deterministic transaction execution, where pre-declared actor access information is used to generate deterministic execution schedules. The deterministic execution can potentially improve transaction throughput significantly, especially with a high contention level. Besides, `Snapper` can also execute actor transactions using conventional nondeterministic strategies, including S2PL, to account for scenarios where actor access information cannot be pre-declared. A salient feature of `Snapper` is the ability to execute concurrent hybrid workloads, where some transactions are executed deterministically while others are executed non-deterministically. This novel hybrid execution is able to take advantage of deterministic execution while being able to account for nondeterministic workloads.

Our experimental results on two benchmarks show that deterministic execution can achieve up to 2x higher throughput than nondeterministic execution under a skewed workload. Additionally, the hybrid execution strategy can achieve a throughput that is close to deterministic execution when there is only a small percentage of nondeterministic transactions running in the system.

3.1 Introduction

The actor model [4] is emerging as a promising concurrent and parallel programming abstraction for building stateful middle-tiers [29, 26] in large-scale interactive applications, including multi-player games such as Halo 4 [160] and

League of Legends [132], telecommunication such as Ericsson [61], E-commerce such as Paypal [169] and Walmart [40], and Internet of Things [126]. There are plenty of programming languages [59, 58] as well as libraries and frameworks [5, 122, 118] that enable actor-based programming. With the actor model, applications are decomposed into concurrent actors, each encapsulating a private state and communicating with other actors via asynchronous message passing.

In the actor model, each actor processes its incoming messages sequentially. Such sequential behaviour frees developers from handling concurrency issues within each actor. However, there are situations where concurrent cross-actor operations require transactional properties. For example, in an online multiplayer game, player actors may exchange game equipment or purchase equipment with digital currencies. As another example, in an e-commerce application, actors maintaining product stocks and those responsible for order checkouts have to interact to complete a purchase transaction. Transactional properties are often needed to ensure application correctness in these scenarios.

Transaction management in actor-based applications is complicated by their design as stateful middle tiers, which react to changes of states in real-time and asynchronously flush accumulated states to database tiers [26]. In particular, with this architecture, transactions are executed within the middle-tier servers rather than as stored procedures in databases. Motivations of this trend include, among others, the flexibility of encoding transaction logic using programming abstractions different from database systems and being able to use the large memory and computing power of cheap middle-tier servers to manipulate data and execute transactions instead of using more expensive [15, 16] database servers [26].

To meet these new requirements and to alleviate the burden on developers, there exist efforts in various actor systems providing high-level programming abstractions for efficient multi-actor transactions while hiding their implementation complexities from developers. Akka introduced the concept of transactors [7], which employs two-phase commit (2PC) and software transaction memory (STM) to support atomic cross-actor transactions.[1] Orleans [122] has recently made efforts to support distributed transactions [55] across multiple actors. It adopts two-phase locking (2PL) and 2PC with early lock release [14, 159], allowing for higher concurrency at the price of cascading aborts.

One way to enable actor transactions is to implement transactions on top of the actor abstraction itself without any modification to the actor runtime. For example, Orleans Transactions [55] adopt such an approach. This non-intrusive approach requires less system development and maintenance effort

[1] When this feature was deprecated in 2014 [6], developers have significantly complained about its absence [175, 49].

in comparison to alternatives with deep integration with the actor runtime. Therefore, we focus on this approach due to its low development cost. However, under this approach, multi-actor transactions are challenging. This is because the state of an actor-based application is partitioned into many fine-grained actor private states. Every multi-actor transaction, no matter if the actors are collocated on the same machine or not, is a cross-partition transaction and has to employ distributed transaction mechanisms, which are expensive.

In this paper, we argue that the existing actor transaction mechanisms have not sufficiently exploited the characteristics of actor-oriented applications to improve transaction performance, particularly transaction throughput. An interesting characteristic is that actors are accessed explicitly in an actor programming abstraction, e.g., by the names or process IDs in Erlang [59], by the paths of the actor hierarchy in Akka [5], and by the user-defined actor identities in Orleans [122]. It is often the case that the set of actors involved and the number of times that they would be accessed in a transaction are known before the transaction starts. For example, in an e-commerce system, a `CheckoutOrder` transaction explicitly specifies a list of product IDs, which targets a list of stock actors, each being accessed once. Another example is, in a social network application, when a user issues a `JoinGroup` transaction, which updates the membership data in a determined user actor and group actor, each being accessed once, respectively.

This characteristic of actor programming enables the exploration of a novel actor-based transaction abstraction, where the identities of the participating actors and the number of times that they are accessed in a transaction are pre-declared. With such apriori information, an actor system would be able to pre-schedule the transactions and execute them in a deterministic order. In comparison to nondeterministic concurrency control methods adopted by existing approaches, such as 2PL, a deterministic ordering strategy would avoid transaction aborts due to conflicts [172, 146]. The latter holds the potential to significantly improve system throughput, especially when the contention level is high. Furthermore, 2PC can be optimized to enhance transaction concurrency [172, 146].

Despite that we envision most actor transactions in an actor-based application can be implemented with the aforementioned transaction abstraction, there could exist transactions that do not fall into this category. For example, in a social network application, a user could issue a `CleanUpFriendList` request, which removes friends who are in the user's friend list but with no recent interactions and would then trigger the removed friends to also update their friend lists. Such a transaction may need to look up a user's friend list and the recent interaction histories to determine the set of actors that would be involved. In other words, the list of participating actors of the transaction may not be known before the transaction starts. Therefore, one may have to resort to conventional actor transaction abstractions based on

non-deterministic concurrency control and 2PC, such as Akka transactor or Orleans Transactions, to execute this type of transaction. Supporting both types of abstractions in the same system is a challenge. Deterministic and nondeterministic concurrency control methods achieve serializability based on different principles, and how to reconcile these two methods in a single system is still an open problem.

In this paper, we propose `Snapper`, an actor transaction library on top of Orleans that enables multi-actor transactions. Our goal with `Snapper` is to improve the performance of cross-actor transactions based on the fact that existing solutions such as Orleans Transaction do not perform so well, especially under high contention, and on a significant observation that deterministic transaction execution is well-suited to the actor model. Specifically, `Snapper` supports two types of actor transaction abstractions, namely Predeclared ACtor Transaction (PACT) and ACtor Transaction (ACT), which employ deterministic and nondeterministic concurrency control mechanisms, respectively. A salient feature of `Snapper` is that it supports a novel hybrid execution strategy that enables concurrent execution of transactions specified using different actor transaction abstractions. As the first cut at the problem, this paper focuses on optimizing and evaluating the performance of single-server transactions, i.e., transactions that only involve actors located on the same server. We focus on this problem because we envision that, in order to maximize transaction performance, the allocation of actors should be optimized so that the majority of transactions are single-server transactions as in high-performance OLTP database systems [45, 124]. Besides, optimizing single-server transactions can be of great value to many applications that are able to scale vertically and are suited to exploiting locality. In summary, the main contributions of this paper include:

- We propose a novel programming abstraction for multi-actor transactions, namely PACT, which enables deterministic execution of multi-actor transactions in an actor system.
- We propose a hybrid transaction execution method that enables concurrent execution of PACTs and ACTs. To the best of our knowledge, we are the first to study how to accommodate both deterministic and nondeterministic transaction execution strategies in a single system.
- To verify practicability, we implement the proposed actor transaction abstractions and execution strategies as a library on top of Orleans, a widely adopted actor system.
- We conduct a series of experiments to evaluate the effectiveness of our transaction execution methods using `SmallBank` and `TPC-C` benchmarks. The results show that compared to ACTs and Orleans Transactions, PACTs can achieve up to 2x higher throughput. Additionally, the hybrid execution can achieve a throughput that is close to PACTs when the percentage of ACTs is small.

3.2 Snapper Programming Model

3.2.1 Conceptual Overview

`Snapper` is a library that supports executing transactions involving method calls over one or more actors. It provides transactional APIs to access the state of the current actor and to invoke method calls on other actors. These APIs are implemented in `TransactionalActor`, which is a base class of actor and has system functionality such as persisting logs and committing/aborting transactions built on top of it. To get transactional guarantees provided by `Snapper`, user-defined actors must extend `TransactionalActor` and use its APIs when accessing actor state and invoking method calls.

In `Snapper`, each transaction is a series of method invocations performed on multiple actors. A transaction is initiated by one actor where the first method is invoked. This actor will start executing the transaction and invoke methods on other actors via asynchronous RPCs. A transaction can invoke methods on the same actor multiple times. The actor that initiates the transaction will also end the transaction when the first invoked method has finished. This actor is responsible for committing/aborting the transaction, thus there is no need to explicitly issue such requests in application codes. `Snapper` guarantees conflict serializability for all concurrent transactions and provides built-in durability for `TransactionalActor`. A transaction can run under one of the following two modes:

• **Pre-declared ACtor Transaction (PACT):** Transactions in this mode must pre-declare the following information: (1) the actor that will initiate the transaction, (2) the first method that will be invoked and the corresponding input data for this method, and (3) `actorAccessInfo` – the set of actors the transaction will access and the number of times each of such actors will be accessed. Based on the pre-declared `actorAccessInfo`, `Snapper` performs deterministic scheduling for PACTs, and each accessed actor will execute PACTs in the pre-determined order. Besides, each `TransactionalActor` has reentrancy enabled to schedule transactional method invocations since they may not arrive in order. Under the pre-scheduling strategy, `Snapper` guarantees no PACTs abort due to concurrency conflicts. However, users are allowed to explicitly abort a PACT by throwing an exception to `Snapper`.

• **ACtor Transaction (ACT):** Transactions in this mode only need to declare information (1) and (2) mentioned above. The actors accessed by an ACT are discovered when methods are invoked during the ACT's execution. `Snapper` applies a conventional nondeterministic concurrency control, e.g., S2PL, for ACTs. Unlike PACTs, ACTs can be aborted due to deadlocks or read/write conflicts.

3.2.2 Transactional API of Snapper

Snapper exposes three APIs, StartTxn, CallActor and GetState. Fig.3.1 gives the definition of each API; Fig.3.2 and 3.3 give an example of how to use those APIs.

PACT	`Task<object> StartTxn(string startFunc, object funcInput, Dictionary<Guid, int> actorAccessInfo)`
ACT	`Task<object> StartTxn(string startFunc, object funcInput)`
both	`Task<object> GetState(TxnContext cxt, AccessMode mode)`
	`Task<object> CallActor(TxnContext cxt, Guid actorID, FuncCall call)`

Figure 3.1: Snapper's transactional API

3.2.2.1 Submitting Transactions to Snapper

Fig.3.2 shows how clients submit transactions to Snapper. A client submits a transaction by calling StartTxn on the first actor that the transaction will access (lines 12, 22). The client can choose to submit a transaction as a PACT or an ACT by passing different data to StartTxn. As for the ACT mode, the name of the first method that will be invoked and the corresponding input data should be given by the client. As opposed to ACT, the PACT mode additionally requires actorAccessInfo as input. Snapper distinguishes transaction modes according to the input data. At last, StartTxn will return the transaction result (e.g., the balance after doing Transfer) as an object to the client. If the transaction is aborted in Snapper, the transaction is rolled back by Snapper, and an exception will be thrown to the client.

3.2.2.2 User-Defined Transactional Actors

Fig.3.3 shows how developers program user-defined actors using Snapper's API. AccountActor is a user-defined actor class with interface IAccountActor, same as ordinary actor definitions in Orleans. To inherit Snapper's transactional actor features, the user-defined actor interface and actor class should derive from ITransactionalActor and TransactionalActor, respectively (lines 1, 7). In addition, the type of the actor state should be explicitly declared, which can consist of primitive or user-defined types. In the example, float is the type of the AccountActor's state, which represents the balance of the account (line 7). The interface of the user-defined actor should always contain two input parameters: TxnContext and the input data for the method involved in the transaction (e.g., Deposit or Transfer). An instance of TxnContext is an internal read-only data structure of Snapper. It contains the transaction's context information such as tid, txnMode, etc. It is generated by Snapper after receiving the transaction request from the client and before executing the transaction. It is passed as a parameter in all three APIs so that Snapper

```
1  public class Client
2  {
3      static void main()
4      {
5          // ... ... ...
6          var funcInput = new Tuple<float, long>(100, toAccountID);
7          var actor = client.GetGrain<IAccountActor>(fromActorID);
8
9          try
10         {
11             // ============== submit an ACT ==============
12             var ACT_balance = await actor.StartTxn("Transfer", funcInput);
13
14             // ============== submit a PACT ==============
15
16             // prepare the actor access info
17             var info = new Dictionary<Guid, int>();
18             info.Add(fromActorID, 1);
19             info.Add(toActorID, 1);
20
21             // submit via PACT API
22             var PACT_balance = await actor.StartTxn("Transfer", funcInput, info);
23         }
24         catch (Exception e)
25         {
26             // ...
27         }
28     }
29 }
```

Figure 3.2: Submission of PACT and ACT to `Snapper`

can schedule and execute method calls transactionally based on the context information.

As for the implementation of the user-defined actor class, the method `GetState` should be used to access the actor state (lines 16, 25). `Snapper` supports two access modes: `Read`, which is read-only, and `ReadWrite`, which reads and writes the state. `CallActor` should be used to invoke method calls on other actors (line 36). Instead of directly calling another actor's method in user-defined code, `Snapper` wraps this operation in `CallActor`. It is designed this way because `Snapper` has to gather and propagate transaction execution information along with actor method calls, and `CallActor` abstracts these actions away from developers.

3.2.2.3 Aborting a Transaction

In `Snapper`, PACTs do not abort due to concurrency conflicts but can abort due to runtime exceptions or user-defined transaction logic, e.g., a `Transfer` transaction might abort because of insufficient balance. By contrast, ACTs can be aborted for all three reasons. Users can abort a PACT or an ACT by throwing an exception to `Snapper` (line 27 in Fig.3.3). `Snapper` catches both internal exceptions caused by runtime issues or concurrency conflicts and external exceptions thrown by user codes. Any exceptions that are not handled by user codes will be caught by `Snapper` and treated by aborting and

```
1  public interface IAccountActor : ITransactionalActor
2  {
3      Task Deposit(TxnContext ctx, float money);
4      Task<float> Transfer(TxnContext ctx, Tuple<float, long> input);
5  }
6
7  public class AccountActor : TransactionalActor<float>, IAccountActor
8  {
9      public AccountActor() : base(typeof(AccountActor).FullName) { }
10
11     private Guid MapAccountIDToActorID(long accountID);
12
13     public async Task Deposit(TxnContext ctx, float money)
14     {
15         // get access to the actor state via the GetState API
16         float myBalance = await GetState(ctx, AccessMode.ReadWrite);
17         myBalance += money;
18     }
19
20     public async Task<float> Transfer(TxnContext ctx, Tuple<float, long> input)
21     {
22         var money = input.Item1;
23
24         // get access to the actor state via the GetState API
25         float myBalance = await GetState(ctx, AccessMode.ReadWrite);
26
27         if (myBalance < money) throw new Exception("balance insufficient");
28
29         myBalance -= money;
30
31         var toAccountID = input.Item2;
32         var toActorID = MapAccountIDToActorID(toAccountID);
33         var funcCall = new FuncCall("Deposit", money, typeof(AccountActor));
34
35         // forward a call to another actor via the CallActor API
36         await CallActor(ctx, toActorID, funcCall);
37         return myBalance;
38     }
39 }
```

Figure 3.3: User-defined actor programs with `Snapper`'s API

rolling back the relevant transactions. Note that submission of a PACT with user-defined aborts should not be the norm because it will lead to performance degradation (Section 4.2.3). A transaction with user-defined aborts is better submitted as an ACT. `Snapper` supports the aborting of PACTs mainly for cases where unexpected runtime exceptions arise.

3.3 Single-Node Architecture

3.3.1 Overview

3.3.1.1 Components

`Snapper` consists of three components, including two types of actors – coordinators and transactional actors – and a group of loggers, which are in-memory C# objects shared by all actors on the machine and responsible for writing logs.

Coordinator actors are responsible for assigning a unique transaction identifier (`tid`) to each transaction. For PACTs, the `tid`s should be assigned according to a global serial sequence order that determines their execution order. The sequence of PACTs is divided into batches in order to amortize the overhead of messaging and logging. Coordinator actors interact amongst themselves to reach consensus on such a global sequential order for PACTs and they also coordinate transactional actors to execute and commit batches in the pre-determined order.

Transactional actors are the base actor class provided by `Snap- per` to program user-defined actors where applications' transactional states are stored. Each transactional actor schedules PACTs according to the deterministic sequential order generated by the coordinator actors and performs nondeterministic execution of ACTs. With hybrid workloads of PACTs and ACTs, transactional actors employ a novel hybrid concurrency control for them.

Loggers implement `Snapper`'s persistence functionality. They handle all logging requests sent from coordinators and transactional actors. Each logger keeps access to a log file in the storage. An actor can invoke the method call on one of the loggers, which is chosen by a simple hash function on the actor ID. A logger may be shared by multiple actors. Each task on the logger is scheduled by the actor who issues the request [122]. In comparing to each actor persisting their own logs, delegating the tasks to loggers, whose number is much smaller than the number of actors, can constrain the number of log files, reduce random IO access to storage, and amortize the IO cost by batching. Another option is to implement loggers as actors, but this would require copying data from coordinators and transactional actors to the logger actors, which is inefficient.

For simplicity, in the remainder of the paper, "coordinator actors" are always referred to as "coordinators", and "transactional actors" and "actors" are used interchangeably.

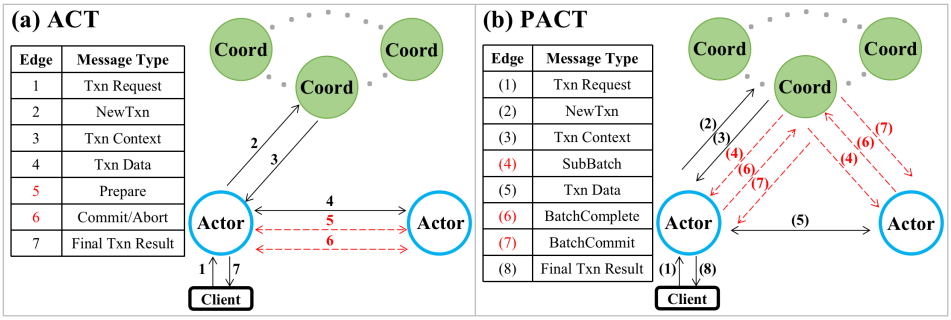


Figure 3.4: Transaction workflow

3.3.1.2 Transaction Workflow

Fig.3.4 illustrates the workflows of PACTs and ACTs. A client submits a transaction by calling the `StartTxn` API on the first actor that should be accessed by the transaction (Edge (1)). This actor then issues the `NewTxn` request to one of the coordinators, selected by a simple hash function on its own actor ID (Edge (2)). In return, the actor gets a `TxnContext` instance, which includes the `tid` assigned by the coordinator (Edge (3)). An actor may invoke method calls on other actors via the `CallActor` API to execute operations in a multi-actor transaction (Edge (5) in Fig.3.4a and (4) in Fig.3.4b). Fig.3.5 shows the data passed along with such actor method calls. After the callee actor finishes executing the operation, it returns to the caller with an instance of `ResultObj` containing data that should be returned to the caller along with transaction execution information (`TxnExeInfo`). The first actor is both the start and the endpoint of the whole workflow. The client receives the result of the transaction (Edge (8) in Fig.3.4a and (7) in Fig.3.4b) after it is either committed or aborted. Each ACT requires two round-trip messages per *transaction* (Edge (5) and (6) in Fig.3.4b) in order to perform 2PC, while each PACT requires three one-way messages per *batch* (Edge (4), (6), and (7) in Fig.3.4a) in order to control deterministic batch processing.

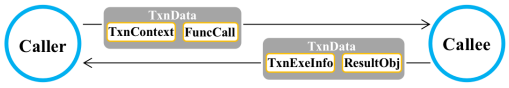


Figure 3.5: Content of `TxnData`

3.3.2 PACT Processing

3.3.2.1 Ordering

To assign deterministic execution order to PACTs, one can use a single coordinator to sequentially assign a monotonically increasing `tid` to each PACT and use the `tid` to determine the order. However, using a single-threaded coordinator may not be able to scale. Instead, `Snapper` exploits parallelism by employing multiple coordinators and each independently receiving transaction requests. To guarantee the monotonicity of `tid` while using multiple coordinators, we essentially need mutual exclusive access to the latest `tid` that has been assigned. To achieve this, `Snapper` adopts the classical token ring algorithm [168] for its simplicity and its natural match with the message-passing abstraction of actors. More specifically, coordinators are logically placed in a ring, where each coordinator has fixed left and right neighbours. A token is circulated in a particular direction in this ring. The token carries all the information that needs to be shared among coordinators, e.g., the latest assigned transaction ID (`last_tid`). A coordinator accumulates the PACTs that it has received while waiting for the token. When it receives the token,

it allocates `tid`s for those PACTs based on the `last_tid` value stored in the token, updates the `last_tid` in the token, and then passes the token onward to its neighbour. By doing so, we guarantee that the `tid` assignment is monotonically increasing across multiple coordinators. Note that the token can be forwarded to the next coordinator immediately when the new batch is formed without waiting for the batch to be emitted, executed, or committed. Thus, the token ring mechanism does not substantially increase transaction latency. Conversely, while a coordinator is waiting for the token, it can perform other tasks, such as communicating with transactional actors, coordinating batch commitment, logging, etc.

3.3.2.2 Batching

With `tid` and `actorAccessInfo` of a PACT, the coordinator can send messages to inform each accessed actor of the existence of the PACT. However, delivering one `tid` per message is inefficient. Therefore, `Snapper` chooses to deliver information about a batch of transactions per message ((4) in Fig.3.4a).

Similar to epoch-based batching [50, 44, 101], the token ring mechanism naturally generates epoch boundaries. Every time a coordinator receives the token, it puts all locally accumulated PACTs into a new batch. The size of the batch depends on the transaction rate and the time that the token takes to be passed around a cycle. Besides, each batch uses the `tid` of the first PACT in the batch as its batch ID (`bid`). As long as all actors execute the batches in the order of `bid` and execute the PACTs within each batch in the order of `tid`, the global sequential order can be guaranteed.

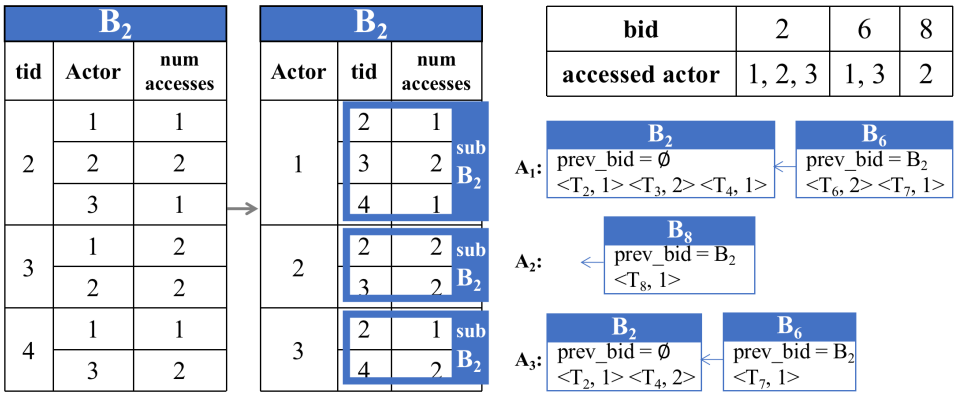


Figure 3.6: Batching

Since not every PACT will access all the actors, each actor may only need to execute a subset of PACTs submitted to the system. Given a batch, a coordinator should generate a sub-batch for each accessed actor. For example, in Fig.3.6a, three actors will be accessed by batch 2, and hence three sub-batches

are generated based on `actorAccessInfo`. A sub-batch can be delivered as one message to an actor. Besides, each sub-batch should also carry a `prev_bid` indicating its previous batch on this particular actor (Fig.3.6b). This is necessary because batches that need to be executed on an actor may not have consecutive `bid`s. Even if they do, the batch messages may arrive out of order due to nondeterministic message delays. With the `prev_bid`, an actor can know the order between batches, thus it can start executing a batch when its previous batch has completed. In `Snapper`, the `prev_bid` for each actor is stored in the token, updated by the coordinator when a new batch is created and removed if the corresponding batch has committed. After the updates of `last_tid` and `prev_bid`s, the coordinator can pass on the token and emit `BatchMsg`s to related actors.

With batching, the overhead of sending messages is amortized over multiple PACTs in a sub-batch. The efficiency of batching can increase with the skew in the workload because more PACTs will be included in one sub-batch. To reduce overhead and improve transaction throughput, `Snapper` schedules, executes and commits PACTs at the batch granularity. This accrues benefits on both the messaging ((4), (6), (7) in Fig.3.4a) and logging.

3.3.2.3 Deterministic Scheduling

Each actor maintains a local schedule to control the transaction execution order. Such a schedule is needed for two reasons. First, we cannot rely on the message arriving order to order the transactions. Second, each actor should have its own schedule because the sets of transactions to be executed on actors are different from each other. In the schedule of an actor, batches are placed in a chain according to the `prev_bid` relation. An actor gradually extends the schedule upon receiving batch messages and removes the batch when it is committed/aborted. If a batch arrives at an actor earlier than its previous batch, this batch creates a vacancy in the chain, which is filled when the previous one arrives. For example, in Fig.3.6b, on A_2, B_8 is currently maintained separately because its previous batch B_2 has not arrived yet. A batch message contains `bid`, `prev_bid` and a list of PACTs, including their `tid`s and the number of accesses on the actor.

When any method invocation of a PACT (called through the `CallActor` API) arrives on an actor, the actor will execute it according to the local schedule. More specifically, the actor first checks the carried `TxnContext` of the received call. If the call comes from a PACT whose turn is yet to come according to the local schedule, the actor will suspend the execution of this method invocation by `await`ing an asynchronous task, which is later resolved when the scheduled previous PACT has completed. A PACT is considered completed when an actor has been accessed the declared number of times. Then, the actor can resume the execution of the suspended method invocation. Notice that when an execution is blocked by an asynchronous operation,

the actor is free to process another request because `Snapper` has enabled *reentrancy* (Section 2) for all `TransactionalActor`s. In Orleans, a reentrant actor is allowed to interleave the execution of multiple requests. As for the caller actor, the invoked call is essentially an asynchronous RPC. Once the callee actor enqueues the call into its message box, a future is returned, and the corresponding promise can be fulfilled later.

Since PACTs are executed under a deterministic schedule and are guaranteed not to abort due to conflicts, a sub-batch on an actor can be speculatively executed as long as its previous sub-batches have completed their operations on this actor without waiting for them to commit. This allows for pipelined execution of batches while respecting the schedules on individual actors without waiting for coordination of batch commitment across different actors. This also brings higher concurrency compared to conventional nondeterministic concurrency control such as S2PL, where locks are only released when a transaction is committed. Besides, S2PL usually introduces non-deterministic blocking due to conflicts, while PACTs can avoid this effect because pre-scheduling is applied.

However, if a PACT is aborted, the whole batch would be rolled back along with all batches that have been speculatively executed. So, submitting a PACT that will eventually abort can cause performance degradation. Thus, a transaction with user-defined aborts is better submitted as an ACT.

3.3.2.4 Commit and Logging Protocol

`Snapper` applies a specialized two-phase commit protocol for PACTs, which logically includes three-round one-way messages, the `BatchMsg`, `BatchComplete` and `BatchCommit` messages ((4), (6) and (7) in Fig.3.4a). `BatchMsg`s are the sub-batches sent from a coordinator to participating actors, which can be analogized to the `prepare` message in 2PC. When an actor finishes executing the sub-batch, it acknowledges `BatchComplete` to the coordinator who emitted the corresponding batch, which is similar to "voting" in 2PC. When the coordinator receives `BatchComplete` from all participating actors and if all vote to commit, the coordinator can commit the batch and send the confirmation message `BatchCommit` back to the actors, which will return the final transaction results to clients for the PACTs within the batch ((8) in Fig.3.4a). The commitment of a batch must be done by coordinators because they are the only ones who know the list of participating actors in a batch. Besides, the commit protocol should guarantee that a batch B commits after all the batches it depends on – batches that have been scheduled before B on the actors that B accessed – have committed. To eliminate the overhead of maintaining the complex dependency graph between batches, `Snapper` instead tracks the logical dependency in which a batch B_i always logically depends on B_j if $i > j$. Further, `Snapper` forces all batches to commit in the order of `bid`. This strategy works well, especially under a highly contended workload where

logical dependencies reflect actual dependencies. The overhead of tracking logical dependencies between batches is negligible. Each coordinator only needs to keep track of the last assigned batch ID for the batches it generates. This batch ID is then passed along in the token.

Now, we explain the process of aborting a batch. An aborted batch is detected by the actor, who catches the exception thrown by a PACT. Recall that an aborted batch may cause cascading aborts of speculatively executed batches. To avoid unbounded numbers of batches being aborted in this process, `Snapper` stops emitting new batches whenever an aborted batch is detected and resumes when the cascading abort has been completed. The classic cascading abort will abort transactions that depend on the aborted ones [32]. Again, instead of maintaining the accurate dependencies between batches, `Snapper` simply aborts all uncommitted batches in the system.

To ensure the durability of committed PACTs and ensure the commit process survives failures, `Snapper` utilizes a Write Ahead Log (WAL) to store related data prior to sending out any messages such as `BatchMsg`, `BatchComplete`, and `BatchCommit`. Fig.3.7 shows the logs written for a batch that accessed two actors. Three types of log records should be written for a batch. (1) Before emitting a batch, the coordinator persists the participating actors of the batch. (2) Before sending `BatchComplete`, an actor logs the updated actor state. If the batch has only read the actor, there is no need to persist the actor state. (3) Before sending `BatchCommit`, the coordinator logs the committed `bid`.

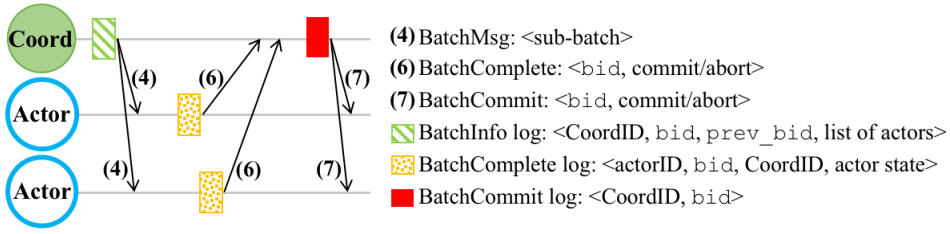


Figure 3.7: PACT Logging

Based on the logged information, `Snapper` is tolerant to failures that happen to both coordinators and actors at any time while executing, committing, or aborting a batch. In the batch commit protocol, the coordinator cannot decide to commit a batch until all participating actors have voted. If an actor fails before sending the `BatchComplete` message, the coordinator must wait until the failed actor is recovered and the message is sent. The recovered actor will retrieve its log records. If the `BatchComplete` record is not found, it will tell the coordinator to abort the batch. If a coordinator fails before sending the `BatchCommit` message, all related actors must wait to return results to clients until the coordinator is recovered and the message is sent. Those actors can autonomously ask the coordinator about the decision to commit or abort the

batch. Snapper follows the principle that the batch that has BatchComplete log records written in all participating actors can commit.

3.3.2.5 Recovery

Assume that the system can fail (crash) at any time, and some or all actors will lose their in-memory data. Snapper relies on the failure recovery mechanism provided by Orleans that a failed actor is automatically re-instantiated when it is called again. In Snapper, failed actors are re-instantiated by loading the state of the last committed batch. A failed coordinator is re-instantiated by loading the information of emitted but uncommitted batches, which needs to be used to continue the batch commit/abort process. Besides, the token may also be lost with the failed coordinator. To make sure the system has exactly one token, a recovered coordinator must trigger a consensus protocol among all other coordinators to check if the token is lost or not and elect a coordinator to re-initiate a new token if needed. If a new token needs to be used, the system must wait to emit new batches until all existing batches have committed/aborted because the prev_bids stored in the old token are lost. When all emitted batches have committed/aborted, all actors also have their local schedules empty; thus, the old prev_bid is not needed.

3.3.3 ACT Processing

3.3.3.1 Transaction ID Assignment

Unlike PACTs, ACTs need to be assigned unique transaction IDs (tids), but not a deterministic execution order. To achieve this in Snapper, every time the token is received by a coordinator, it will pre-allocate a range of contiguous tids for ACTs that may arrive in the future. Those ACTs will get tids assigned immediately without having to wait for the token.

3.3.3.2 Nondeterministic Concurrency Control

When a method invocation of an ACT arrives on an actor, the invocation is controlled by a traditional nondeterministic concurrency control protocol. Currently, we have implemented S2PL in Snapper. Multiple ACTs can invoke method calls on an actor concurrently. The S2PL protocol is executed when an actor accesses the state using the GetState API, which grants logical read-/write locks to ACTs and releases them after the second phase of 2PC. Besides, wait-die [148] is used to proactively avoid deadlocks by aborting transactions if they are suspected to be involved in a deadlock.

3.3.3.3 Commit and Logging Protocol

ACTs are committed via 2PC [90], and presumed abort [111] is used to save messages and logging. When an ACT completes its operations, all the actors

that have been accessed within the transaction context are known. The list of participating actors of an ACT is propagated as part of `TxnExeInfo` (Fig.3.4c) along the method call chain back to the first actor who initiates the ACT. This information is utilized to perform the 2PC protocol with all the participating actors. The actor where the ACT is initiated is designated as the coordinator of the 2PC protocol. While doing 2PC, logs are persisted before sending any messages. Fig.3.8 shows logs written for an ACT that spans two actors. Again, if an actor involved in the ACT did not perform any writes, there is no need to persist the state of that actor.

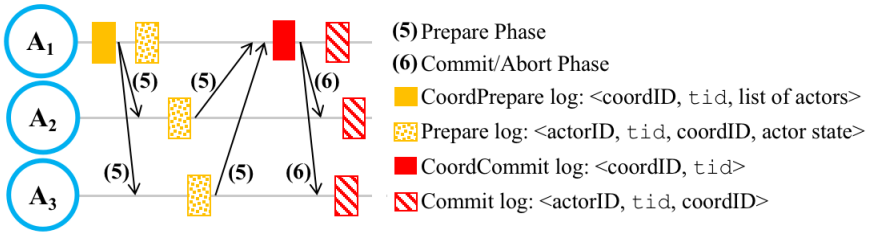


Figure 3.8: ACT Logging

3.3.3.4 Recovery

Upon failure, every actor finds the latest `CoordCommit` record and reloads the state of the latest committed ACT on all participating actors. And every actor reads the `CoordPrepare` and `Prepare` records to resume the 2PC process. Incomplete transactions will be aborted by the 2PC protocol.

3.3.4 Hybrid Processing

3.3.4.1 Hybrid Scheduling

With hybrid workloads of PACTs and ACTs, each actor's local schedule contains PACT batches in sorted order by `bid` and `tid`, and ACTs that are dynamically inserted between two adjacent batches. When receiving an ACT method invocation, the actor always appends the ACT to the tail of the current schedule. Fig.3.9 gives an example. ACT T_0 is appended after B_6 on A_1 and after B_2 on A_3. The existence of ACTs does not affect the batch order, e.g., on A_3, B_6 is still placed after its previous batch B_2 even though ACT T_0 and T_5 are scheduled in-between them.

With a hybrid schedule, actors need to carefully switch the execution between PACTs and ACTs. `Snapper` abides by the following rules: (1) an ACT can start executing when the previous batch has completed its operations but is not necessarily committed; (2) a batch can start executing when all previous ACTs have committed or aborted. By doing so, ACTs will not see the results of a half-done PACT, and a PACT will not operate on data that will be

aborted by an ACT. Thus, PACTs will still not abort due to concurrency control. Besides, multiple ACTs can be concurrently executed if they are placed between the same two batches on an actor. For example, in Fig.3.9, on A_3, T_0 and T_5 are unblocked at the same time when B_2 completes.

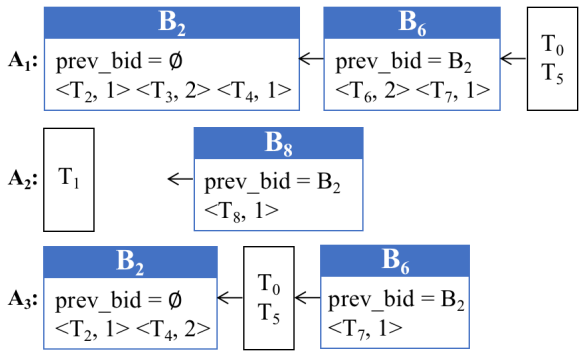


Figure 3.9: Actor local schedule

Despite the fact that these two rules nicely isolate the executions of PACTs and ACTs on each individual actor, they are unfortunately insufficient to guarantee deadlock-freedom or serializability of hybrid workloads across multiple actors, which we address next.

3.3.4.2 Deadlock

Under hybrid execution, deadlock can happen between PACTs and ACTs due to the non-deterministic scheduling of ACTs and blocking method invocations. The following two cases illustrate how such deadlocks occur: (a) an ACT T_i is scheduled before a batch B_j on one actor A_1, but scheduled after B_j on another actor A_2, and at the same time, a PACT of B_j on A_2 has to wait for A_1 to invoke the expected – by number of accesses information – method call (Fig.3.10a). (b) An ACT T_i is waiting for T_j to release the lock on an actor A_3, and T_i is scheduled before a batch B_g on A_2, T_j is scheduled after a batch B_k on A_4, and at the same time, a PACT of B_k on A_4 has to wait for A_1 to invoke the method call (Fig.3.10b). In both cases, the global waits-for graph is cyclic. In addition to the patterns shown, a deadlock can easily involve more actors and transactions. To solve such deadlocks, we need to abort one of the transactions in the cycle. Since PACTs require more information from clients and are deterministically scheduled, `Snapper` always prioritises PACTs and aborts ACTs in the case of deadlocks. In our current implementation, a simple timeout mechanism is applied to detect deadlock [32].

3.3.4.3 Serializability Check

To enforce serializability, `Snapper` employs deterministic transaction execution for PACTs and nondeterministic concurrency control for ACTs. However, un-

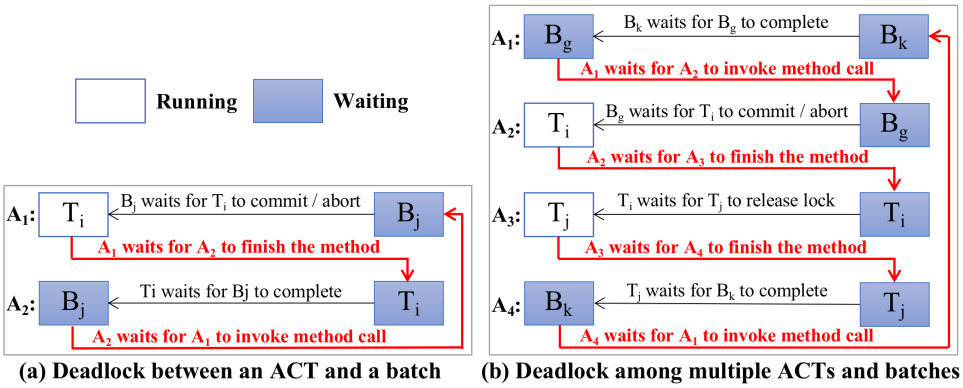


Figure 3.10: Deadlock under hybrid execution

der hybrid execution, the nondeterministic interleaving between batches and ACTs makes it challenging to achieve global serializability. Fig.3.11 illustrates two scenarios where the global serialization graph is cyclic: (a) An ACT is scheduled before and after the same batch on two different actors, respectively. (b) Each single ACT does not manifest cyclic dependencies between any other batches, but the dependencies between ACTs make the global serialization graph cyclic. Our deadlock mechanism already aborts some ACTs that break global serializability. However, there exist cases that do not form deadlocks but still break serializability. Such cases can happen when the dependency between two actors – any cross-actor edge, e.g., in Fig.3.10 – is in the opposite direction.

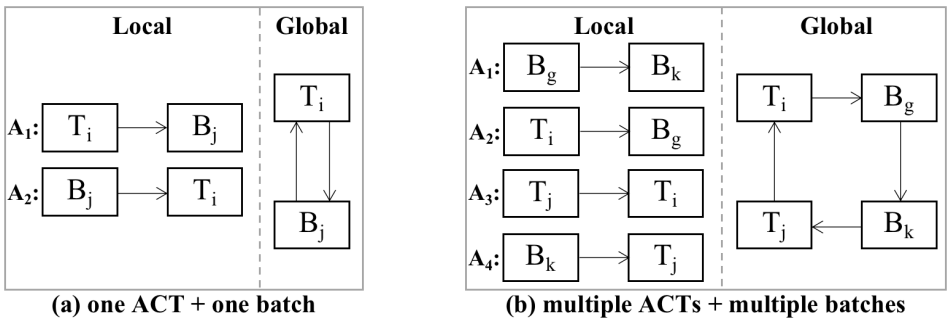


Figure 3.11: Cyclic serialization graph

Similar to the handling of deadlocks, `Snapper` enforces global serializability by choosing to abort ACTs that cause the problem. `Snapper` applies a serializability check for the ACTs that have finished execution. ACTs that fail to pass the check should thus be aborted. We rely on scheduling information defined as follows.

Definition 1 *Given a history H generated by* `Snapper`*'s hybrid processing and the corresponding serialization graph $SG(H)$, $\forall$ ACT $T_i \in SG(H)$, its* `BeforeSet` *(BS_{T_i}) and* `AfterSet` *(AS_{T_i}) are defined as:*

1. $BS_{T_i} = \{B.bid|$ *there exists a path* $B \rightarrow ... \rightarrow T_i\}$
2. $AS_{T_i} = \{B.bid|T_i \rightarrow B\}$

In addition, $max(BS_{T_i})$ and $min(AS_{T_i})$ are the maximum and minimum numbers (bids) in BS_{T_i} and AS_{T_i}, respectively.

Above, we borrow the concepts of history and serialization graph from [31]. B denotes a PACT batch that can be considered as one large transaction, while $\rightarrow$ denotes a precedence relation between two conflicting transactions.

Furthermore, we propose the following theorem as the theoretical basis of the serializability check. The detailed formalization and proof of the theorem can be found in the appendix A

Theorem 1 *A history H generated by* `Snapper`*'s hybrid processing is conflict serializable if:*

(1) $\forall B_i \rightarrow B_j$, $i < j$ (i, j are batch IDs);

(2) the execution of all ACTs is conflict serializable;

(3) $\forall ACT$ $T_i \in SG(H)$, $max(BS_{T_i}) < min(AS_{T_i})$.

Conditions (1) and (2) of the theorem are enforced by `Snapper`'s PACT and ACT concurrency control protocols, respectively, which provide serializability for either purely deterministic or purely nondeterministic processing. Condition (3) of Theorem 1 is the key point for enabling serializability of hybrid schedules. For each ACT T_i, `Snapper` must check if $max(BS_{T_i}) < min(AS_{T_i})$ holds. If not, T_i should abort. To calculate $max(BS_{T_i})$, we have to consider batches that have a path to T_i in the serialization graph. On each actor that is accessed by T_i, we consider the *bid* of the batch that is before T_i and closest to T_i in the actor's local schedule. This batch is guaranteed to have the maximum *bid* among all the batches that are in the local schedule and belong to BS_{T_i}.

However, considering only the actors accessed by T_i is not enough. There may exist batches that belong to BS_{T_i} but do not access any actor accessed by T_i. For example, if $B_k \rightarrow T_j$ on actor A_1 and $T_j \rightarrow T_i$ on A_2, then B_k should also be included in BS_{T_i}. To take such batches into account, on an actor accessed by T_i, we also consider $max(BS_{T_j})$ in the calculation of $max(BS_{T_i})$ if $T_j \rightarrow T_i$ is true in the actor's local schedule. Since the serializability check focuses on $max(BS_{T_i})$, not a complete BS_{T_i}, it is sufficient to only take $max(BS_{T_j})$, instead of querying all possible B_k. Besides, as for the cases that a transaction T_p transitively precedes T_i, e.g., $T_p \rightarrow T_j \rightarrow T_i$, we

do not need to consider $max(BS_{T_p})$ directly in the calculation of $max(BS_{T_i})$, because $max(BS_{T_p}) \leq max(BS_{T_j})$.

The intermediate results of $max(BS_{T_i})$ and $min(AS_{T_i})$ collected on each participating actor after executing T_i are propagated as part of `TxnExeInfo` (see Fig.3.4c) all the way to T_i's local coordinator, which calculates the final values of $max(BS_{T_i})$ and $min(AS_{T_i})$ and performs the serializability check. If T_i passes the check, $max(BS_{T_i})$ should be propagated together with the `Commit` message to all of T_i's participating actors when T_i commits. This value may be useful for the serializability check for the subsequent ACTs.

Note that this implementation does not guarantee that we can obtain the complete AS_{T_i}. When T_i finishes execution on an actor, there may not be any batch B such that $T_i \rightarrow B$. Due to asynchrony in actor systems, a batch can take an arbitrarily long time to reach an actor. It is also possible that there is no batch scheduled after T_i for the actor. An incomplete AS_{T_i} could result in an incorrect $min(AS_{T_i})$ and a wrong decision in the serializability check.

A possible solution is that T_i's local coordinator obtains the complete AS_{T_i} from the PACT coordinators. To achieve this, it has to contact all the PACT coordinators and obtain the schedules of all the actors involved in T_i. This is costly and would increase the commit latency of T_i significantly.

For efficiency and simplicity, `Snapper` only performs the serializability check based on the information available in the local coordinator. It fails an ACT T_i's serializability check if AS_{T_i} is incomplete. AS_{T_i} is said to be incomplete if there exists an actor A involved in T_i such that no $T_i \rightarrow B$ can be found, where B is a PACT batch. However, this approach may cause unnecessary aborts. To mitigate the problem to a certain degree, `Snapper` adopts an optimization in the cases where AS_{T_i} is incomplete: if BS_{T_i} is empty or all the PACT batches in BS_{T_i} have already committed, T_i can pass the serializability check. This optimization is based on the fact that all the batches in AS_{T_i} have not yet started their execution because they must wait for T_i to commit or abort. Since PACT batches are executed in *bid* order, there will not exist $bid \in AS_{T_i}$ such that $bid \leq max(BS_{T_i})$.

3.3.4.4 Commit Protocol

Under hybrid execution, PACTs and ACTs can interleave and depend on each other. The commit protocol guarantees that a transaction commits before the transactions that depend on it. For PACTs, a batch starts executing after previous ACTs have been committed or aborted, so PACTs can commit the same way as described in the PACT commit protocol (Section 4.2.5). By contrast, ACTs may start executing before the previous batch has committed, so an ACT must wait for its dependent batches – batches in its BS – to commit. Upon the completion of the operations of an ACT T, `Snapper` first carries out the serializability check on T and then commits it using 2PC when

the batch with $bid = max(BS)$ has committed, which indicates that all the batches that T depends on have been committed.

3.3.4.5 Recovery

Upon failure, with the log records written for PACTs (Fig.3.7) and ACTs (Fig.3.8), each actor is able to rollback to the state where the last ACT or last batch committed.

3.4 Single Node Evaluation

In this section, we evaluate the performance of PACT, ACT and hybrid execution of `Snapper`. Specifically, we investigate the characteristics of PACT and ACT under different transaction sizes (Section 3.4.2.1) and workload skewness (Section 3.4.2.2). In Section 3.4.3, we present the performance of hybrid execution under different workload skewness and distribution of hybrid workload, and study the trade-offs of hybrid execution with regards to transaction throughput, latency, and abort rate. In Section 3.4.4, we study how well each concurrency control method in `Snapper` can scale.

3.4.1 Experimental Settings

3.4.1.1 Benchmarks

We use two benchmarks throughout the experiments: **SmallBank** [11] and **TPC-C** [174]. SmallBank is an OLTP benchmark simulating basic operations on bank accounts [11]. Each user account is implemented as an actor in the SmallBank benchmark. We employ SmallBank as it is a synthetic workload that approximates well a realistic actor-oriented workload, which is usually reactive, write-intensive, and latency-sensitive. To simulate multi-actor workloads, we implement a *MultiTransfer* transaction that withdraws money from one account and deposits money to multiple other accounts in parallel [155]. SmallBank is used in most of our experiments because it is easy to configure and has predictable behavior. A similar choice for a synthetic workload that can be run under different access distributions was also used to evaluate other actor-based transactional implementations [155, 55]. TPC-C is an industrial-standard OLTP benchmark. Similar to previous work [172], we only use the `NewOrder` transaction of TPC-C in our evaluation, as the `NewOrder` transaction accesses products stored in different warehouses and thus is naturally distributed. In our experiments, we can flexibly control the distribution and the size of `NewOrder` transaction by modeling a warehouse as an actor and partitioning the stock table into multiple actors [172].

3.4.1.2 Deployment

We run `Snapper` on Orleans 3.4.3. We deploy Orleans clients and server (the latter called silo in Orleans [122]) on two AWS EC2 instances (c5n), respectively. Each instance has a 4-core 3.0GHz processor, 10.5GB memory and the silo instance is attached with a 16GB io2 SSD volume with 8K IOPS. All the instances are located in the same region and availability zone. In the scalability experiment, as is shown in Fig.3.12a, all the computing resources scale proportionally with the 4-core setting as a base unit.

On the client side, we implement a push-pull queue, where a producer thread keeps generating transactions and pushing transaction requests to the queue, and multiple client threads pull from the queue concurrently. Each client thread simulates an Orleans client. Instead of spawning a large number of threads per silo with each sending one request at a time, a single client thread is used to asynchronously emit a pipeline of transactions. Whenever a transaction result returns, the client pulls a new transaction from the queue and issues it to replenish the pipeline. The number of client threads and their pipeline size decide the maximum number of concurrent transactions running in the system. Fig.3.12b presents the pipeline size we set for different workloads and concurrency control methods. The pipeline size is tuned such that PACT/ACT can reach a good performance while the system is not over-saturated.

Silo				**#client thread**		
CPU	coordinator	persisting object	actor	PACT	ACT	hybrid
4N	8N	8N	10,000N	1N	1N	1N + 1N

(a) Scalability setup (scale factor: N)

pipeline size	non-transactional	64				
	PACT					
	ACT	64	16	8	4	4
	hybrid	64 + 64	64 + 16	64 + 8	64 + 4	64 + 4
transaction size		2, 4, 8, 16, 32, 64				
skewness (*zipfian* constant)		uniform (0)	low (0.9)	medium (1.0)	high (1.25)	very high (1.5)

(b) Pipeline size per client thread

Figure 3.12: Experimental setting

3.4.1.3 Methodology

All our experiments are run in 6 epochs with the first 2 epochs used for warming up the system. Each epoch lasts for 10 seconds. Three metrics are measured in our experiments: throughput, latency, and abort rate. Throughput and latency only include statistics of successfully committed transactions.

Transaction latency is recorded as the interval from the time that a transaction is emitted by the client thread to the time that the client receives the transaction result. Note that we only report the processing latency, but not the queuing latency, i.e., the time that a transaction is buffered in the push-pull queue. The latter heavily depends on the input rate, and it increases exponentially when the input rate is getting closer to the system throughput.

Besides comparing the different execution strategies provided by `Snapper`, we also compare `Snapper` with non-transactional execution (`NT`) on Orleans and Orleans Transaction (`OrleansTxn`) shipped with Orleans 3.4.3.

3.4.2 PACT vs. ACT Execution

To examine the performance of PACT and ACT with different degrees of contention, we compare PACT and ACT under various transaction sizes and workload skewness. This group of experiments are run with `MultiTransfer` transactions on a 4-core silo with 10K transactional actors.

3.4.2.1 Effect of Transaction Size

We define transaction size (*txnsize*) as the number of actors accessed by a transaction, which reflects the transaction complexity. In this experiment, we vary *txnsize* to measure the overhead of the transactional support provided by `Snapper`. Transaction overhead is measured as the relative throughput of PACT and ACT with regards to the throughput of a non-transactional (`NT`) implementation. As `NT` only processes actor calls without any logic of concurrency control, its throughput comprises an upper bound for executing transactions on Orleans. In this section, we set the workload skewness to be `uniform` and fix the pipeline size to 64.

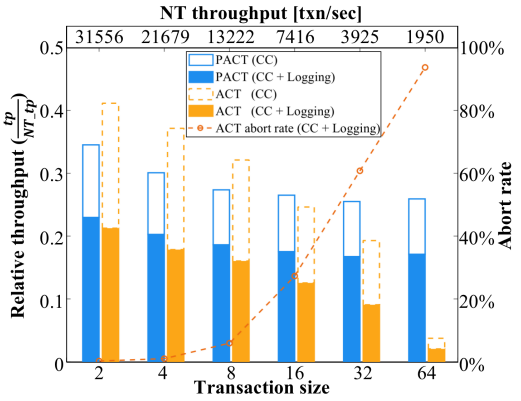


Figure 3.13: Transaction overhead

CC + Logging [milliseconds]

txnsize		2	4	8	16	32	64
PACT	50th	8	13	23	46	94	189
	90th	15	20	40	76	140	261
	99th	27	47	65	101	185	311
ACT	50th	7	13	27	57	92	125
	90th	20	33	54	100	176	301
	99th	34	54	86	151	287	589

Figure 3.14: Percentile latency

Fig.3.13 shows that, compared to `NT`, when $txnsize = 2, 4, 8$, concurrency control (`CC`) brings more throughput degradation for PACT than ACT. It

is because PACT costs more messages per transaction under low contention. In this case, each `BatchMsg` can only deliver one transaction to an actor. For example, when $txnsize = 2$, each PACT costs 6 one-way messages (2 `BatchMsg` + 2 `BatchComplete` + 2 `BatchCommit`) while each ACT only costs 2 double-way messages (`Prepare` + `Commit`) to commit a transaction (Fig.3.4). When $txnsize$ increases, the throughput of ACT decreases much faster than PACT because ACT suffers a lot from workload contention. When $txnsize$ grows, more conflicts arise, and thus more transactions will be blocked during execution and possibly aborted to avoid deadlock. As is shown in Fig.3.13, the abort rate of ACT reaches 90% when $txnsize = 64$. By contrast, PACT guarantees no transaction abort due to conflicts by pre-scheduling and PACT benefits more and more from batching because it can amortize the messaging overhead (Section 4.2.2).

As for the overhead of logging, the throughput of PACT (`CC` + `Logging`) and ACT (`CC` + `Logging`) are 70% and 50% compared with the case without logging, respectively. PACT always has lower logging overhead than ACT because PACT writes less to the log than ACT. PACT can amortize the logging overhead by batching even under low contention because the coordinator always writes `BatchInfo` and `BatchCommit` log records for a batch of PACTs no matter which actors they access. When the contention level grows, the `BatchComplete` log record benefits more and more from batching (Fig.3.7). Instead, an ACT always writes two times to logs on the ACT coordinator (`CoordPrepare` and `CoordCommit`) and two times to logs (`Prepare` and `Commit`) per accessed actor (Fig.3.8). With the combined effects of `CC` and logging, PACT outperforms ACT under all contention levels.

Fig.3.14 shows the difference between PACT and ACT in terms of transaction latency when both `CC` and logging are enabled. When $txnsize < 64$, PACT has almost the same median latency as ACT. When $txnsize = 64$, however, PACT exhibits higher median latency than ACT, namely 189 vs. 125 milliseconds. The latter occurs because all PACTs are delayed to be executed and committed in batches. When $txnsize < 64$, this impact is not very evident because PACT does logging in a more efficient way. When $txnsize = 64$, the enforced batching dominates the influence on latency. By contrast, ACT always has much higher 90th- and 99th-percentile latencies than PACT. When $txnsize = 64$, ACT gets almost 2x higher 99th-percentile latency than PACT. This effect emerges as ACTs that experience high contention would be blocked for a significantly long time. PACT has its tail latency lying in a moderate range (around 1.3x of 90th-percentile latency), because every actor follows a deterministic schedule without non-deterministic blocking.

In conclusion, ACT introduces more overhead than PACT because ACT suffers from high contention and its logging protocol is less efficient. In contrast, PACT reaches good throughput and predictable transaction latency under different $txnsize$.

3.4.2.2 Effect of Workload Skewness

Workload skewness defines the asymmetry in the chance that each actor is accessed by a transaction. In a highly skewed workload, transactions access only a small set of actors, which causes high contention on them. We use a `zipfian` function implemented in the `MathNet.Numerics.Distributions` package [105] to generate different skewed workloads by varying the `zipfian` constant. Fig.3.12b gives the `zipfian` value of five skew levels we used in the experiments.

In this section, we compare the throughput of PACT and ACT under different workload skewness. We fix *txnsize* to 4 and enable both `CC` and logging. We also run the same experiment using `OrleansTxn`. Both ACT and `OrleansTxn` have S2PL as concurrency control method and 2PC as commit protocol [26]. The main protocol differences between them are that ACT does not perform Early Lock Release [55, 159, 14] and ACT uses wait-die to avoid deadlocks, while `OrleansTxn` uses a timeout mechanism. We set the pipeline size for `OrleansTxn` the same as for ACT and implement its transactional storage provider [119] by forwarding logging requests to the same number of loggers as ACT does.

Fig.3.15 shows that the throughput of both ACT and `OrleansTxn` decreases with increasing skewness. Both ACT and `OrleansTxn` suffer from high contention. We also ran `OrleansTxn` with a deadlock-free workload, which is generated by accessing actors in the order of actor ID. Without deadlocks, `OrleansTxn` gets 0% abort rate and relatively higher throughput compared to the one with possible deadlocks. Either with or without deadlocks, `OrleansTxn` gets lower throughput than ACT.

By contrast, the throughput of PACT increases under higher skewness. PACT benefits from high skewness because batching becomes more efficient. As discussed in Section 3.3.2.2, one message can deliver more transactions for processing under a skewed workload, and one log record can cover committed data of more transactions.

In conclusion, ACT is more sensitive to contention, while PACT can benefit from it by batching. Changing from ACT to PACT can thus bring about performance improvements in this scenario.

3.4.2.3 Microbenchmarking ACT and Orleans Transaction

In Fig.3.15, we observe a surprisingly significant performance gap between ACT and `OrleansTxn` even on the cases without deadlocks. This effect is unexpected because both of them adopt very similar mechanisms in concurrency control (2PL) and transaction commit (2PC). In this section, we microbenchmark both systems to investigate the causes of this performance gap. As `OrleansTxn` adopts early lock release, which may suffer from high abort

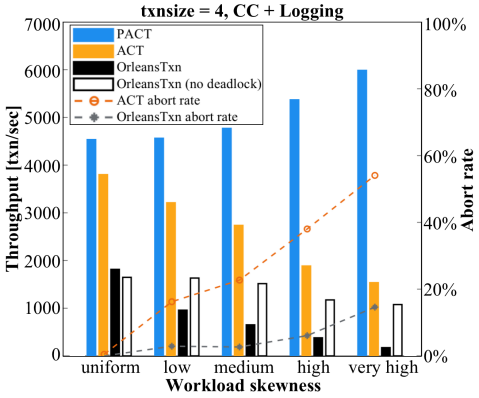


Figure 3.15: Throughput

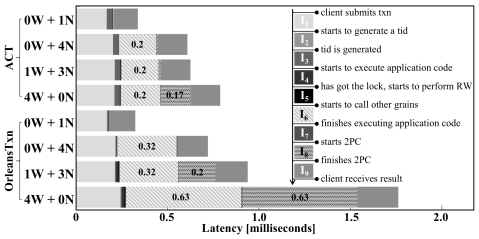


Figure 3.16: Breakdown latency

rate when the workload has high contention, we run the experiment with a conflict-free workload as described below to eliminate this effect.

We use a variant of the `MultiTransfer` transaction that allows for a varying number of actors to perform no-op grain calls in each transaction instead of calls with `ReadWrite` (`RW`) operations. Actors that perform a no-op will not be involved in the commit protocol. We use $xW + yN$ to represent a transaction that accesses $x+y$ actors with the first x actors each performing a `RW` operation and the subsequent y actors executing a no-op. The experiments are run on a 4-core silo with 4 transactional actors. Logging is enabled, and the pipeline size is set as 1 so that the workload has no conflicts.

As is shown in Fig.3.16, the transaction life cycle is divided into 9 time intervals $(I_1, ..., I_9)$. For example, I_2 is the time that the coordinator in `Snapper` or the `TransactionAgent` (`TA`, an in-memory singleton object) in `OrleansTxn` assigns a `tid` for the transaction. I_6 is the time that the first accessed actor serially invokes calls to other actors. I_8 is the time to commit the transaction.

In Fig.3.16, for $0W + 1N$, ACT and `OrleansTxn` have almost the same total latency. For $0W + 4N$, I_6 is the time for serially performing 3 actor calls. `OrleansTxn` takes 1.6x more time on I_6 than ACT (0.32ms vs. 0.2ms). This difference indicates that actor calls under a transaction context are more expensive for `OrleansTxn`.

For $1W + 3N$, one `RW` operation is performed on the first accessed actor and the actor needs to carry out one-phase commit. `OrleansTxn` takes substantially more time on I_8 than ACT (0.2ms vs. 0.01ms). This effect occurs because `OrleansTxn` incurs on one `Prepare` message from the `TA` to the first accessed actor to start the commit process, while in this case ACT requires no messages for 2PC since the first accessed actor is designated as the coordinator of the 2PC protocol. Furthermore, `OrleansTxn` spends significantly more time on performing 2PC than ACT does. The gap increases as more actors are involved in the commit.

ACT and `OrleansTxn` have distinct codebases and they adopt disparate software stacks. So despite similar algorithms being used in both systems, we observe that dissimilarities in performance are spread over many operations and components. Thus, we ascribe their performance gap to their differences in implementations. A more detailed analysis and comparison of implementation details, e.g., data structure overheads, between OrleansTxn and ACT exceed the scope of our work.

3.4.3 Performance of Hybrid Execution

In this section, we investigate the performance of hybrid execution under different transaction distributions, namely the percentage of PACTs among all transactions ($PACT\%$) and different workload skewness. We use the `SmallBank` benchmark for this group of experiments. We fix $txnsize$ as 4 and enable both `CC` and logging. On the client side, we spawn two client threads to handle PACT and ACT requests, respectively. The settings of pipeline size are shown in Fig.3.12b. To vary the $PACT\%$, we let the producer randomly generate $PACT\%$ PACTs among all transactions.

3.4.3.1 Throughput

Fig.3.17 shows the throughput of hybrid execution. Under each level of skewness, we vary $PACT\%$. In each bar, different colors represent the part of the throughput contributed by PACT or ACT. We observe that the total throughput decreases with decreasing $PACT\%$. Ideally, $total\_tp = PACT\% \times PACT\_tp + ACT\% \times ACT\_tp$, but the actual throughput is lower. This effect arises because: (1) the scheduling of PACTs and ACTs influences each other, i.e., PACTs force ACTs to wait for batch processing, and PACTs are blocked until the previous ACTs are committed or aborted; (2) ACTs will also be aborted due to conflicts with PACTs, in addition to conflicts between ACTs themselves.

This mutually and transitively blocking behavior between PACTs and ACTs is even more severe under high skew levels, where most transactions access the same one or two actors. That is why there is a notable throughput degradation in this case from 100% to 99% PACT and from 0% to 25% PACT under `high` and `very high` skew levels. So with an extremely skewed workload, we can benefit from hybrid execution only if we have a small percentage of ACTs.

In conclusion, hybrid execution can bridge the performance gap between pure PACT and pure ACT in most scenarios. Under higher skew levels, hybrid execution performs better than pure ACT if the $PACT\%$ is high.

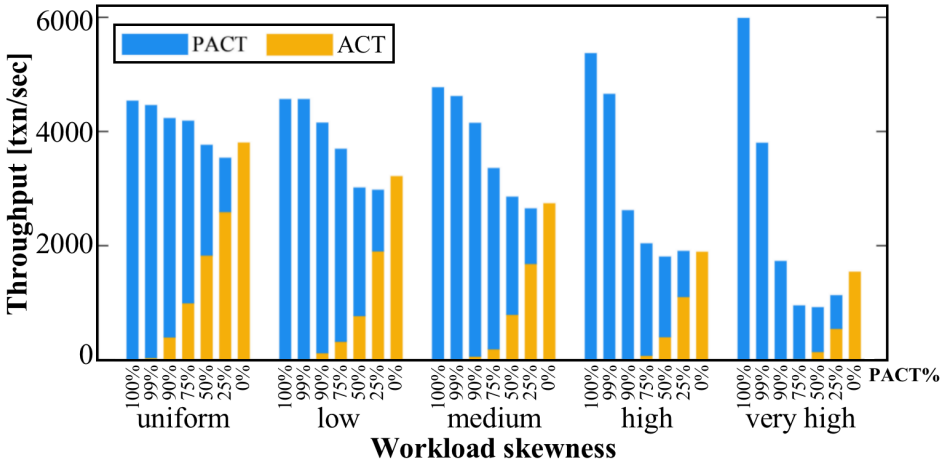


Figure 3.17: Hybrid execution (throughput)

3.4.3.2 Latency

Fig.3.18 shows the latency of PACT and ACT under hybrid execution. Similarly to Fig.3.14, PACT has higher 50th-percentile latency than ACT because of batching. Under hybrid execution, PACT's 90th-percentile latency is influenced by ACT.

When the workload skewness is fixed, for both PACT and ACT, both the 50th- and 90th-percentile latencies increase first and then decrease when $PACT\%$ decreases.

As for PACT, when adding some ACTs to a pure PACT workload, PACTs scheduled after ACTs are blocked until the ACTs finish 2PC. When more ACTs are added, PACT latencies start to decrease. The latter is because there are less and smaller batches, which indicates a lower possibility that a batch be influenced by transitive blocking. Besides, PACT latency starts to decrease at higher $PACT\%$ under higher skewness. This effect arises because more ACTs are quickly aborted due to high contention.

As for ACT, when adding a few PACTs to a pure ACT workload, ACTs have their latency increased due to the blocking caused by PACTs. Then, ACT latency decreases when adding more PACTs. The latter occurs because many long-latency ACTs were actually aborted due to deadlocks between PACTs and ACTs as well as ACTs failing the serializability checks. Recall that the aborted ACTs are not counted in latency statistics.

3.4.3.3 Abort Rate

In hybrid execution, an ACT can be aborted in multiple scenarios: (1) aborted due to read/write conflicts between ACTs; (2) aborted due to deadlocks between PACTs and ACTs; (3) aborted to guarantee global serializability even though we are not sure whether the ACT breaks global serializability because

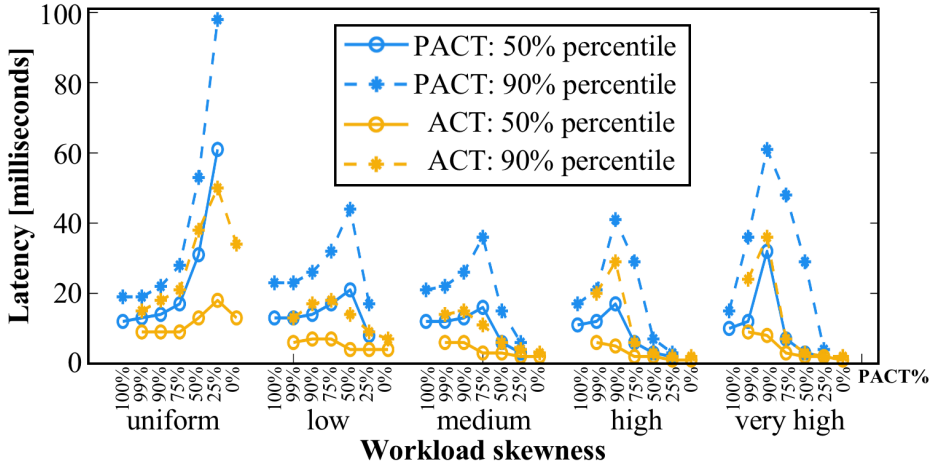


Figure 3.18: Hybrid execution (latency)

the ACT has an incomplete $AfterSet$; (4) aborted because the ACT surely breaks the global serializability. Fig.3.19 shows the breakdown of the transaction abort rate. Most of the aborts are from (1) and (3). Under higher skewness, more ACTs are aborted due to (2). When adding a few PACTs to a pure ACT workload, (3) emerges and causes the total abort rate to become higher than for a pure ACT workload.

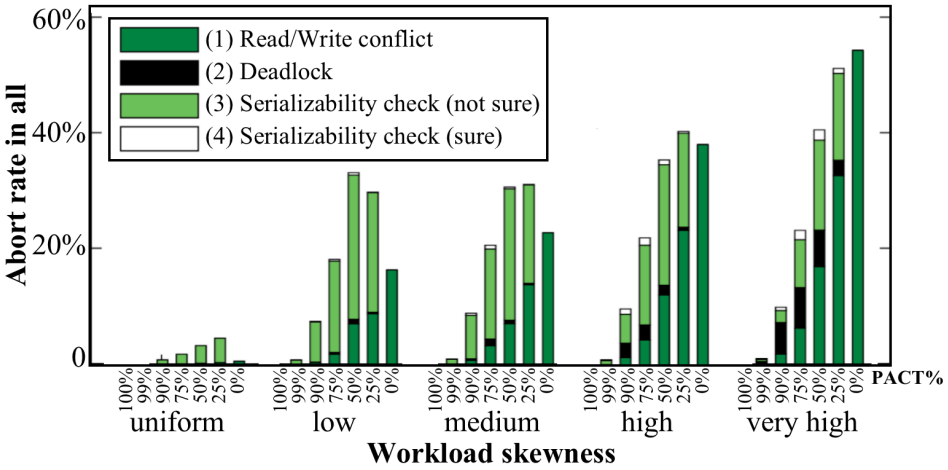


Figure 3.19: Hybrid execution (abort rate)

3.4.4 Scalability

Here, we evaluate the scalability of `Snapper` with both `SmallBank` and `TPC-C` benchmarks by increasing the number of cores in the silo from 4 to 32.

3.4.4.1 SmallBank

We set up the silo as shown in Fig.3.12a. We fix *txnsize* as 4 and enable both `CC` and logging. We present results under uniform and skewed workloads. By following the experiments in [172], the skewed workload is generated by a *hotspot* method that has 1% of the actors in the hot set and each transaction accesses three such actors in the hot set. For the skewed workload, we set pipeline size as 64 for PACT and 4 for ACT. As shown in Fig.3.20, PACT, ACT, and hybrid execution all scale nearly linearly with the uniform workload. With the skewed workload, however, PACT outperforms ACT.

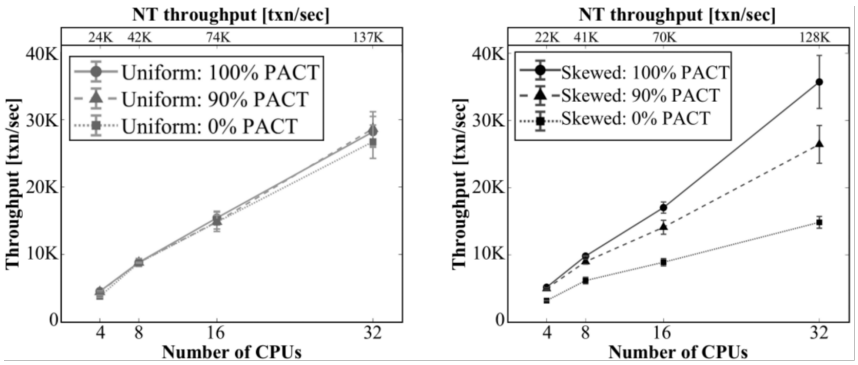


Figure 3.20: Scalability (`SmallBank`)

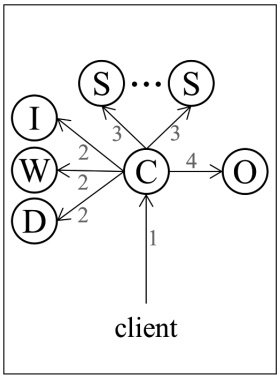
3.4.4.2 TPC-C

In this experiment, we model each warehouse as an actor. Within one warehouse, different actors are used to store different tables and the tables are partitioned as shown in Fig.3.21. In our implementation, every `NewOrder` transaction accesses on average 15 actors, three of which are read-only, allowing us to control the footprint of state updates and to spread transaction processing across multiple actors. We deploy two warehouses for a 4-core silo and the number of warehouses scales with the number of CPUs. We run the experiment with workloads under two skew levels by varying the number of partitions of the `Order` table.

Fig.3.22 shows that both PACT and ACT can scale nearly linearly under low skew. Similarly to the result of `SmallBank`, PACT performs better than ACT under high skew. Compared to NT, both PACT and ACT introduce around 90% throughput degradation, which is comparable with the 85% degradation observed for `MultiTransfer` with $txnsize = 16$ (Fig.3.13). The degradation is mainly due to inefficient logging. In our implementation, all the actors always log the whole actor state instead of doing incremental logging. The latter is due to the fact that we have not implemented a data model for actor states in `Snapper`, which then treats each actor's state as a value blob. It is inefficient to log a whole table when it is insertion-only such as the `Order`,

actor	table	# partitions	# records per actor	read only
I	Item	1 per warehouse	100K	✓
S	Stock	10K per warehouse	10	✗
W	Warehouse	1 per warehouse	1	✓
D	District	1 per district	1	✗
C	Customer	1 per district	3K	✓
O	Order	low skew: 2 per district high skew: 1 per district	grow with NewOrder transactions	✗
	NewOrder	1 per district		
	Orderline			

(a) Table partitioning



(b) NewOrder transaction

Figure 3.21: TPC-C Setup

`NewOrder` and `Orderline` tables. In an avenue for future work, data models can be implemented in `Snapper` to enhance logging performance.

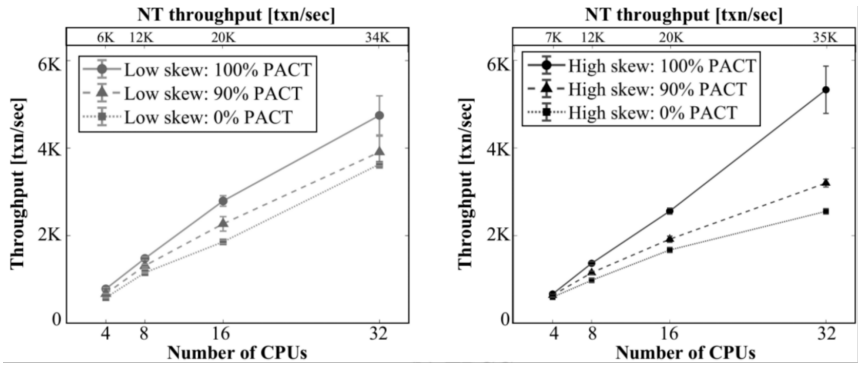


Figure 3.22: Scalability (`TPCC`)

3.5 Related Work

3.5.1 Actor-Oriented Databases (AODBs)

The concept of AODBs is to enrich actor systems with database abstractions in a pluggable fashion [30]. In recent years, several studies have contributed to enriching the features of AODBs, including indexing [30] and geo-distribution [27] of actor states, and distributed transactions across actors [55]. `Snapper` also contributes to this direction by introducing novel transaction execution techniques to actor systems.

3.5.2 Deterministic Database Management Systems

Deterministic database management systems (DDBMS) like Calvin [172] also apply deterministic pre-scheduling to execute transactions in a pre-determined order. Deterministic database systems primarily focus on ordered state machine replication, such that given the same sequence of transactions, replicas would end up in a consistent state [146]. Therefore, DDBMS assumes transactions being deterministic, i.e., generating the same results when executed multiple times. To process non-deterministic transactions, the system has to employ a pre-processing layer to analyse the procedure calls and substitute any non-deterministic codes with deterministic ones. By contrast, `Snapper` does *not* require the computation logic of PACTs to be deterministic, but only requires the actors that are accessed by PACTs and the number of times they are accessed to be declared upon invocation. `Snapper` leverages deterministic execution in order to gain performance. Besides, DDBMS usually handle failures by replaying transactions. `Snapper` does not follow this design. With hybrid execution, recovering by replaying may not be more efficient than loading the logged states because a PACT batch may depend on ACTs. Moreover, `Snapper` supports transactions implemented using the actor model, which is significantly different from the programming model of stored procedures in deterministic database systems.

In addition to the above, `Snapper` proposes the hybrid execution strategy to concurrently execute transactions with and without pre-declared information, thus providing developers with the flexibility to execute transactions in two different modes instead of forcing the deterministic paradigm. Some DDBMS support transactions whose read/write set is unknown by inferring the read/write set through a read-only reconnaissance query [171] or an offline symbolic execution [81]. Other recent work [53, 100] applied deterministic optimistic concurrency control (DOCC), which does not require a known set of data items in the execution phase and performs a validation phase in a deterministic order. All of these existing methods only consider executing transactions deterministically.

3.5.3 Transaction Dependency Analysis

Transaction dependency analysis has been exploited in many studies with an aim to achieve higher throughput [157, 113, 65, 179, 182, 128] and lower latency [186]. Existing approaches usually decompose a transaction into pieces according to different rules, such as SC-cycle [157], and analyze dependencies between transaction pieces. With the dependency graph, a schedule can be generated, where independent pieces of transactions are executed in parallel and conflicting operations across transactions are serialized. By contrast, a transaction in `Snapper` is already naturally decomposed by developers into pieces, one per actor. `Snapper` analyzes transaction dependencies at actor

granularity and ensures that every actor executes transactions by following the same global order. Some approaches make assumptions about the execution order of transaction pieces [157, 186]; `Snapper`; however, does not constrain how and how many times each actor is accessed by a transaction. Some approaches may still have transactions abort [157, 186], while `Snapper` guarantees PACTs do not abort due to concurrency conflicts. Some approaches combine transaction decomposition with batching and resolve dependencies between a batch of transactions [182, 128] to facilitate dynamic data partitioning at the batch level. Differently, `Snapper` applies batching to amortize the overhead of messaging and logging.

3.6 Conclusion

This paper presents `Snapper`, which is a transaction library for actor systems providing two actor transaction abstractions, namely PACT and ACT. Transactions using ACT are executed using conventional non-deterministic strategies, while those using PACT can be executed deterministically and can achieve a significantly higher transaction throughput than ACTs, especially under a highly contended workload. The hybrid execution strategy of `Snapper` is able to execute both types of transactions concurrently to improve system performance under a hybrid workload. It is especially beneficial when most of the transactions in the system are PACTs. Furthermore, all the execution strategies in `Snapper` scale well with the number of CPUs under both benchmarks used in our experiments.

As future work, we intend to extend the optimization and evaluation of `Snapper` in a multi-server environment, investigating the trade-offs in algorithms and mechanisms to partition and coordinate transactions across multiple servers. Deploying `Snapper` in a distributed environment is non-trivial. First of all, consider that in a system with both distributed and non-distributed transactions, it is obvious that non-distributed transactions do not need to be globally ordered. In this case, a hierarchical ordering service may be needed to differentiate these two types of transactions. In addition, different deployments can affect system performance differently. For example, the placement of coordinators may significantly influence the token circulation latency, which will also have an impact on transaction latency. In future work, we plan to thoroughly explore different alternatives based on the current single-server design.

Chapter 4

SnapperD: A Scalable Transactional Actor System

The actor model is becoming increasingly popular for developing interactive, large-scale, and highly concurrent applications. Supporting transactions spanning multiple actors and nodes emerges as a critical feature to ensure data consistency. This paper proposes `SnapperD`, a distributed, efficient, and scalable transaction execution library designed for actor systems. `SnapperD` introduces several novel features for distributed actor transaction execution. `SnapperD` employs a hybrid concurrency control approach that can execute transactions in deterministic and non-deterministic modes simultaneously. This is the first of its kind in a distributed environment, allowing for optimizing transactions with deterministic actor access patterns while still accommodating those with non-deterministic actor access patterns.

To minimize the impact of coordination of distributed transactions on non-distributed ones, `SnapperD` adopts a novel hierarchical architecture and optimization techniques for transaction batching and commit protocol. Moreover, to ensure data consistency when actors are migrated between nodes caused by actor life-cycle management, dynamic system scaling, or locality optimization, `SnapperD` supports transactional actor migration with a minimized transaction abort rate. We have implemented `SnapperD` on top of Orleans [122], a popular actor programming framework.

According to our benchmark experiments, `SnapperD` achieves up to 2x and 6x higher throughput compared to `Snapper` [99], the state-of-the-art actor transaction method, and Orleans Transactions [55], Orleans' official transaction execution implementation, respectively.

4.1 Introduction

In the past decades, the actor model [4] has been adopted in a wide range of applications, such as multi-player games [160, 132], social networks [60],

telecommunication [60], E-commerce [169, 40], IoT [126], web applications [2] and blockchain [70]. An application can be modelled as a set of actors by decomposing its functionalities and states into many lightweight and independent units. Each such unit is named an actor, where its business logic is carried out in the computing layer, and its state can be persisted into the backend storage asynchronously. In this way, actors can well serve latency-sensitive and highly concurrent requests. In addition, an actor-based system is easy to scale because its architecture naturally fits in a distributed setup. In the actor model, an actor does not have direct access to the state of another; instead, they interact with each other via asynchronous messages. Thus, to deploy an actor system in a distributed environment, no extra efforts from the developers are needed to enable cross-node communications. Despite the promising features, ensuring application safety and data consistency while achieving high performance is challenging, especially for workloads across multiple actors and nodes.

First of all, nodes often coordinate with each other to guarantee application safety and consistency for cross-node requests or distributed transactions. For example, distributed concurrency control and commit protocols are employed to synchronize different nodes to safeguard transaction ACID properties. There are plenty of research works optimizing the performance of distributed transactions [43, 137, 85, 102] in database systems, while relatively little effort has been devoted to migrating those methods to actor systems [30]. `Orleans` [122], one of the representative actor frameworks, proposed `OrleansTxn` [55]. It optimizes distributed multi-actor transactions by applying Early Lock Release (ELR) [13] to 2PC [90], but its throughput degrades significantly under high contention. `Snapper` [99], an actor-based transaction library built on Orleans, takes advantage of deterministic concurrency control and achieves high transaction throughput in a single node. However, `Snapper` has obvious drawbacks when scaling out. `Snapper` generates deterministic schedules by putting all transactions into one sequential order, which is unnecessary for transactions that can be partitioned into different nodes. Besides, in `Snapper`, all transactions are forwarded to one centralized component for scheduling, which is a potential bottleneck in a distributed setting.

In addition, when the system scales in/out to deal with varying workloads [151] or optimizes data locality, it is required to re-partition or migrate actors between nodes. [1] It is a critical issue how to migrate an actor from one node to another while guaranteeing application safety and data consistency. It is insufficient to follow a simple deactivate-then-reactivate pattern, as is supported in some of the existing actor frameworks [35]. This is because deactivating an actor with concurrent workloads can face the risk of data loss

[1] Looking for an optimal data/actor partitioning strategy to reduce cross-node communications is beyond the scope of this paper because this problem has been thoroughly discussed in a variety of works [115, 95].

and message loss. The former makes it difficult to keep actor states consistent before and after the migration, and the latter could cause transaction aborts. The challenge is to migrate actors in the context of transaction processing while minimizing transaction aborts and, hence, the impact on system performance.

In this paper, we address the aforementioned challenges and develop a distributed, efficient, and scalable transaction actor system, called `SnapperD`. `SnapperD` is built as a library on top of Orleans. In summary, we have made the following contributions:

• `SnapperD` is the first system of its kind that can execute distributed actor transactions in a hybrid deterministic and non-deterministic mode. This allows transactions with pre-declared actor access patterns to take advantage of the high efficiency of deterministic transaction execution, while transactions with non-deterministic access patterns can be executed non-deterministically.

• A naive approach to produce a deterministic execution order for all transactions is to collect all (both distributed and non-distributed) transactions on a central node, which produces a global order. However, this may create a bottleneck, harming the system's scalability.

To address this problem, we propose a novel hierarchical method, where a coordinating master node collects and produces a global order for distributed transactions while each worker node independently produces a local order of their own non-distributed transactions. Then, each worker node merges the global order and its local order to form its local deterministic execution order. Both steps are carried out in parallel to prevent any bottleneck when ordering the transactions.

• While `SnapperD` executes transactions in a deterministic order, it differs significantly from deterministic databases. Deterministic databases assume transactions have deterministic logics; hence, in systems like [172], every node, after a data sharing phase, will execute every distributed transaction in parallel independently. Therefore, distributed transactions will have a limited impact on the performance of non-distributed transactions. However, since we cannot assume deterministic transaction logic, distributed transactions in `SnapperD` require coordination between multiple nodes, both during transaction execution and at commit time. As transactions must be committed in a pre-determined order in `SnapperD`, the cross-node coordination of distributed transactions can slow down non-distributed transactions.

To address this particular problem, we propose two optimizations. First, we optimize how transactions are batched, reducing the time that a node is idle while waiting for the execution of a distributed transaction. This will allow the node to process non-distributed transactions while it is waiting. Second, we optimize the commit protocol to reduce the number of messages blocking the critical path of some distributed transactions. This will further enhance the system's throughput.

• As previously stated, utilizing the traditional deactivate-then-reactivate method for runtime actor migration can result in both message and data loss. This can lead to a significant number of transaction aborts. Additionally, given the intricate dependency relationships between actor transactions, migrating an actor A would not only block transactions that access A but also those transactions that do not access A but have a dependency relationship with transactions that do. This would cause many transactions to be aborted due to timeout.
To solve this issue, `SnapperD` employs a novel actor migration method integrated with actor transaction processing. Moreover, `SnapperD` coordinates with the transaction ordering service to prevent unnecessary blocking of transactions not accessing the migrating actors.

• We conducted extensive experiments using the SmallBank [11] benchmark, which is suitable for the actor model. Our experiments demonstrated that `SnapperD` provides a scalable solution for actor transaction execution, outperforming Orleans Transactions [55] by a factor of 6, and a simple distributed extension of Snapper [99] by a factor of 2 in terms of transaction throughput. Additionally, our proposed method for actor migration has a minimal impact on system performance, resulting in only $\sim$ 0.01% transaction aborts.

4.2 Multi-node Architecture

In this section, we divert our attention to a more complex scenario where actors are distributed across multiple nodes, and transactions may access one or more nodes. In this case, it is non-trivial to implement deterministic concurrency control for PACTs with different access patterns and to safeguard the serializability between PACTs and ACTs across multiple nodes while preserving high throughput and scalability. This section discusses challenges and solutions for extending the single-node architecture to a multi-node architecture.

4.2.1 Overview

In the single-node design, all PACTs are put into one sequential order, determining their execution and commit order. However, PACTs accessing one node (**local PACTs**) do not need to be ordered together with PACTs accessing other nodes. Otherwise, as shown in Fig. 4.1a, both the ordering and commit phases of PACTs are slowed down. Apparently, it is optimal to have each node progress independently when the workload can be 100% partitioned. In this scenario, each node should maintain its own transaction schedule (Fig.4.1b). However, in practice, there may exist PACTs crossing multiple nodes (**global PACTs**); thus, a common cross-node ordering is needed. With such a workload, we can adopt a hierarchical ordering strategy

(Fig.4.1c) where a global order is determined for global PACTs and a local order is generated on each node for local PACTs.

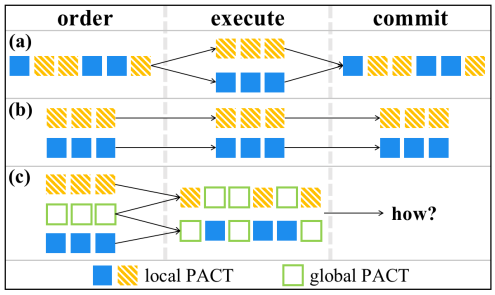


Figure 4.1: Ordering Strategies

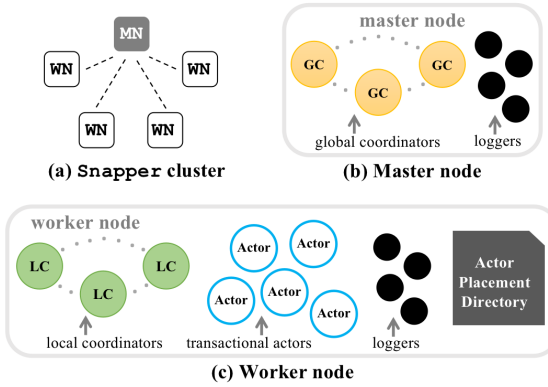
With hierarchical ordering, new challenges emerge, including how to preserve the order of global PACTs in different nodes, how to efficiently coordinate these nodes to commit global PACTs, and how to reduce the impact of global PACTs on the performance of local PACTs while they are executed concurrently. Furthermore, in distributed actor systems, actors sometimes need to be migrated from one node to another for various reasons, such as optimization for enhancing locality and actor life-cycle management. Supporting transactional workload in the presence of actor migration is challenging, too. The design of the multi-node architecture focuses on addressing these challenges.

4.2.2 Extended Architecture

• **Master and Worker Nodes:** In the distributed environment, a `X` cluster consists of one master node and an arbitrary number of worker nodes (Fig.4.2). The master node is a centralized component responsible for orchestrating worker nodes to process distributed transactions. Each worker node accommodates a group of local coordinators, a partition of transactional actors and a configurable number of loggers. In addition, each worker node keeps a local cache of the **actor placement directory**, which maps the actor IDs to corresponding worker node IDs. In a `X` cluster, a worker node can be dynamically added or removed from the cluster without interrupting other nodes.

• **Global and Local Coordinators:** On the master node, a group of global coordinators (`GC`) is organized as the same coordinator-ring structure (§ 3.3.2.1) to provide a global ordering service. They generate globally unique transaction identifiers and do ordering and batching for global PACTs. On each worker node, a group of local coordinators (`LC`) function in a similar way as in the single-node architecture.

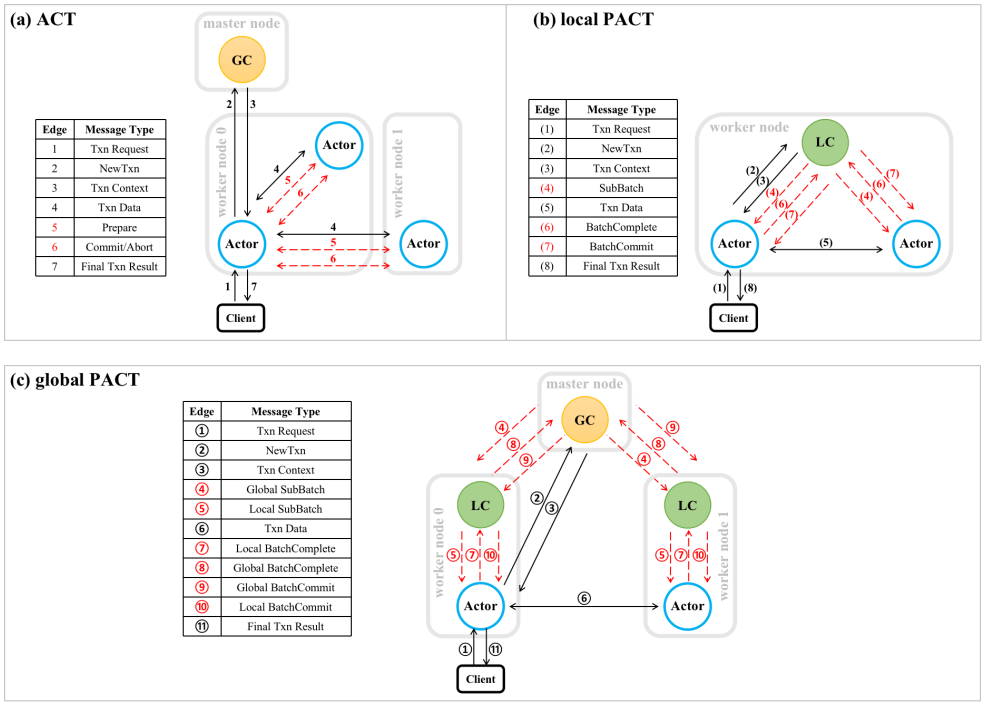
• **Global and Local Transactions:** The global and local transaction intuitively refer to a transaction that accesses actors located in multiple and one worker nodes, respectively. Each transaction is identified by a `TxnContext` (Fig.4.2d)in including a local transaction ID (`ltid`), a local batch ID (`lbid`), a global transaction ID (`gtid`) and a global batch ID (`gbid`). These values also imply the type of transaction.



transaction type	TxnContext			
	gtid	gbid	ltid	lbid
ACT	✓	-1	-1	-1
global PACT	✓	✓	✓	✓
local PACT	-1	-1	✓	✓

(d) Transaction context

Figure 4.2: Multi-node architecture



Edge	Message Type
1	Txn Request
2	NewTxn
3	Txn Context
4	Txn Data
5	Prepare
6	Commit/Abort
7	Final Txn Result

Edge	Message Type
(1)	Txn Request
(2)	NewTxn
(3)	Txn Context
(4)	SubBatch
(5)	Txn Data
(6)	BatchComplete
(7)	BatchCommit
(8)	Final Txn Result

Edge	Message Type
①	Txn Request
②	NewTxn
③	Txn Context
④	Global SubBatch
⑤	Local SubBatch
⑥	Txn Data
⑦	Local BatchComplete
⑧	Global BatchComplete
⑨	Global BatchCommit
⑩	Local BatchCommit
⑪	Final Txn Result

Figure 4.3: Transaction workflow (multi-node)

4.2.2.1 Transaction Workflow

Fig.4.3 illustrates the workflows of all three types of transactions – ACT, local PACT and global PACT. In the beginning, a client submits a transaction by calling the `StartTxn` API on the first actor (Edge 1, (1), ①). This actor then identifies the type of transaction. It is an ACT if `actorAccessInfo` is not given. To further distinguish between local and global PACTs, the locally cached actor placement directory is retrieved to extract the list of worker nodes where the actors specified in `actorAccessInfo` are located. In the case of a

local PACT, the actor issues a `NewTxn` request to one of the local coordinators (Edge (2)) to get the `TxnContext` (Edge (3)). Differently, for an ACT or a global PACT, a `NewTxn` request is forwarded to one of the global coordinators (Edge 2, ②) to get the `TxnContext` (Edge 3, ③). While executing the transaction, the same as in the workflow of the single-node architecture (§ 3.3.1.2), every cross-actor transactional call carries along the `TxnContext`. To fulfil the deterministic schedule, each actor executes PACTs in the pre-determined order, and the ordering information is sent from coordinators (Edge (4), ④, ⑤). In the transaction commit phase, an ACT performs 2PC across all participant actors (Edge 5, 6). For a local PACT, the commit protocol is carried out within one worker node (Edge (4), (6), (7)). In contrast, for a global PACT, the batch commit process needs coordination between the master node and multiple worker nodes (Edge ④, ⑤, ⑦-⑩).

4.2.3 PACT Processing

4.2.3.1 Hierarchical Ordering

The master node collects and produces a global order for global PACTs by assigning them `gtid`s, while each worker node independently produces a local order of their own local PACTs by assigning them `ltid`s. Then, each worker node merges the global order and its local order to form a local deterministic execution order. More specifically, each worker node assigns `ltid`s for received global PACTs as well to order them together with the local PACTs. To consistently maintain the order of global PACTs across all worker nodes, on each worker node, `ltid`s must comply with `gtid`s, i.e. for $T_{gtid=i,ltid=j}$ and $T_{gtid=i',ltid=j'}$, if $i > i'$, then $j > j'$. It is nontrivial to fulfil this condition.

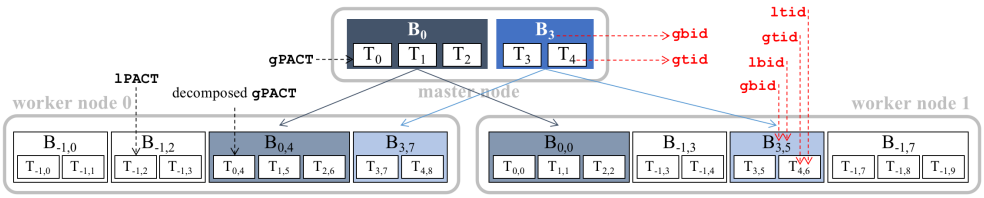


Figure 4.4: Hierarchical Ordering and Batching

Fig.4.4 shows the `gtid` and `ltid` assignment we aim to achieve. A sequence of global PACTs is divided into batches, each of which is later decomposed into several `GlobalSubBatch`es and delivered to corresponding worker nodes (Edge ④ in Fig.4.3c). The decomposition is done based on the list of worker nodes accessed by the batch of global PACTs. Note that different `GlobalSubBatch`es may access different sets of worker nodes, and `GlobalSubBatch` messages may arrive at a worker node in a nondeterministic order. To identify the order between two of such messages, we again resort to the `prev_gbid` relation – each `GlobalSubBatch` message carries a `prev_gbid`, specifying the previous

`gbid` sent to the target worker node. It is the same mechanism that an actor orders different `LocalSubBatches` (§ 3.3.2.2). Recall that the token-passing method is able to generate monotonically increasing IDs. Thus, we just need to guarantee that the `ltid` assignment for PACTs in a `GlobalSubBatch` happens after the specified previous one has been assigned.

To keep track of the last `GlobalSubBatch` that has been assigned with `lbid` and `ltids` across multiple local coordinators, the `last_processed_gbid` information should be included in the token. Every time a local coordinator receives the token, it assigns `ltids` for accumulated local PACTs, as well as checks received `GlobalSubBatches` and assigns `ltids` for the ones with `prev_gbid` equals `last_processed_gbid`. In this way, global batches are inserted into the sequence of local batches while preserving the pre-determined order.

4.2.3.2 Transaction Context Remapping

In the transaction execution phase, as long as every actor executes PACTs in the order of `lbids` and `ltids`, the pre-determined serial order for all PACTs is guaranteed. As explained in § 3.3.2.3, an actor identifies a PACT using the `TxnContext` carried along the actor call. However, in the multi-node setup, when a call is forwarded from a different worker node, the `ltid` and `lbid` in the `TxnContext` are not valid in the new worker node. For example, in Fig.4.4, $T_{gtid=0,ltid=4}$ in worker node 0, should be mapped to $T_{gtid=0,ltid=0}$ in worker node 1. Recall that every `LocalSubBatch` message carries a sorted list of `ltids` (§ 3.3.2.3), which informs the execution order of PACTs in the batch. Additionally, we can include a dictionary that maps *gbid* and *gtid* of each global PACT to *lbid* and *ltid*. By retrieving this map, an actor is able to convert a remotely forwarded `TxnContext` to the local context.

4.2.3.3 Tuned Batching

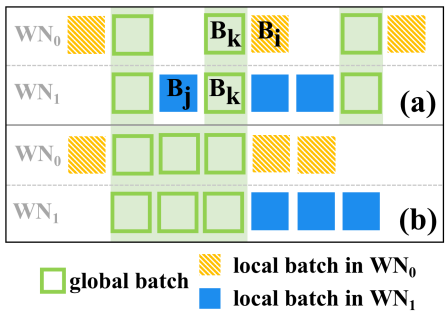


Figure 4.5: Effect of batch tuning

Although local batches are processed in parallel in different worker nodes, the existence of global batches may lead to performance degradation because they force coordination across multiple worker nodes. As is shown in Fig.4.5a, B_i is blocked by B_j through a global batch B_k. Such blocking brings some vacancies to the progress of a worker node. To fill these gaps, we adopt a simple strategy – increase the global batch size to reduce the interleaving between local and global batches. As is shown in Fig.4.5b, if we merge all three global batches into one,

the vacancies are reduced. Note that the batch size should not grow without limit because it sacrifices global PACTs' latency and brings uneven waiting time for local PACTs. Recall that in the single-node architecture (§ 3.3.2.1), every time a coordinator receives the token, it generates a new batch with all cached PACTs. To support a tuneable global batch size, we make a little adjustment. When a `GC` receives the token, it first checks if the pre-configured time interval (e.g. 20ms) has passed since the last time it created a global batch. If not, the `GC` will pass on the token without forming a new batch and wait until it receives the token again. Our experiment results show that a tuned batch size improves throughput significantly (§ 4.4.2).

4.2.3.4 Hierarchical Commit Protocol

In § 3.3.2.4, a 2PC-like batch commit protocol is introduced for local PACTs (Fig.3.7). The general idea is to commit a batch B when (1) B has completed execution on all related actors and the execution results have been persisted, and (2) all the batches that B logically dependent on have been committed. Here, we adopt the same principle. The first condition can be fulfilled by letting each participant actor log the actor state and then acknowledge the completion of B to a local or global coordinator. To meet the second condition, local and global coordinators must commit batches in the order of `lbid`s and `gbid`s, respectively. However, it is non-trivial to enforce this constraint because the commit of a global batch B is restricted by not only its `gbid` but also its `lbid`s assigned in different worker nodes. In other words, before committing B, we must ensure the commit of all batches with smaller `gbid`s and smaller `lbid`s in corresponding worker nodes.

Here, we resort to a hierarchical commit process to enforce the commit order of local and global batches. As is shown in Fig.4.6, we still do the three-round communications (batch information, batch complete and batch commit) between actors and global coordinators, but the difference is each message has an extra hop at local coordinators. Now, we explain the process to commit a global batch, whereas the commit of a local batch remains the same as in the single-node architecture (§ 3.3.2.4). The batch information is first delivered from a global coordinator to a local coordinator (Edge ④), then sent to related actors (Edge ⑤). When an actor completes the batch, it informs the local coordinator (Edge ⑦). When this local coordinator has collected acknowledgements from all relevant actors in the current worker node, it can inform the global coordinator (Edge ⑧). Afterwards, when the global coordinator decides to commit the batch, the batch commit message goes to the local coordinator first (Edge ⑨), then actors (Edge ⑩). The global and local coordinators involved in this whole process are the ones who generate and assign IDs to the global batch.

Note that before sending the `GlobalBatch- Complete` message to the global coordinator (Edge ⑧), the local coordinator needs to wait for the com-

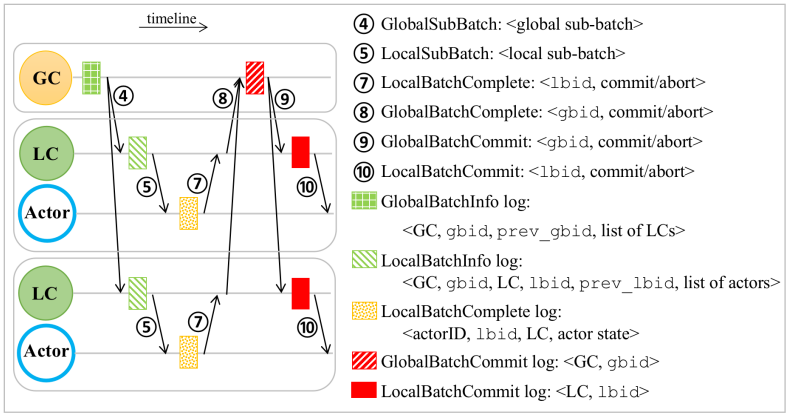


Figure 4.6: PACT commit and logging protocol (multi-node)

mit of all batches with smaller `lbid`s. More specifically, for each batch, only its `last_lbid` is checked since it forms the complete chain of logical dependencies on the current worker node. This synchronization is necessary here because global coordinators lack information about the dependencies between global and local batches in each worker node. Similarly, before a global coordinator decides to commit a global batch, its `last_gbid` is checked in order to enforce the commit order of `gbid`s.

Recall that the `gbid`s of global batches comply with their `lbid`s (§ 4.2.3.1). It indicates possible redundant synchronizations. More specifically, suppose two global batches B_i and B_j are placed consecutively on a worker node – no local batches are inserted between them. When committing B_j, the local coordinator first waits for the commit of B_i on the worker node before sending the `GlobalBatchComplete` message; then, the global coordinator waits for the commit of B_i again on the master node. In this case, the synchronization on the worker node is unnecessary. Fig.4.4 presents a more concrete example. On worker node 0, the global batch $B_{gbid=3}$'s last one is $B_{gbid=0}$, thus, $B_{lbid=7}$ does not need to wait for the commit of $B_{lbid=4}$ because the global coordinators will ensure $B_{gbid=0}$ commits after $B_{gbid=3}$. We apply optimization for such cases to shorten the critical path of the commit process and improve the system performance. This optimization is especially effective for workloads with a high proportion of global PACTs.

4.2.3.5 Abort a batch

In the multi-node setup, due to the speculative execution of PACTs, the abort of a batch, whether local or global batches, may cause cascading abort in multiple worker nodes. This is because the existence of global batches introduces transitive dependencies between two local batches in different worker nodes. In addition, the abort of a batch B will cause the abort of all batches with logical dependencies on B. For example, in Fig.4.4, the abort of a local batch

$B_{lbid=2}$ on worker node 0 may lead to the abort of global batches $B_{gbid=0}$ and $B_{gbid=3}$, as well as local batches $B_{lbid=3}$ and $B_{lbid=7}$ on worker node 1. In `X`, the moment a PACT abort is detected, we first stop the current worker node emitting more batches or assigning IDs to global batches. This is to avoid scheduling more batches that have logical dependencies on the aborted ones. Then, we check if any uncommitted global batch is influenced to decide if the master node or any other worker nodes need to be stopped. During this process, the *min_gbid* is found, which is the earliest global batch that needs to be aborted. Then, the *min_lbid* is found on each affected worker node. When a worker node stops the batch generation process, it continues executing and committing unaffected batches – those ordered before the earliest aborted one, while rolling back the state of affected actors – those accessed by aborted batches. When a worker node completes the cascading abort, its local coordinators resume the batch generation. When the abort of global batches has been completed on all relevant worker nodes, the master node resumes.

4.2.3.6 Logging

During the three-round communications to commit a global batch, `X` persists certain information to guarantee the durability of committed batches and survive failures that may happen at any time on any node. As discussed in § 3.3.2.4, the three logs (`LocalBatchInfo`, `LocalBatchComplete` and `LocalBatchCommit` in Fig. 4.6) are sufficient for a local batch to continue its commit process upon failures or recover its state after failures. Here in the multi-node architecture, we add two extra logs, `GlobalBatchInfo` and `GlobalBatchCommit`. The first tells how the global batch is distributed across worker nodes, and the second keeps a record of the committed *gbid*. In our design, the commit process will be blocked if any message is lost. With the logged information, all missing messages can be retrieved again. For example, the master node can collect information about the progress of a global batch from certain worker nodes based on the `GlobalBatchInfo`. Besides, each worker node can get the commit status of a global batch from the master node by retrieving the `GlobalBatchCommit` log.

4.2.3.7 Recovery

§ 3.3.3.4 explains how to recover failed actors or local coordinators on a worker node. On the master node, global coordinators can apply the same recovery algorithm as local coordinators. It includes three tasks. First, each global coordinator retrieves the log file and loads the information of batches that have not been finished. Second, all global coordinators collaboratively generate a new token if needed. Third, global coordinators start to generate new batches after unfinished ones have all been committed or aborted.

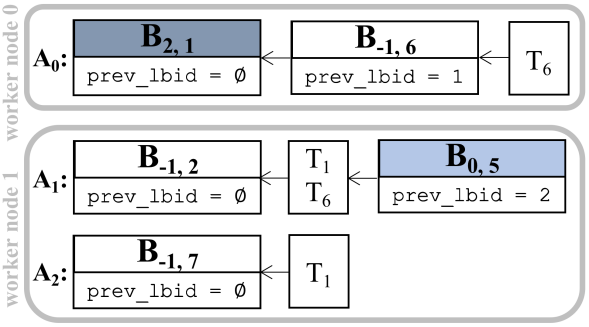


Figure 4.7: Hybrid Schedule

4.2.4 Hybrid Processing

In the multi-node architecture, on each transactional actor, we apply the same hybrid scheduling for ACTs (§ 3.3.4.1), where the ACT is dynamically inserted between ordered batches. In § 3.3.4.3, a serializability check algorithm is used for checking cyclic dependencies between an ACT and PACT batches. More specifically, for an ACT T_i, we check if $max(BS_{T_i}) < min(AS_{T_i})$. Note that an ACT now may have dependencies on both local and global batches because it may invoke calls on multiple worker nodes.

As is shown in the example (Fig.4.7), an ACT T_1 only accesses worker node 1, and is scheduled before $B_{lbid=5}$ on actor A_1 and after $B_{lbid=7}$ on A_2. So T_1 violates serializability. the ACT T_6, accessing both worker nodes, is scheduled after $B_{gbid=2}$ on A_0, and before $B_{gbid=0}$ on A_1. So T_6 also violates serializability. The two cases represent two possible cyclic dependencies that could happen to an ACT. First is the intra-node cycle, which is formed with `lbid`s in one worker node. To detect such cycles, we can simply run the serializability check over the before and after `lbid`s collected on this node. Second is the inter-node cycle, which is formed with more than one `gbid`s. Similarly, to detect an inter-node cycle, the serializability check is run over the before and after `gbid`s collected across multiple worker nodes. In summary, the collected information should include both the closest *lbid*s and the closest *gbid*s scheduled before and after the ACT. For example, for T_6 in Fig.4.7, both $lbid = 6$ and $gbid = 2$ should be included.

The same as in § 3.3.3.3, the scheduling information is collected while executing an ACT, and the serializability check is carried out on the first actor that the ACT accesses before starting 2PC. While performing the serializability check, the algorithm proposed in § 3.3.4.3 should be run for $1 + n$ times – one for detecting the inter-node cycle and one for detecting the intra-node cycle for each involved worker node, in a total of n worker nodes. Only if all checks are passed can the ACT continue to commit. Again, in the existence of an incomplete `AfterSet`, as is discussed in § 3.3.4.3, we allow possible false abort of ACTs to avoid the overhead of collecting accurate and complete batch

schedule information.

In addition, the ACT must commit after the dependent batches have committed. Thus, $n + 1$ `WaitBatchCommit` messages are sent to the master node and related worker nodes. To avoid spamming the system with a lot of such messages, each worker node should cache the *max_committed_gbid* and the *max_committed_lbid* on each worker node. The latest batch commit information is carried in all types of cross-node messages. By doing so, the `WaitBatchCommit` message is needed only when the batch has not been marked as committed in the local cache.

4.3 Live Actor Migration

In the distributed environment, `X` may need to scale in or out to deal with varying workloads [151] or optimize data locality. It usually requires re-partitioning or migrating actors between nodes. Looking for an optimal partitioning strategy to reduce cross-node communications has been thoroughly discussed in various works [115, 95]. This section focuses on another critical challenge: migrating an actor while guaranteeing application safety and data consistency. Following a simple deactivate-then-reactivate pattern is insufficient, as is supported in some of the existing actor frameworks [122]. This is because deactivating an actor with concurrent workloads can face the risk of data loss and message delivery failures [35]. The former makes it difficult to keep actor states consistent before and after the migration, and the latter could cause transaction aborts. This section addresses the challenge of migrating actors under the context of transaction processing while minimizing transaction aborts and, hence, the impact on system performance.

4.3.1 Overview

Unlike live data migration discussed in database systems where the migration task can be easily carried out in between transactions [94] or as a step inside a transaction [95], actor migration requires careful treatment due to the special features of actors and actor runtime. To migrate an actor in Orleans, the basic steps include (1) updating the actor placement directory if necessary, (2) deactivating the actor in the old node, and (3) reactivating the actor in the new node. As for step (1), Orleans provides several built-in actor placement strategies and supports customized ones. This is where an actor partitioning algorithm can be plugged in. The strategy is a method invoked by the Orleans runtime to resolve the location of a newly activated actor. In step (2), Orleans allows the completion of the current request [35] on the actor, then deactivates it by clearing all the in-memory data stored on the actor and forwarding the pending requests to the new location. As for step (3), recall that with virtual actors [29], each request sent to the deactivated actor will automatically trigger a reactivation attempt, and multiple such requests may

lead to race conditions on the actor reactivation process, then eventually end up with message delivery failures. The following discusses how to deal with data and message loss when migrating actors in `X`.

4.3.1.1 Data Loss

An intuitive solution for data loss is to take a snapshot of the actor state before the deactivation and use this snapshot to restore the actor state when reactivating it. However, it is non-trivial to ensure data consistency – requests executed before the deactivation are all reflected in the snapshot, and requests sent after the deactivation should be executed after the actor state is restored. This requires extra control to coordinate the snapshot taking and requests processing and forwarding. Furthermore, the snapshot is supposed to contain all the in-memory data stored on the actor, including the user-defined transactional state, the actor's local schedule, the metadata of unfinished transactions or batches, etc. In this case, the snapshot size grows with the concurrent workload on the actor, which takes more time to copy and transfer between actors and nodes and further slows down the whole system. To limit the snapshot size, we tend to time the migration task – start the migration when there are no unfinished transactions on the actor; therefore, only the actor's transactional state needs to be copied.

4.3.1.2 Message Loss

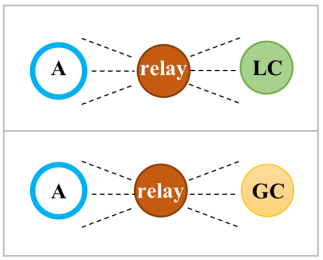
In addition, message loss should be reduced as much as possible. When a transactional request fails to be delivered and processed, the corresponding transaction should be aborted. The abort of an ACT can be handled with 2PC; however, the abort of a PACT may cause severe cascading abort, especially in the multi-node architecture (§ 4.2.3.5). To reduce message loss, an ideal practice is to send the pending or future requests to the migrated actor after the actor is reactivated. By doing so, multiple requests racing to re-create the actor can be avoided. Furthermore, there is the possibility to eliminate the abort of PACT by using the deterministic PACT transaction processing schedule.

4.3.1.3 Implementation

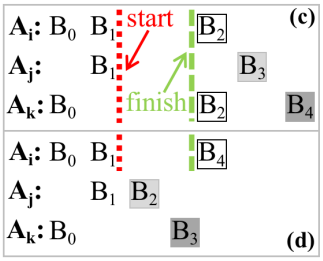
To implement the aforementioned strategies, we introduce two new types of actors. The **migration helper actor** carries out actor migration tasks. Multiple migration helpers are distributed across worker nodes, each responsible for migrating a certain set of actors. More specifically, we assign the migration task of each actor to a fixed migration helper by a simple hash function. The **transaction relay actor** relays messages between transactional actors and local/global coordinators (Fig.4.8b, Edge (2), (3), ②, ③ in Fig.4.3). In each worker node, a group of relay actors gather the `NewTxn` requests from local

```
1  MigrateActor (Ai, targetNode)
2    // STEP 0: validate the request
3    currentNode = actorPlacementDir.Get(Ai)
4    if targetNode = currentNode then return
5    if actorsToMigrate.Contains(Ai) then return
6    actorsToMigrate.Add(Ai)
7    // STEP 1: freeze Ai
8    Ai.prepareToFreeze()
9    max_lbid = -1, max_gbid = -1
10   for relayActor in relayActorList do
11       (lbid, gbid) = relayActor.FreezeActor(Ai)
12       max_lbid = Max(max_lbid, lbid)
13       max_gbid = Max(max_gbid, gbid)
14   // STEP 2: deactivate Ai
15   state = Ai.PrepareToDeactivate(max_lbid, max_gbid)
16   actorPlacementDir.Set(Ai, targetNode)
17   Ai.Deactivate()
18   // STEP 3: reactivate Ai
19   Ai.Activate(state)
20   for relayActor in relayActorList do
21       relayActor.UnFreezeActor(Ai)
22   // STEP 4: garbage collection
23   actorsToMigrate.Remove(Ai)
```

(a) Migration helper



(b) Relay actor



Set a boundary

Figure 4.8: Actor Migration

transactional actors. A relay actor controls when to acquire a `TxnContext` for a request from the corresponding coordinator. As for requests relevant to an actor that is under migration, they are delayed until the migration is done. The number of both types of actors is configurable. Instead of integrating these functionalities into coordinators, we create different types of actors for handling different tasks, given that each actor can only do one thing at a time.

Fig.4.8a shows how a migration helper migrates an actor step by step. The `MigrateActor` API requires an actor ID (A_i) and a target node ID as input. First, the migration helper validates the request (lines 3 - 6). A_i can only be migrated if the target node differs from the current one and A_i is not involved in a previous unfinished `MigrateActor` request. Afterwards, the migration helper freezes A_i to stop sending more PACTs to A_i (lines 8 - 14). When A_i finishes all scheduled transactions (line 16), it is deactivated and reactivated (lines 18 - 20). More details are discussed as follows.

4.3.2 Integrating with PACT Processing

To integrate actor migration with PACT processing, the idea is to set a boundary on A_i such that batches located before the boundary can execute and commit before migrating A_i and batches after the boundary should wait. For example, in Fig.4.8c, the boundary is put between B_1 and B_2 on A_i. In this way, batches with $bid > 2$ (e.g. B_3, B_4) are also blocked by B_2 even if they do not access A_i. It is because B_3 must commit after B_2, and B_4 can only be

executed on A_k after B_2. Due to the restriction brought by PACT ordering, to avoid such unnecessary blocking, we need to re-order batches (e.g. B_2) that will access A_i and arrive after the boundary. Specifically, the `tid` and `bid` assignment of transactions or batches relevant to A_i should be delayed until A_i finishes migration, which can be done with the help of the relay actors. As is shown in Fig.4.8d, PACTs that are initially put in B_2 now are not assigned with *bid* and *tid* until A_i is migrated. In the meantime, PACTs that initially belonged to B_3 and B_4 are put into B_2 and B_3, respectively. By doing so, these two batches can proceed without being blocked by the migration of A_i.

As is shown in Fig.4.8a, the boundary is set with two actions. First (line 8), we freeze A_i by changing its status from `active` to `unavailable`, thus stopping A_i from admitting more `StartTxn` requests. Any `StartTxn` request sent to A_i before this action will successfully get its `TxnContext`, be executed, and committed. The ones sent after will get an *ActorUnavailable* exception indicating that A_i will be migrated soon, and the request should try again later. Second (line 10 - 13), all relay actors in the whole `X` cluster are informed to freeze A_i by postponing `NewTxn` requests that will access A_i. At the same time, *max_lbid* and *max_gbid* are collected from the relay actors, representing the last local and global batches that will be or have been sent to A_i before starting the migration.

Afterwards, the migration helper waits for A_i to commit the batch with $lbid = max\_lbid$ and the batch with $gbid = max\_gbid$. So far, A_i has completed all the PACTs that are scheduled on it, and it is guaranteed no more PACTs will access A_i before the migration. Next, A_i's latest committed state is returned (line 15). Then, the actor placement directory is updated (line 16), and A_i is deactivated (line 17). A_i cannot accept any transactional method invocations until its state is restored and its status is set as `active` (line 19). Besides, all relay actors should also unfreeze A_i (lines 20 - 21) to resume the process of postponed `NewTxn` requests.

4.3.3 Integrating with Hybrid Processing

Similarly, a boundary is set for ACTs as well, such that ACTs that have been added into A_i's local schedule before the boundary can be completed before the deactivation. More specifically, once A_i is set as `unavailable` (line 8), it stops accepting more ACTs by throwing *ActorUnavailable* exceptions to all `StartTxn` requests or transactional invocations of ACTs; thus, no more ACTs will be added to A_i's local schedule. Besides, before A_i is deactivated (line 17), A_i should wait for all ACTs in the current local schedule to be completed. Those ACTs are once executed on A_i, and their 2PC-related messages are expected to be delivered to A_i in the near future; thus, we allow these ACTs to complete their 2PC process.

4.4 Multi-Node Evaluation

In this section, we evaluate X on the multi-node architecture. We first evaluate the effect of two optimizations applied for the hierarchical architecture (§ 4.4.2). Then, we investigate the performance of PACT, ACT, and hybrid execution with workloads mixing both distributed and non-distributed transactions. In § 4.4.3, we compare PACT and ACT execution by varying the percentage of distributed transactions (§ 4.4.3.1), the number of worker nodes accessed by each transaction (§ 4.4.3.2), and the workload skew levels (§ 4.4.3.3). § 4.4.4 focuses on hybrid execution, presenting its throughput, latency, and transaction abort rate under multiple workload settings. In § 4.4.5, we validate X's horizontal scalability. Lastly, an actor migration experiment is conducted (§ 4.4.6) to evaluate the effectiveness of the proposed actor migration method.

4.4.1 Experimental Settings

4.4.1.1 Deployment

Master	**# vCPU**	**# GC**	**minimum global batch size (ms)**
	N	N	$b \times 1.1^{(\log_2 N)-1}$ (b: a tunable parameter)

Worker	**# vCPU**	**# Worker**	**# LC**	**# actor**	**local batch size (ms)**
	4	N	8	10000	depends on token passing speed

Client	**# vCPU**	**# Client**	**# client thread**
	4	N	1 for PACT + 1 for ACT

(a) Scalability setup (scale factor: N)

pipeline size	Non-Transactional (NT)	128				
	PACT					
	ACT	128	8			
	OrleansTxn					
	hybrid	128 + 128	128 + 8			
workload skewness		uniform	low skew ⟶ high skew			
(*hot%*)		(100%)	(2%)	(1%)	(0.50%)	(0.20%)

(b) Pipeline size per client thread

Figure 4.9: Experimental setting

In this section, X is run on Orleans 3.6.5. In each experiment, four types of processes are spawned: `Controller`, `Client`, `Worker` and `Master`. Each process is deployed on an AWS EC2 instance (c5n) and affiliated with a different number of vCPUs. All of the instances are located in the same region and availability zone. The `Controller` initiates X's environment and orchestrates all `Client`s to start and end experiments. On each `Client` instance, the same

as described in § 3.4.1.2, multiple threads pull transaction requests from a push-pull queue, forward to `Workers`, and collect metrics data. The `Worker` and `Master` refer to the worker and master nodes (§ 4.2.2), respectively.

As for the scalability experiment (§ 4.4.5), while varying the number of `Workers`, the number of `Clients`, and the configurations of the `Master` instance change accordingly. As is shown in Fig.4.9a, the number of vCPUs and the number of global coordinators ($\#GC$) vary with the scale factor N. In addition, the minimum size of each global batch is controlled by a parameter b, which can be tuned to improve system throughput (§ 4.2.3.3). On each `Client`, the pipeline size for each type of client thread is shown in Fig.4.9b.

4.4.1.2 Workload Generation

In this section, in addition to $txnSize$ and $PACT\%$, we configure a workload with three extra variables: $txnDistLevel$, $dist\%$, and $hot\%$. $txnDistLevel$ is the number of `Workers` a distributed transaction accesses. For a transaction with $txnSize = 8$ and $txnDistLevel = 4$, two actors are selected from each of the `Workers`. $dist\%$ indicates the percentage of distributed transactions in the whole workload, while $hot\%$ represents the size of the hot set among actors on each worker. When $hot\% = 2\%$, on each `Worker`, 2% of the actors form the hot set. Thus, a smaller $hot\%$ indicates a smaller hot set and higher workload skew level. Besides, every transaction has half of the actors selected from the hot set. For instance, for a transaction with $txnSize = 4$ and $txnDistLevel = 2$, suppose two actors are selected from `Worker` 0 (W_0) and two from `Worker` 1 (W_1). Among the two actors in W_0, one is from the hot set of W_0, and the other is not. The same goes for the two actors in W_1. Differently from § 3.4.2.2 where the workload skew level is configured using the `zipfian` function, here, the hotspot mechanism is applied for better controlling the contention level in the scalability experiment.

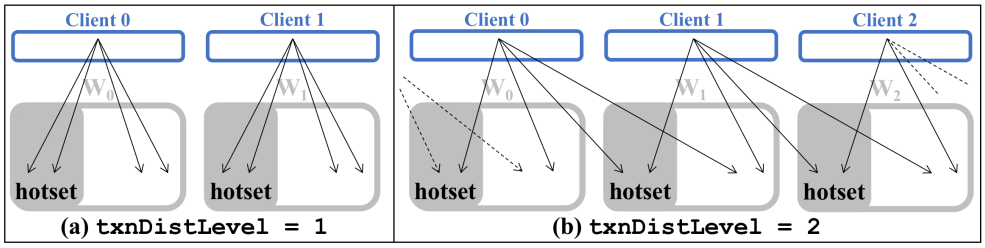


Figure 4.10: Workload generation

In all experiments, actors are uniformly distributed on multiple `Workers`. To balance the workload on each `Worker`, each `Client` only generates transactions that access a specific set of `Workers` – the `Client` i is mapped to `Worker` i and $(i+1)\%N$ (N is the total number of `Worker`). Fig.4.10 shows how to generate a workload with different $txnDistLevel$ when $txnSize = 4$.

The arrow indicates an actor is selected from the corresponding set of actors on the `Worker`.

4.4.2 Effect of optimizations

In this section, we evaluate how the system performance is improved by the two optimizations proposed for the PACT execution in the hierarchical architecture. One is applied to speed up the commit of global batches (§ 4.2.3.4). The other is to tune the size of global batches (§ 4.2.3.3). To thoroughly investigate the effects of these two optimizations, we run experiments under two different setups. In Fig.4.11a, we set $txnSize = 4$, $txnDistLevel = 2$, $\#Worker = 2$. In Fig.4.11b, we set $txnSize = 16$, $txnDistLevel = 8$, $\#Worker = 8$. In both figures, we fix $PACT\% = 100\%$, $hot\% = 100\%$ and vary $dist\%$. Note that when $dist\% = 0\%$, $txnDistLevel = 1$.

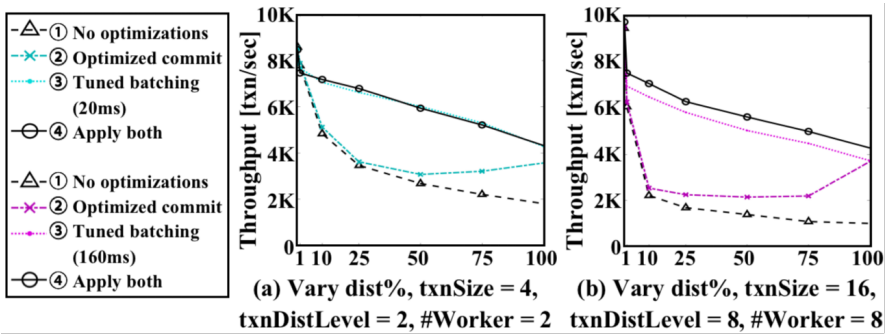


Figure 4.11: Effect of optimizations

The throughput of ① drops significantly fast from $dist\% = 0\%$ to 10% in both figures, showing the influence of global PACTs that they enforce synchronization between different `Worker`s and slow down the progress of local PACTs. In addition, ① decreases much faster in Fig.4.11b than 4.11a because each global PACT in 4.11b involves more number of `Worker`s and exacerbates the synchronization latency. When $dist\%$ grows to 100%, ① drops 78% in Fig.4.11a and 90% in 4.11b compared to the values of $dist\% = 0\%$, respectively.

The throughput of ② presents the effect of the optimization where the commit of a global batch can be sped up in a `Worker` if its previous one is also a global batch (§ 4.2.3.4). In both figures, ② outperforms ① when $dist\% > 10\%$, and ② even starts to increase when $dist\% > 50\%$. This is because the possibility of taking the shortcut increases when there are more global PACTs. In addition, this optimization performs more effectively for a larger $txnDistLevel$. When $dist\% = 100\%$, ② increases 2x from ① in Fig.4.11a, and 4x in 4.11b. This is because a larger $txnDistLevel$ indicates more `Worker`s are involved in a global batch, and more `GlobalBatchCommit`

messages are needed when committing the batch. Applying this optimization helps to remove these messages from the critical path of committing the batch.

The throughput of ③ shows the effect of tuned batching where the minimum size of each global batch is bounded – 20ms for Fig.4.11a and 160ms for 4.11b. These numbers are tuned so as to benefit the total throughput the most. In both figures, ③ decreases smoothly and much slower than ① and ② with increasing *dist*%. It indicates that a proper batch size can better amortize the overhead of logging and messaging, as well as reduce the blocking between different `Workers` due to the synchronization of global batches.

The throughput of ④ shows the case of the combined effect of both optimizations. ④ remains the same as ③ in Fig.4.11a, while outperforming ③ in 4.11b. Again, it indicates that the optimized commit process is more advantageous for a workload with larger *txnDistLevel*. In the rest of the experiments, both optimizations are adopted.

4.4.3 PACT vs. ACT Execution

In this section, we investigate the characteristics of PACT and ACT executions in the existence of distributed transactions.

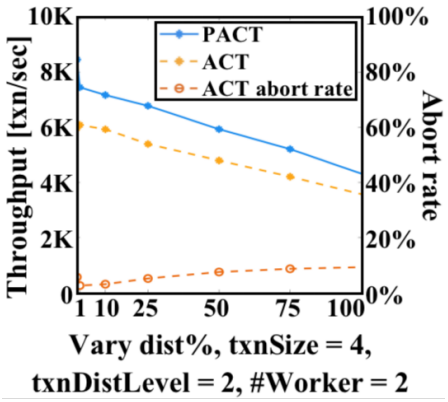


Figure 4.12: Vary *dist*%

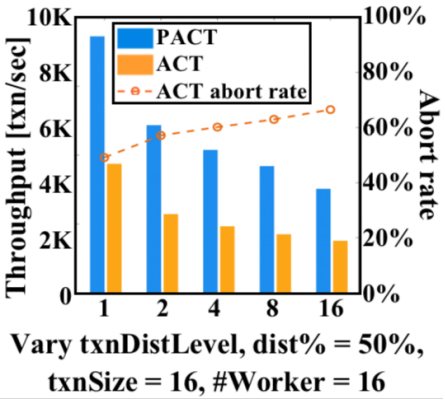


Figure 4.13: Vary *txnDistLevel*

4.4.3.1 Vary *dist*%

This experiment is run with two `Workers` and we set $txnSize = 4$, $txnDistLevel = 2$, $hot\% = 100\%$, with varying *dist*%. Fig.4.12 shows that ACT throughput decreases and abort rate increases with growing *dist*%. ACTs crossing multiple `Workers` have longer latency and hold locks for a longer time, thus leading to more contention and more aborts.

Besides, ACT throughput is always lower than PACT. Recall that the overhead of processing ACT is higher than PACT according to the experiment on the single-node architecture (§ 3.4.2.1) because every ACT requires

two round-trip messages and two blocking IOs to perform 2PC, while PACTs are committed and logged in batches, which is more efficient. Here, when transactions access two `Worker`s, although both PACT and ACT are affected by more costly inter-node messages compared to intra-node communication, we conjecture that the batching mechanism, combined with the two optimizations, helps PACTs outperform ACTs.

4.4.3.2 Vary *txnDistLevel*

In this experiment, we deploy 16 `Worker`s, set $txnSize = 16$, $dist\% = 50\%$, $hot\% = 100\%$, and vary $txnDistLevel$. Note that when $txnDistLevel = 1$, $dist\% = 0\%$. Fig.4.13 shows that the throughput of both PACT and ACT decreases significantly when $txnDistLevel$ changes from 1 to 2, and decreases at a slower pace when $txnDistLevel$ goes higher. From $txnDistLevel = 1$ to 2, the advent of inter-node communications leads to significant throughput degradation for both PACT and ACT. When $txnDistLevel$ becomes larger, for ACT, the more `Worker`s a transaction accesses, the more number of cross-node calls are invoked in both transaction execution and commit phases. Thus, the longer transaction latency becomes, and again, the longer time locks are held. Due to the higher contention, ACT gets a higher abort rate and lower throughput. For PACT, when a transaction accesses more `Wokers`, its corresponding global batch is divided into a larger number of sub-batches, which also means more messages (`GlobalSubBatch`, `GlobalBatchComplete` and `GlobalBatchCommit`), and accordingly more logging, are needed. However, despite both PACT and ACT throughput decreasing with increasing $txnDistLelve$, PACT outperforms ACT throughout the time.

4.4.3.3 Vary *hot%*

In this experiment, two `Worker`s are deployed, $txnSize = 4$, $txnDistLevel = 2$. Under this setting, we compare the throughput of PACT, ACT and `OrleansTxn` with varying workload $hot\%$ when $dist\% = 0$, 50%, and 100% respectively. Fig.4.14 presents the results. All three groups have a decreasing trend when $dist\%$ increases, which shows the overhead of distributed transactions. Within each group, when $hot\%$ decreases, the workload skew level increases. For PACT, its throughput rarely changes with varying $hot\%$ because PACT is not sensitive to contention, given that all PACTs are pre-ordered and executed in a deterministic way. Besides, PACT may even benefit from high contention because batching becomes more efficient. For instance, when $dist\% = 100\%$, PACT throughput slightly increases when $hot\%$ decreases. By contrast, the throughput of ACT and `OrleansTxn` drops because they suffer from high contention, while an increasing abort rate is observed. In addition, the throughput of `OrleansTxn` is always lower than ACT, even with a deadlock-free work-

load. This is a similar result to the experiment in § 3.4.2.2 and § 3.4.2.3 where `OrleansTxn` has higher overhead on different operations.

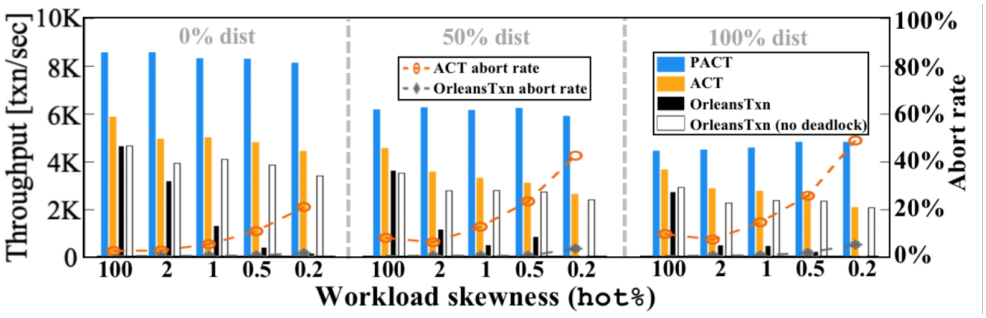


Figure 4.14: Effect of workload skewness

4.4.4 Performance of Hybrid Execution

In this section, we deploy two `Workers`, set $txnSize = 4$, $txnDistLevel = 2$, and vary $dist\%$, $hot\%$, and $PACT\%$. As is shown in Fig.4.15, within a group with fixed $dist\%$ and $hot\%$, the total throughput mostly smoothly decreases from 100% to 0% PACT, which validates that the hybrid execution can bridge the gap between pure PACT and pure ACT, even with distributed transactions.

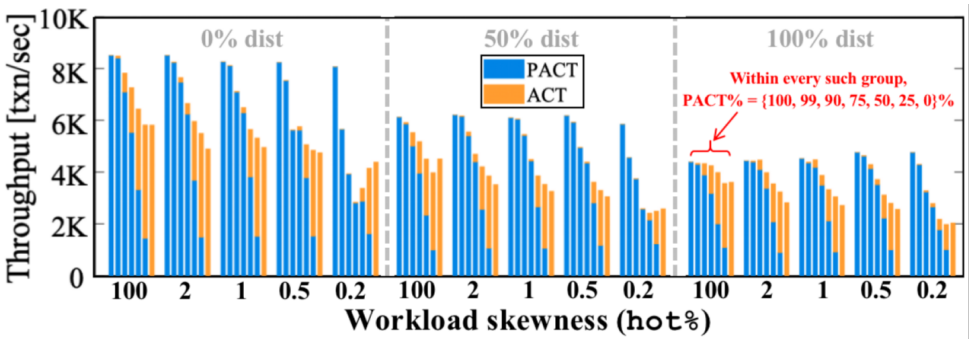


Figure 4.15: Hybrid execution

In addition, similarly to Fig.3.17 (§ 3.4.3.1), the total throughput may experience a significant decrease from 100% to 99% PACT when the workload skew level is very high ($hot\% = 0.2\%$). Again, this shows how the blocking between PACT and ACT scheduling is exacerbated under a highly content workload, given that an ACT has to wait for the previous batch to complete while a PACT has to wait for the previous ACT to commit or abort. Another interesting finding is that this throughput degradation is alleviated when $dist\%$ becomes higher. More specifically, when $hot\% = 0.2\%$, $dist\% = 0\%$, 50%, and 100%, the total throughput drops 30%, 22% and 10% respectively.

This is because, with larger *dist*%, more PACTs need to be processed as global PACTs, which results in more time being spent on the batching and commit phases and, thus, less sensitivity to the time these PACTs are blocked by ACT processing.

4.4.5 Scalability

In this section, we evaluate the horizontal scalability of `X` using both `SmallBank` and `TPC-C` benchmarks with varying *PACT*%, *dist*%, and *hot*%.

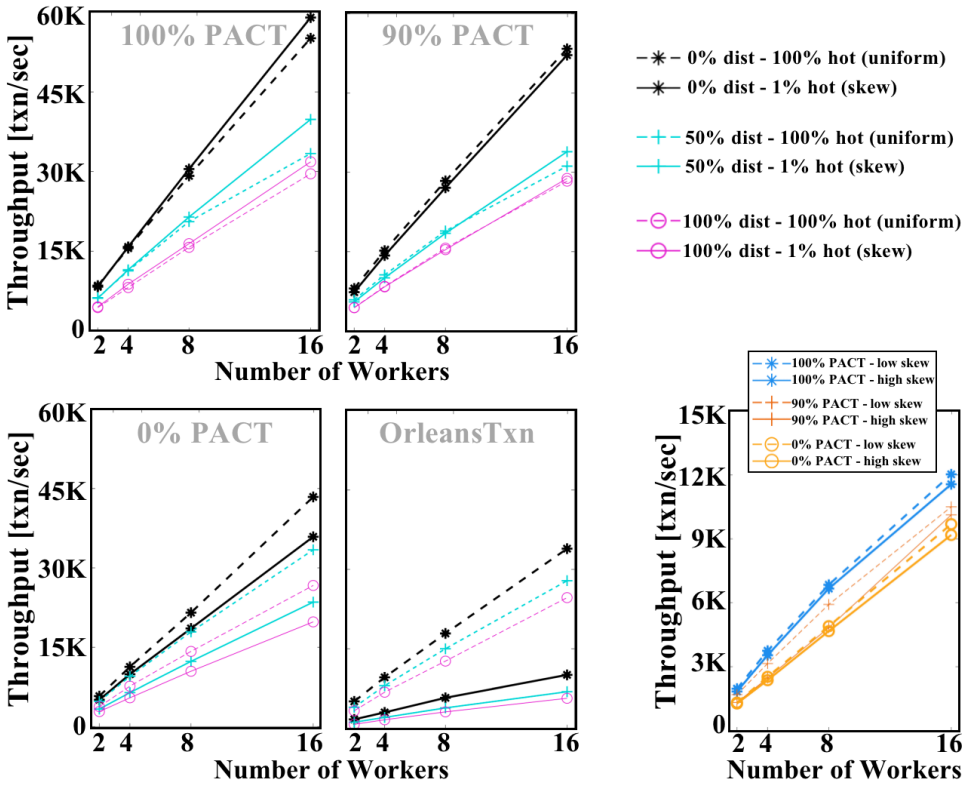


Figure 4.16: Scale out

Fig.4.16a-d shows the results with the `SmallBank` benchmark. All lines scale up linearly from 2 to 16 `Workers` (64 CPUs in total), and the throughput increases ∼ 1.8-2x when $\#Worker$ doubles. In each figure, a higher *dist*% results in lower throughput due to the overhead introduced by distributed transactions. Among the first three figures (Fig.4.16a-c), *PACT*% decreases and the total throughput decreases as well. In addition, the gap between *hot*% = 100% (dotted lines) and *hot*% = 1% (solid lines) is widening for each fixed *dist*%. This effect indicates that the impact of contention becomes more severe when there are more ACTs in the system. If we further look into

Fig.4.16d, the gap is much larger for `OrleansTxn`. When $hot\% = 100\%$, ACT has a slightly higher throughput than `OrleansTxn`, while under higher contention where $hot\% = 1\%$, ACT performs significantly better. The relatively inferior ability of `OrleansTxn` to deal with contention has also been observed and discussed on several occasions [34, 98, 36].

Fig.4.16e shows the results with the `TPCC` benchmark. We implement the `NewOrder` transactions as in § 3.4.4.2 such that each of such transactions accesses 15 actors on average. The mapping between different tables to actors is illustrated in Fig.3.21. In this experiment, each `Worker` contains two warehouses. Furthermore, 10% of transactions will access warehouses located on another `Worker`, thus $dist\% = 10\%$ and $txnDistLevel = 2$ for transactions accessing remote warehouses. In Fig.4.16e, all lines scale near linearly up to 16 `Workers`, and the higher the $PACT\%$, the higher the throughput that is observed, as expected.

4.4.6 Actor Migration

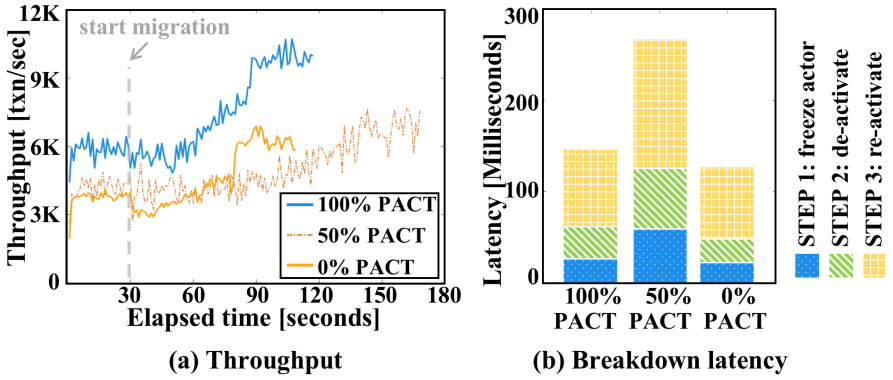


Figure 4.17: Actor Migration

In this experiment, we simulate a scenario in which an overloaded system achieves better performance by adding more nodes and migrating actors for load balancing. To set up this case, we first initiate two `Workers`, each containing 10k actors, 8 migration helper actors and 8 relay actors. We deploy four `Clients` to fill the system with excessive workloads. Each `Client` spawns two client threads to submit PACT and ACT transactional requests (pipeline size is 256 and 64, respectively). After running for 30 seconds, two more `Workers` are added to the cluster, and each `Client` spawns a new thread to emit actor migration tasks (pipeline size is 8). In the end, each `Worker` will contain 5k actors. When all the migration tasks are done, the system keeps running for another 30 seconds to reach a stable final throughput. As for the transaction workload, we set $hot\% = 100\%$, $txnSize = 4$, $txnDistLevel = 2$, and $dist\% = 100\%$ both before and after the actor migration. Therefore, if

we double the number of Workers; ideally, the total throughput should also be doubled.

Fig.4.17a presents the system throughput measured per second during the whole process. For all three cases ($PACT\% = 100\%, 50\%, 0\%$), the throughput grows gradually, with fluctuations but no significant drop, and ends $\sim$ 1.7x higher than before. Besides, according to the statistics we collect, only $\sim 0.01\%$ transactions are aborted due to the migration. Fig.4.17b shows the average breakdown latency for migrating an actor. In X, since an actor can only be migrated when a certain PACT batch or a certain ACT is completed, the time spent on actor migration is expected to be largely influenced by transaction latency. Besides, the actor migration tasks are carried out on the migration helper actors and scheduled by the actor runtime with other transaction-related actor calls. Thus, the more CPUs consumed by transaction processing, the more time the actor migration task will wait to be scheduled. In Fig.4.17b, the workload with 0% PACT has the lowest actor migration latency, and the workload with 50% PACT has the highest latency in all three steps. The latency also matches the total time to finish the migration in Fig.4.17a.

4.5 Related Work

4.5.1 Global and Local Scheduling

In SnapperD, we use a group of global coordinators and a group of local coordinators within each node to schedule distributed and non-distributed transactions, respectively. Several deterministic database systems also adopt this idea of global and local scheduling. VoltDB [165] uses a node controller to process transactions within a partition and a global controller to process multi-partition transactions in serial. Different from VoltDB, SnapperD supports distributed transactions to be executed concurrently and parallelly. As another example, Granola [43] uses timestamps to determine the order between transactions. A non-distributed transaction gets a local timestamp directly from the local node, while a distributed transaction gets a global timestamp via coordination among all involved nodes. In contrast, SnapperD generates *gtid*s in the global node, and the global order is later delivered to local nodes in batches. In addition, SLOG [145] is a partitioned and geo-replicated deterministic database system. SLOG applies global ordering for multi-home transactions, the ones whose master copies of accessed items are located in multiple regions, and SLOG applies local ordering for single-home transactions. However, unlike SnapperD, SLOG does not further distinguish between transactions that span single and multiple nodes, either within or across regions.

4.5.2 Assumptions about Determinism

In SnapperD, we assume that the list of actors a transaction will access is given in advance. Similarly, DDBS usually assume a known read/write set for each transaction. With such information, we are able to apply deterministic transaction processing techniques, which avoid deadlocks, eliminate conflict aborts and give chances to simplify the commit protocol. Differently, most DDBS, such as Calvin [172], TPart [181], QStore [137], and Granola [43], further assume deterministic transaction logic where the re-execution of the same transaction will generate the same results. With this feature, these DDBS commit transactions by simply logging the ordered sequence of transaction inputs because failed transactions can always be recovered by a replay. SnapperD does not rely on this assumption and SnapperD commits a batch of transactions by coordinating all of the involved actors. In addition, some DDBS also make assumptions about user aborts. For example, PWV [64] assumes that a transaction does not abort after the pre-declared *commit point* in the transaction workflow. Caracal [139] assumes that user aborts only happen before performing any write. In contrast, SnapperD puts no constraints on user aborts. Apparently, different assumptions make it possible to apply different optimizations. SnapperD has the potential to further improve its performance by adding more assumptions and we will leave it as future work.

4.5.3 Distributed and Non-Distributed Transactions

Optimizing distributed transactions is a topic that continuously draws attention from industry and academia. While most of them focus on improving distributed transactions, achieving good performance for non-distributed transactions in the distributed environment is rather important. Many of the existing methods, such as deterministic techniques, variants of 2PC [102, 55, 54] and some other commit protocols [167, 187], improve the performance of distributed transactions and benefit non-distributed transactions as well. This is because non-distributed transactions scheduled after distributed transactions are blocked for less time if distributed transactions are executed more efficiently. However, some methods are beneficial for distributed transactions but less conducive for non-distributed transactions. For example, Primo [85] simplifies 2PC by acquiring exclusive locks for all read items which can reduce the performance of non-distributed transactions. In SnapperD, the optimisations applied for batching and commit protocol both aim to reduce the impact of distributed transactions on non-distributed transactions.

4.5.4 Actor State Persistence

Actor systems usually follow the deactivate-then-reactivate pattern to migrate actors. In the reactivation phase, the state of the actor is restored in different ways depending on how it is persisted. In general, there are three types of

persistence. First, in Akka [5] and Riker [147], the actor state persists as a sequence of historical events and is restored via event sourcing, which is efficient for writes but time-consuming for reads. Second, in Dapr [46], Orbit [118] and Pykka [135], the actor state persists as an object and only the latest version is stored. This disallows us from retrieving a certain old version that constitutes a globally consistent snapshot. There are also actor systems, such as Orleans [122] and Proto.Actor [131], supporting both mechanisms. Third, under the context of multi-actor transaction processing, both OrleansTxn [55] and Snapper [99] write transactional logs to guarantee a recoverable state. In their cases, the actor is restored by walking through a failure-recovery algorithm, assuming that the actor can be de-activated and migrated at any time. Different from all of them, SnapperD keeps each actor's latest committed state in memory and delivers it to the newly activated actor instance as a message. SnapperD resorts to this simple and efficient solution because it starts the migration at a proper time such that no more transactions will be executed on this actor.

4.6 Conclusion

This paper presents SnapperD, a high-performance and scalable system for executing distributed actor transactions. SnapperD enables hybrid deterministic and nondeterministic distributed transaction processing. Its hierarchical architecture provides a scalable deterministic transaction ordering and committing service, which not only avoids the potential bottleneck but also minimizes the idling time of non-distribution transactions while waiting for distributed transactions to commit. SnapperD also supports transactional actor migration with minimized impact on the system throughput, facilitating efficient runtime system reconfiguration.

Chapter 5

SnapperX: Fine-Grained Actor State Management

THE actor model has emerged as an effective solution for building stateful middle tiers. By promoting the partitioning of application functionalities and associated states into actors, developers benefit from their inherent concurrency model to achieve scalability and isolation. Despite its increasing popularity, the actor model lacks support for complex state management tasks, such as enforcing foreign key constraints and ensuring data replication consistency across actors. These are crucial properties in partitioned application designs, such as microservices, which inhibit the further adoption of the actor model.

To fill this gap, we start by analyzing the key impediments in state-of-the-art actor systems. We find it difficult for developers to express complex data relationships across actors and reason about the impact of state updates on performance due to opaque state management abstractions. To solve this conundrum, we develop `SnapperX`, a novel data management layer for actor systems, allowing developers to declare data dependencies that cut across actors, including foreign keys, data replications, and other dependencies. `SnapperX` can transparently enforce the declared dependencies, reducing the burden on developers. Furthermore, `SnapperX` employs novel logging and concurrency control algorithms to support transactional maintenance of data dependencies.

We demonstrate `SnapperX` is able to support core data management tasks where dependencies across components appear frequently without jeopardizing application logic expressiveness and performance. Our experiments show `SnapperX` significantly reduces the logging overhead and leads to increased concurrency level, improving by up to 2X the performance of state-of-the-art deterministic scheduling approaches. As a result, `SnapperX` will make it easier to design and implement highly partitioned and distributed applications.

5.1 Introduction

In the modern application programming landscape, developers design their applications in tiered architectures with a stateless middle tier encoding the application logic and a database tier storing the application state [29]. For each client request, the middle tier executes the business logic by retrieving the necessary data from the database tier. This architecture simplifies the development of applications by pushing complex data management tasks to the database tier. However, this data shipping paradigm has limitations. The long end-to-end latency and excessive data transfer from the database to the middle tier during peak periods may not meet the applications' requirements [30, 154]. In-memory data caching in the middle tier can reduce the latency and data transfer. However, it can suffer from low cache hit ratios, e.g., when different client requests need to access diverse data or data are updated frequently, and cache data inconsistency that can lead to application safety problems [127].

To address these problems, an alternative architecture employs a stateful middle tier, where data are stored in the computing nodes, and the function shipping paradigm is adopted. Client requests are shipped to the computing nodes, which store the corresponding data for processing. This does not involve data transfer from the data storage, minimizing the bandwidth overhead and latency. Data from the middle tier are asynchronously shipped to the database layer for analytics and disaster recovery. Furthermore, to achieve system scalability, software agility, and fault isolation, we are witnessing the emergence of microservice architectures [89]. In such architectures, the application is decomposed into independent and fine-grained components that interact via synchronous or asynchronous communications, each encapsulating its own state.

Meanwhile, the actor model [4] has emerged as a promising concurrent programming model for middle-tier development [10]. The encapsulated states of actors allow developers to build loosely coupled applications with a design that remounts a cache with data locality, characterizing the function shipping paradigm. Client requests are processed by one or more actors interacting via asynchronous messages, triggering the updates of their encapsulated actor states. Such design principle makes it very attractive to model microservices as actors [87]. In addition, the advent of virtual actor [29], originally developed in the context of Microsoft Orleans, further alleviates developers' burden by providing actor state management functionalities, including state persistence, transactional state manipulation, and fault tolerance via state persistence, which lays a solid foundation for addressing the data management challenges in microservice systems [89]. The state of a virtual actor is modeled as an opaque transactional object, no matter how many entities are encapsulated in an actor. Such an opaque state model puts almost no limitation on how developers implement the operations on actor states other than using the

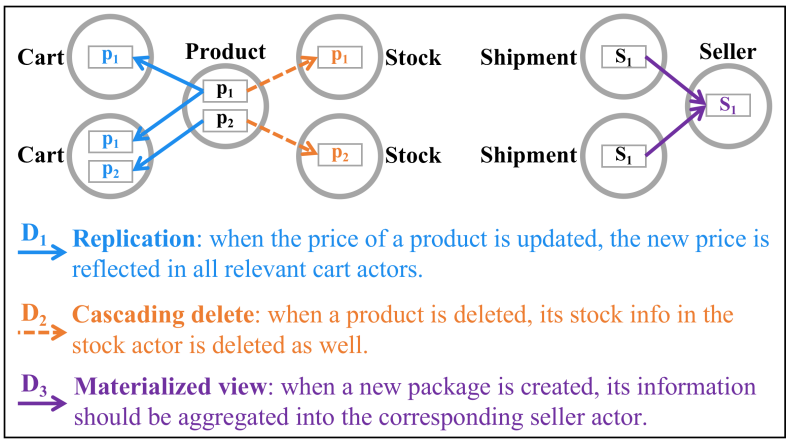


Figure 5.1: Cross-actor relations in `Online MarketPlace`

required APIs to interact with an underlying data store for persistence. Thus, an update in a single entity is treated as an update in the object as a whole.

Despite the recent advancement of actor state management, modern data-intensive applications exhibit complex relationships among entities cutting across distributed components [176], which typically are not natively supported by actor runtimes. These cross-component relationships include foreign-key constraints, replication of data items, and functional dependencies [89]. Taking an e-commerce application case, `Online Marketplace` [88], as an example (Fig.5.1), product data managed by the product component can be replicated to the cart component, which manages products added to customer carts. The replication favors performance since it avoids successive round trips between the cart and product components for ensuring the correctness of product information (e.g., product price) on checkout time [176, 140]. However, the opaque state model of virtual actors exposes little semantic information, such as how and which part of the state is modified, thereby limiting the ability of developers to express important safety properties of data-intensive applications. Due to the lack of support from the actor frameworks, developers must map relationships across actors explicitly (e.g., which carts contain a certain product) and ensure their correctness in application code. Besides being complex and error-prone, this practice is oblivious to transaction management, leading to isolation anomalies.

Taken together, enhancing the state management features of actor frameworks for ensuring application safety across actors is a missing key to fully realizing the envisioned benefits of the actor model in developing scalable stateful middle tiers. To bridge the gap, we develop a data management library for virtual actors, which provides an API for developers to register cross-actor dependency constraints dynamically and supports high-throughput transactional enforcement of these constraints. It accounts for emerging cross-component state management requirements that arise in partitioned and distributed applica-

tions, such as microservices [89, 83, 176]. Our contributions are summarized as follows:

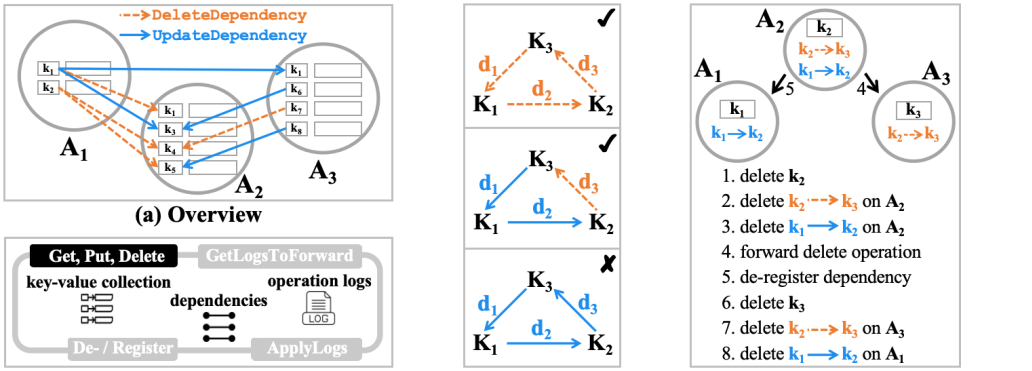
- To strike a balance between simplicity, state translucency, and expressiveness, we propose an extended key-value model for actor states with an associated set of state access and dependency APIs. This design enables the shift of cross-actor dependency management from the application codes to the actor framework and hence unloads the burden of developers from complex and error-prone state management codes.
- We develop `SnapperX`, a data management layer in a virtual actor framework. `SnapperX` is implemented by integrating our data model and APIs with the state-of-the-art transaction library for virtual actors – `Snapper` to support transactional data dependency enforcement. To maximize the transaction throughput, we extend `Snapper`'s concurrency control to take advantage of the new data model of actor states, providing fine-grained concurrency control outperforming vanilla `Snapper`.
- We map the `Online Marketplace` [88] benchmark to the actor abstraction and implement it on `SnapperX`. `Online Marketplace` models key safety properties related to relationships across components that arise in real-world partitioned and distributed applications. Our implementation validates the expressiveness and ease of use of our data model, as well as addresses the challenges of cache coherence, referential integrity, and strong isolation transactional guarantee.
- We conduct extensive experiments on two benchmarks, `Online Marketplace` and `SmallBank`. Our experiment demonstrates the efficiency of fine-grained concurrency control over state-of-the-art deterministic methods. As a result, `SnapperX` will facilitate the design and implementation of actor-based stateful middle-tiers.

5.2 Actor State Management for Data Integrity

In this section, we present `SmSa`, a data model we introduce for the actors, which includes novel concepts of the actor state and data dependencies cut across actors.

5.2.1 Conceptual Overview of `SmSa`

In `SmSa`, each actor's state is modeled as a key-value collection. `SmSa` relies on developers to determine how keys are partitioned across actors. Each key is unique among those in an actor, while the same key may exist in different actors. Keys located in different actors are related through a dependency constraint (§ 5.2.2). As shown in Fig.5.2a, actors and dependencies form a



`IValue? Get (IKey k)`	retrieve a key on the current actor
`bool Put (IKey k, IValue v)`	add or modify a key-value pair
`bool Delete (IKey k)`	delete a key on the current actor

```
DependencyType: DeleteDependency, UpdateDependency
Dependency: <DependencyType d, ActorRef A1, Key k1, ActorRef Af, IKey kf, IFunc? func>

OperationType: Add, Modify, Delete
OperationLog: <OperationType o, IKey k, IValue? old_v, IValue? new_v>

IValue IFunc.ApplyUpdate(IKey k1, IValue old_v1, IValue new_v1, IKey kf, IValue old_vf);
```

(e) Programming abstractions

Figure 5.2: State management of stateful actors (SmSa)

directed graph. The arrow $k_i \rightarrow k_j$ represents that k_j depends on k_i, and a change made on k_i may cause a change to k_j. SmSa supports basic operations to get, put, and delete keys; meanwhile, it allows dynamic registration and de-registration of dependencies between two specific pairs of key-value items in two different actors. Thus, the dependency graph may change with time, well reflecting the dynamic nature of actor topologies [1]. Besides, we opt for a key-value abstraction because developers are familiar with this model to manage data in stateful middle-tiers [127, 109].

The dependencies in SmSa are used to model different application constraints, such as data integrity constraints, foreign key constraints, data replication, and other functional dependencies. For example, Fig.5.1 illustrates the replicated key ($\mathbf{D}_1$), the foreign key with cascading delete ($\mathbf{D}_2$), and the materialized view ($\mathbf{D}_3$), respectively. A prominent feature of SmSa is the system-level support to enforce these constraints instead of relying on developers to encode them case by case in the application logic. More specifically, SmSa keeps track of the operations performed on the keys during the execution of the actor methods. When the execution is done, SmSa scans the list of operation logs and figures out the operations that need to be forwarded to the other keys on the other actors for preserving the relevant dependency constraints. For example, in Fig.5.1, the dependency $\mathbf{D}_1$ represents a scenario

when the price of a product is changed, the update operation is forwarded to all cart actors that contain this product, thus guaranteeing that all the cart actors consistently see the latest price. With `SmSa`, the forwarded update operations are calculated and carried out transparently at the system level. § 5.2.2 discusses the algorithm applied to derive such information and how operations are forwarded and applied to another actor.

Fig.5.2b illustrates that each actor maintains not only a key-value collection but also a list of dependencies and a list of operation logs. `SmSa` wraps them together into a `DictionaryState` object. This object exposes two sets of APIs; one is external for the user request to access application data, and the other is internal for the system to enforce the dependency constraints. By enforcing certain operations being invoked via these APIs, `SmSa` captures all the changes made to the key-value collection of an actor and further automates the dependency constraint enforcement.

5.2.2 Dependency Management

5.2.2.1 Definition

`SmSa` defines a dependency record with six data fields (Fig.5.2e): the type of the dependency, the leader actor, the leader key, the follower actor, the follower key, and a customized function. `SmSa` generalizes two types of dependencies – delete dependency and update dependency. The **delete dependency** describes the relation between two keys (k_l, k_f) on two different actors (A_l, A_f) that the deletion of k_l on A_l will cause the deletion of k_f on A_f. The **update dependency** refers to the relation in which the update that happened to k_l on A_l will trigger a specified update function being applied to the key k_f on A_f. In an update dependency, the deletion of k_l will not cause the deletion of k_f. A key where the delete or update operation originates is named **leader key**, the affected key is called **follower key**, and the actors where the keys are located are referred to as **leader actor** and **follower actor** respectively. The leader key and the follower key do not have to be the same, depending on how users define the dependency. To be able to retrieve the follower keys of a leader key, `SmSa` stores the dependency information on the leader actor. `SmSa` does not maintain a shared global dependency graph; instead, it distributes the information across actors. By doing so, `SmSa` avoids a centralized component becoming a bottleneck when it needs to serve frequent queries, meanwhile guaranteeing that each actor has sufficient information stored in the local private state to enforce the dependencies.

In addition, an update function needs to be specified for every update dependency. `SmSa` defines an interface $IFunc$ and an abstract method $Apply-Update$ (Fig.5.2e) that each update function should implement. This method requires input: the leader key, the values of the leader key before and after applying the user-invoked update operation, the follower key, and the existing

value of the follower key. Then, the method returns the calculated new value for the follower key. As is shown in Fig.5.3 (lines 79-84), data replication can be facilitated by using an update function that directly returns the value of the leader key. Thus, the follower key remains the same as the leader. This function can be used for defining dependency $\mathbf{D}_1$ where the new price of a product is replicated from the product actor to all relevant cart actors (lines 3-17). Similarly, customized functions can contain more complex calculations and be applied to build materialized views. As the example in Fig.5.3 (lines 62-77), a function is defined for a seller actor to maintain a view of all created orders cumulatively.

5.2.2.2 Deletion Rules of Keys

`SmSa` allows user code to delete both leader and follower keys. When a leader key is deleted, the deletion operation is forwarded to the corresponding followers. When a follower key is deleted, it indicates its dependency on the leader key expires; thus, this dependency record should be deleted to avoid further operations being sent to the follower actor (e.g., lines 35-40 in Fig.5.3). In summary, the deletion of a key indicates the deletion of follower keys, as well as the de-registration of all dependencies pointing to and from this key. To retrieve the dependency in the backward direction, i.e., from the follower key to the leader key, `SmSa` keeps a copy of the dependency record on the follower actor as well.

5.2.2.3 Leader and Follower Keys

In `SmSa`, a leader key may have multiple follower keys. It resembles the scenario in an RDBMS where multiple foreign keys refer to one primary key. `SmSa` allows a leader key to impose different effects on different follower keys. For example, in Fig.5.1, a product ID p_1 on the product actor may relate to multiple cart actors through update dependencies ($\mathbf{D}_1$). Meanwhile, p_1 is also related to the same key on the stock actor through a delete dependency ($\mathbf{D}_3$).

`SmSa` allows a follower key to have multiple leader keys, while a foreign key can only reference a single primary key in RDBMS. `SmSa` identifies three scenarios. First, a follower key may have update dependencies on multiple leader keys. For example, as is shown in Fig.5.1, a follower key s_1 on the seller actor may need to aggregate the information of newly created orders from the leader keys on multiple shipment actors. By doing so, the seller actor can maintain an up-to-date materialized view. Second, a follower key may have delete dependencies on multiple leader keys, which models the case that the existence of an entity is dependent upon the existence of other entities. Third, a follower key may be affected by both update and delete operations on different leader keys. Note that `SmSa` does not control the priority or the order of executing operations related to different dependencies. When a delete

```
1  public class CartActor : TransactionalActor, ICartActor{
2
3    public async Task AddItemToCart(TxnContext cxt, ProductID productID){
4      // STEP 1: get the product actor reference
5      var productActorRef = IdMapping.GetProductActor(productID);
6
7      // STEP 2: define an update dependency
8      var dependency = new Dependency(
9        DependencyType.UpdateDependency,              // the dependency type
10       productActorRef, productID, myRef, productID,  // the leader and follower info
11       new ReplicationFunction());                    // the custom function
12
13     // STEP 3: register dependency (D1)                                    D1
14     /* In this step, the dependency is registered, and the key productID is added
15        automatically, with the latest value fetched from the product actor. */
16     await RegisterDependency(cxt, dependency);
17   }
18
19   public async Task DeleteItemsInCart(TxnContext cxt, HashSet<ProductID> items){
20     // STEP 1: get reference to the DictionaryState that belongs to this transaction
21     var state = await GetState(cxt, new Dictionary<IKey, AccessMode>());
22
23     // STEP 2: the following two options are discussed in Section 3.3.3
24
25     // Option 1: acquire access right to keys one by one
26     foreach (var productID in items)
27       await GetState(cxt, new Dictionary<IKey, AccessMode>{{productID, AccessMode.RW}});
28
29     // Option 2: acquire access right to a set of keys all at once
30     var modePerKey = new Dictionary<IKey, AccessMode>();
31     foreach (var productID in items) modePerKey.Add(productID, AccessMode.RW);
32
33     await GetState(cxt, modePerKey);
34
35     // STEP 3: delete keys
36     foreach (var productID in items){
37       /* When the key is deleted, its dependency on the product actor is
38          de-registered, too. */
39       state.Delete(productID);
40     }
41   }
42 }

43 public class SellerActor : TransactionalActor, ISellerActor{
44
45   public async Task Init(TxnContext cxt){
46     // STEP 1: find all shipment actors
47     var list = GetAllShipmentActors();
48     foreach (var shipmentActorRef in list){
49
50       // STEP 2: define an update dependency
51       var dependency = new Dependency(
52         DependencyType.UpdateDependency,     // the dependency type
53         shipmentActorRef, myID, myRef, myID, // leader and follower info
54         new SellerDashboardFunction());      // the custom function
55
56       // STEP 3: register dependency (D3)                                  D3
57       await RegisterDependency(cxt, dependency);
58     }
59   }
60 }
61
62 public class SellerDashboardFunction : IFunc{
63
64   // calculate the new value of the follower key with given info
65   public IValue ApplyUpdate(
66     IKey kl, IValue old_vl, IValue new_vl, IKey kf, IValue old_vf){
67
68       var newPackage = new_vl as PackageInfo;
69       var aggregatedInfo = old_vf as AggregatedOrderInfo;
70
71       // update the materialized view with new package info
72       aggregatedInfo.totalNumOrder.Add(1);
73       aggregatedInfo.totalMoney.Add(newPackage.totalMoney);
74
75       return aggregatedInfo;
76   }
77 }
78
79 public class ReplicationFunction : IFunc{
80   public IValue ApplyUpdate(
81     IKey kl, IValue old_vl, IValue new_vl, IKey kf, IValue old_vf){
82     return new_vl;   // return the new value of the leader key
83   }
84 }
```

Figure 5.3: Example codes of developing `Online Marketplace` on `SnapperX`

operation and an update operation are both forwarded to the same follower key, the final result is the same – the follower key is deleted. However, when two different update operations are forwarded to the same follower key, the result may vary depending on which update operation arrives at the follower actor first or which update function is applied by the follower actor first. Users can resolve this issue by defining commutative update functions. Another solution is to rely on transaction management (§ 5.3) to restrict the order of different operations performed on each actor.

In `SmSa`, when a key acts both as a leader and a follower, the deletion of such a key becomes more complicated. For example, in Fig.5.2d, when the key k_2 is deleted, the actor A_2 first enquiries forwards to send the delete operation to the follower actor A_3, then backwards to de-register the dependency on A_1.

5.2.2.4 Cyclic Dependencies

If dependency registration is not restricted, `SmSa` may create a cyclic chain of dependencies across multiple keys, which will further result in an infinite loop of operation forwarding process. As is shown in Fig.5.2c, the cycle may be formed in three cases. The first case involves only delete dependencies, the second case is a mix of both types of dependencies, and the third one only has update dependencies.

The first two can be processed to completion within limited steps. In the first case, the deletion of the key k_1 will cause the deletion of k_2, then k_3. When the deletion operation is forwarded back to the actor where k_1 is located, the actor will find out that k_1 does not exist in the key-value collection because it has already been deleted. Thus, the deletion will not be performed again, and no more delete operations will be forwarded along the dependency chain. As for the second case, the update of k_3 will be forwarded to k_1 and k_2, and it does not form a cycle. When k_1 (k_3) is deleted, the two connected dependencies d_1 and d_2 (d_3 and d_1) are removed, and no further operations are needed. When k_2 is deleted, k_3, d_2, d_3 and d_1 will be deleted.

In the third case, deleting any of the keys will cause the deletion of two dependencies. However, updating any of the keys will cause the update operation to circulate endlessly in the cycle. The third type of cyclic dependencies can be prohibited by adopting the following three strategies: $\mathbf{S}_1$, `SmSa` does additional checks whenever an update dependency is registered, which may be time-consuming when there exists a very long chain of dependency, or when each key points to many other keys. $\mathbf{S}_2$, `SmSa` can restrict that each actor should not forward operations for a request more than once. For example, if an update is already forwarded from k_1 to k_2, then when k_1 receives the update from k_3, no more updates will be generated. However, this strategy requires actors to store information about different requests, introduces extra semantics, and makes it more complicated to reason about the application

logic. $\mathbf{S}_3$, `SmSa` can disallow a key to becoming both leader and follower, thus eliminating any type of cycle.

5.2.2.5 Dependency Registration and De-registration

In `SmSa`, a dependency registration request should be initiated by the user on the follower actor A_f. A_f first checks if the specified k_f is allowed to be attached with a dependency. For example, `SmSa` may check if it will cause a cyclic chain of update dependency when the strategy S_1 is adopted, or check if k_f is already identified as a leader when S_3 is adopted. If the check is passed, the leader actor A_l is called to continue the registration. A_l will check if the key k_l can be declared as a leader, i.e. if k_l already exists and there is no dependency that has k_l as a follower key. If yes, the dependency is registered on A_l, and the latest value v_l of the leader key k_l is retrieved and returned to A_f. On A_f, if the follower key k_f does not exist yet, v_l is used as the initial value of k_f, and a new key-value pair $< k_f, v_l >$ is inserted. If k_f already exists, the custom function specified in the newly registered dependency is applied by using $old\_v_l = v_l$ and $new\_v_l = v_l$. The above explains how `SmSa` deals with a dependency registration request. Next, we explain how developers should send such a request to `SmSa`. Fig.5.3 shows the example codes of registering dependencies $\mathbf{D}_1$ (lines 3-17) and $\mathbf{D}_3$ (lines 45-59). In both cases, a dependency is defined by specifying the six required data fields. Afterward, the API *RegisterDependency* is called to forward the necessary information to internal `SmSa`. `SmSa` exposes this API for developers to explicitly declare dependencies.

The de-registration of a dependency can be carried out in two ways. First, it can be done in a similar process as dependency registration, where a de-registration request is sent explicitly to the follower actor and then forwarded to the leader actor. Second, the user can also simply delete the follower key (Fig.5.3 lines 35-40). According to the deletion rule of `SmSa` introduced in § 5.2.2.2, when a follower key is deleted, the corresponding dependency records are removed as well.

5.3 Transactional Actor State Management

Having laid out a rich state management abstraction for the actor model, we turn our attention to how to leverage such advancements to optimize key aspects of data systems, logging and concurrency control.

5.3.1 The Dangers of Unordered Operations

Although equipping actors with a data model enables the declaration of complex relationships across actors, developers must still account for the dangers of arbitrary function execution order and their possible impact on ac-

tor states [17]. For instance, a request in `SmSa` may perform operations on multiple actors, especially in the existence of cross-actor dependencies. The concurrent execution of such requests may drive the system to an inconsistent state. For example, in Fig.5.4, suppose an actor A_2 intends to create a replica of key k, whose master copy is stored on A_1. The example illustrates a possible interleaving of two requests; one is a dependency registration request initiated by A_2, and the other performs an update operation on k on A_1. Ideally, the updated value of key k should be reflected to the replica on A_2. However, if the forwarded update operation arrives earlier than the confirmation message, A_2 will not perform the update and results in a replica that is inconsistent with the master copy on A_1.

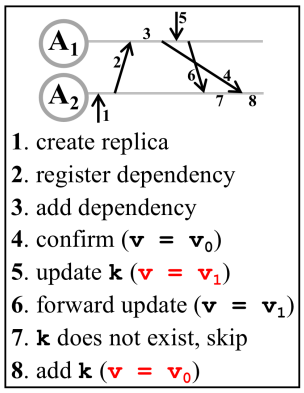


Figure 5.4

`SmSa` can benefit from a transaction management solution to provide a stronger application correctness guarantee. Existing works have proposed a myriad of methods to enforce the order of concurrent tasks [99, 55] and to converge to a consistent state in the presence of disorder [180, 129, 156, 92]. `Snapper` [99], one of these solutions, supports ACID transactional properties for multi-actor operations through performant deterministic concurrency control, making it a good fit to integrate `SmSa` with.

5.3.2 An Actor Transaction Library – `Snapper`

5.3.2.1 Transaction Management

`Snapper` [99] is an actor transaction library designed to enhance the performance of multi-actor transactions meanwhile preserving the ACID transactional properties. `Snapper` leverages deterministic scheduling for PACT, a type of transaction that declares the actor access information when it is submitted to the system. Based on this pre-declared information, `Snapper` employs a group of coordinator actors to generate deterministic transaction execution schedules for every batch of PACTs. Each such schedule consists of a sequence of PACTs that are relevant to a particular actor. Afterwards, `Snapper` delivers such batch schedules to corresponding actors via asynchronous messages. Each actor is supposed to execute transactional invocations of PACTs one by one in the pre-determined order. Under the PACT execution, transactions are scheduled, executed, committed, and logged all at the batch level.

In addition to deterministic execution, `Snapper` also supports conventional non-deterministic strategies, such as Strict Two-Phase Locking (S2PL) and Two-Phase Commit (2PC), for transactions whose actor access information cannot be pre-declared. This type of transaction is called ACT. Another standout feature of Snapper is its ability to execute concurrent hybrid work-

loads, wherein some transactions are executed deterministically while others follow non-deterministic methods. This hybrid execution model maximizes the advantages of deterministic execution, which can achieve high transaction throughput while maintaining flexibility for non-deterministic workloads.

5.3.2.2 Programming Model

`Snapper` provides a base actor class, `TransactionalActor`, that provides key system-level functionalities, such as transaction processing and logging. As listed in Fig.5.5a, the `TransactionalActor` exposes APIs for transaction requests submission (`StartTxn`), for transactional actor state access (`GetState`), and for invoking transactional methods on other actors (`CallActors`).

In `Snapper`, when the `GetState` API is called by a transaction T, the `TransactionalActor` transparently controls when T can get access to the actor state. More specifically, `Snapper` first identifies if T is a PACT or an ACT by using the `TxnContext` information. Then, based on the different concurrency control strategies applied for PACT and ACT, `Snapper` inserts T into the actor's local schedule and waits for the turn to execute T, i.e. grant T access to the actor state. In addition, the `CallActor` API must be used when invoking a transactional call from one actor to another. This API acquires the `TxnContext` as one of the input parameters as well, so as to ensure the actor method invocations are executed under the same transaction context.

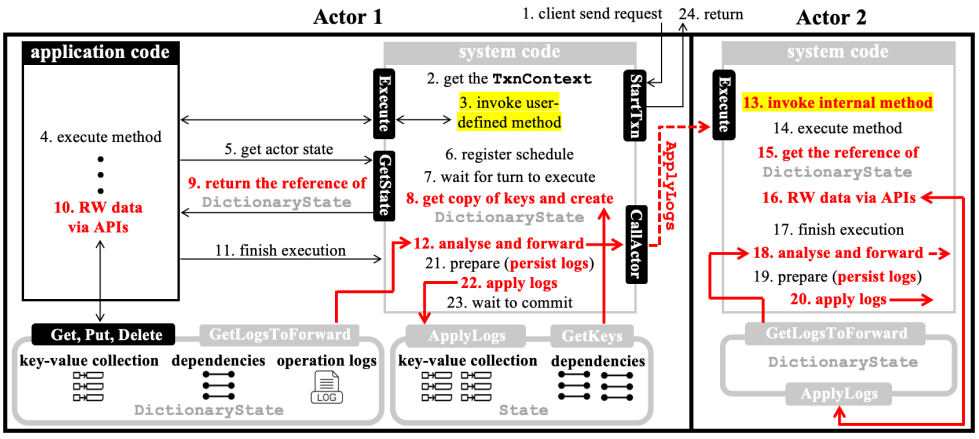
5.3.3 SnapperX: Integration of SmSa and Snapper

In this section, we explain how we advance `Snapper`'s transactional layer to account for `SmSa` and achieve transactional guarantees on dependencies that cut across actors. We start by removing from `Snapper` the opaque state management inherited by Orleans. As actor states are stored as a single object of the generic type, `TState`, we switch to a `SmSa`-managed map-based data structure, referred to `Dictionary- State` along this text. This prevents developers from creating arbitrary types to represent their application states for each actor, jeopardizing application design [114, 79]. In this mode, `SmSa` allows developers to manage entities through keys and associated actor references via the `DictionaryState` and its corresponding APIs (Get, Put, and Delete).

The management of data dependencies in `SmSa` comprises two parts: (1) the registration and de-registration of dependencies and (2) the enforcement of dependencies. To manage dependency constraints transactionally, `SmSa` needs to rely on `Snapper`'s transactional APIs on certain occasions. First, when registering or a dependency, as is discussed in § 5.2.2.5, the follower actor invokes the *RegisterDependency* API, which in turn invokes a call to the leader actor via the `CallActor` API. Since the dependency list is stored in the `DictionaryState` object, adding or removing the dependencies to or from the list must occur by accessing the `DictionaryState` object via the

Snapper	PACT	`Task<object> StartTxn(string startFunc, object funcInput, Dictionary<Guid, int> actorAccessInfo)`
	ACT	`Task<object> StartTxn(string startFunc, object funcInput)`
	PACT & ACT	`Task<object> GetState(TxnContext cxt, AccessMode mode)`
		`Task<object> CallActor(TxnContext cxt, Guid actorID, FuncCall call)`
SnapperX	PACT	`Task<object> StartTxn(string startFunc, object funcInput, Dictionary<Guid, (int, HashSet<IKey>)> keyAccessInfo)`
	PACT & ACT	`Task<IDictionaryState> GetState(TxnContext cxt, Dictionary<IKey, AccessMode> modePerKey)`
		`Task<IValue?> De-/Register(TxnContext cxt, Dependency dep)`

(a) Transactional API



(b) Transaction workflow

Figure 5.5: `SnapperX`: integration of `SmSa` and `Snapper`

`GetState` API. Second, `SmSa` enforces dependencies by resolving and forwarding operation logs to relevant actors; each will apply the forwarded logs and perform corresponding updates to the keys stored locally. Again, keys can only be accessed on the condition of getting access to the `DictionaryState` object by calling the `GetState` API.

5.3.3.1 Workflow

Fig.5.5b presents the workflow of a transaction in `SnapperX` – the `Snapper` integrated with the advancements of `SmSa`. The figure marks the differences as red text compared to `Snapper` and uses the red arrows to represent steps that are relevant to dependency management. Step 8 creates a `DictionaryState` object for the transaction. `SnapperX` does not allow transactions to access the original version of the actor state directly. Instead, a new `DictionaryState` object is created for every transaction so as to isolate their read/write sets. In step 10, the transaction accesses the `DictionaryState` via its external APIs. When the actor finishes executing the user method, the operation logs are scanned to resolve the list of logs to forward (step 12) in order to enforce

user-declared dependencies. Then, an `ApplyLogs` method is invoked on each dependent actor via the `CallActor` API.

The actor who receives this method invocation will identify it as an internal method (step 13), which means it is implemented in the system code. Even if it is an internal method, it still needs to be executed under the transaction context; thus, a reference to the `DictionaryState` object should be acquired via the `GetState` API (step 15). While executing the internal method, the system code has access to the internal APIs of the `DictionaryState` object (step 16), such as `ApplyLogs`, which will read through the forwarded logs and apply updates to keys according to the registered dependencies. When the actor finishes executing the internal method, again, the operation logs need to be analyzed (step 18) because there might be updates performed on keys that trigger more operations to forward to other actors. Afterwards, a transaction log is persisted (step 19), and the logged changes made by the transaction are applied to the actor state (step 20). Note that, the same as in `Snapper`, the log writing happens once for every ACT or every batch of PACTs. As for step 20, given that each transaction has only operated on its own `DictionaryState`, `SnapperX` needs to apply the changes again to the actual actor state when an ACT is committed or a batch of PACT has completed. By doing so, the results of one transaction are made visible to subsequent transactions. In `Snapper`, this step is carried out by overwriting the whole actor state, incurring a higher overhead than `SnapperX`.

Dependency registration and de-registration are implemented as internal methods. Changes made to the dependencies, including adding and removing dependencies, also need to be recorded first, persisted in the log record, and reflected in the actual actor state afterward. Note that adding or removing dependencies does not have a further effect, like an update on a leader key.

5.3.3.2 Incremental Logging

In `Snapper`, the actor state is always logged as a single object due to the lack of information about how the state is changed. We call this logging method as snapshot. It can be inefficient when only a small part of the actor state is modified. Differently, `SnapperX` keeps a record of each change made to keys or dependencies (see the *OperationLog* format in Fig.5.2); thus, `SnapperX` only needs to write an incremental log for each transaction, which can largely reduce the logging overhead.

5.3.3.3 Key-level Concurrency Control for PACT

`Snapper` supports deterministic scheduling of transactions that provides the actor access information – the set of actors to access and the number of times each actor will be accessed. As is listed in Fig.5.5a, the `StartTxn` API for PACT acquires an extra input parameter. `Snapper` uses this information to

generate a deterministic transaction execution order for every related actor. As long as each actor executes PACTs in the ascending order of their transaction ID (*tid*), `Snapper` can guarantee global serializability. By integrating `SmSa`'s state management layer into `Snapper`, we devise a finer-grained transaction scheduling strategy. Given that `SmSa` acquires developers to specify which key to read or write in the application code, we can further assume that the set of keys accessed on each actor by the transaction is declared even before the transaction is started. Based on this key access information, a transaction execution schedule can be created for every single key. As is shown in Fig.5.6, following a key-level schedule, a PACT on an actor only needs to wait for another PACT when the set of keys they access overlaps.

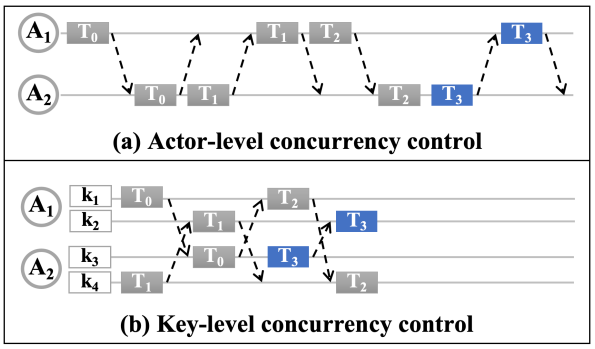


Figure 5.6: Granularity of concurrency control

Apparently, adopting key-level scheduling can achieve higher concurrency on each actor. However, it is not straightforward to implement it. First, a new interface should be created to receive transaction requests that tend to adopt key-level concurrency control. As is shown in Fig.5.5a, we introduce another `StartTxn` API that accepts the key access information as its input when a client submits the transaction request. Besides, the algorithm applied to generate the transaction execution schedule needs to be modified such that metadata is maintained for every single key.

Furthermore, on each actor, it becomes insufficient to only check a PACT's `TxnContext` when the `GetState` API is called. A set of keys that the PACT tends to access must be given (Fig.5.5a). It is because the key-level schedule only specifies when it is safe for a PACT to access a specific key; therefore, the time when the `GetState` API returns the result varies depending on which key or a set of keys the PACT is acquiring. Besides, the PACT should only get access to the keys that it has acquired; otherwise, the transaction execution schedule may be violated. For example, suppose two PACTs T_1 and T_2 are scheduled on one actor. T_1 will only access the key k_1 and T_2 will access both k_1 and k_2. If T_2 acquires k_2 first, it can get the `DictionaryState` immediately. However, T_2 is not yet allowed to access k_1 because the status of T_1 is unknown. To safely access k_1, T_2 must call the `GetState` API with k_1 included in the key

set. In SnapperX, to accurately restrict the keys that a PACT can access, each DictionaryState object will only include the information of keys that the PACT has acquired. As illustrated in Fig.5.5b, in step 8, SnapperX makes a copy of the key's information, including the key-value pair itself and its related dependencies. It also implies that to access the dependency information of a key, the transaction also needs to get access to the key first.

In addition, SnapperX allows a transaction to call GetState API multiple times, each time may acquire the access right to different sets of keys. SnapperX does not restrict the order to acquire different keys and the number of times each key is acquired. SnapperX only maintains one DictionaryState instance for each transaction. When the GetState API is called for the first time, a new DictionaryState object is created for the transaction. Then, every time a new key is acquired via the GetState API, this key is added to the existing DictionaryState object. If the transaction asks for access to a key that already exists in the DictionaryState, no extra waiting is needed (e.g., steps 6, 7, 8 in Fig.5.5b). Note that the GetState API always returns the reference to the DictionaryState object that belongs to the current transaction, and developers can get access to this object or use this reference via any of the GetState calls. By doing so, a transaction can always access the latest actor state via the same reference, and the GetState API can be used for gradually extending the access right to different keys.

An example is shown in Fig.5.3 (lines 19-41). In this example, a DeleteItemsInCart transaction first gets access to its DictionaryState object by calling the GetState API without specifying any keys (line 21). Within this call, SnapperX will create a new DictionaryState object for this transaction, which contains no keys. Afterward, the transaction can acquire access rights to a set of items (or keys) in two ways (lines 25-33, options 1 and 2). In both ways, the state object eventually gains the RW access to all acquired keys. In other words, keys are gradually added into the DictionaryState object and, therefore, can be accessed by the transaction. At last, the transaction deletes these items from the cart (lines 35-40).

As prescribed by Snapper, SnapperX also requires for PACT the declaration of actors and keys accessed as part of a transaction, including the ones accessed during the dependency management process. As the example in Fig.5.3, each item in the cart actor is a replica of the corresponding product on the product actor (lines 3-17), which contains the latest product price. The replication here is expressed as an update dependency in SnapperX. When items are deleted on the cart actor, it indicates the deletion of their dependencies stored on both the cart actor and the product actor (lines 35-40). Therefore, the transaction DeleteItemsInCart actually involves two actors and will access the same set of keys on each actor. The correct and complete key access information must be given to allow SnapperX to schedule PACTs at the key level correctly.

5.3.3.4 Key-level Concurrency Control for ACT

`Snapper` also supports ACT, the type of transaction that does not declare the actor access information and applies non-deterministic execution. More specifically, each ACT gets access to an actor state by acquiring the RW lock maintained on the actor via the S2PL+wait-die protocol. To extend the actor-level concurrency control to the key-level concurrency control, `SnapperX` can simply maintain a lock for every single key. Given that keys may be dynamically added and deleted on an actor, locks need to be added and removed accordingly. In `SnapperX`, an ACT can add or delete a key k on the condition that it gets the write lock of k and a new lock for k is created if it does not exist yet. Note that a key can only be deleted when there are no ACTs holding or waiting for the lock to avoid anomalies.

`SnapperX` provides interfaces for transactions to access keys individually while still preserving the ability to support actor-level concurrency control. The actor-level concurrency control is useful for cases where a transaction needs to scan the whole key-value collection on an actor or to query keys with certain predicates. `SnapperX` allows developers to configure an actor to apply either actor-level or key-level concurrency control when it is created.

5.4 Evaluation

In this section, we conduct an extensive range of experiments to investigate the features of `SnapperX` under various workloads. The first part of the experiments (§ 5.4.3, 5.4.4, 5.4.5) explores the characteristics of the basic building blocks of the data model. More specifically, we focus on the trade-offs of maintaining fine-grained key-value actor states and transaction processing performance. In the second part (§ 5.4.6), we turn our attention to popular cross-microservice correctness criteria sought by developers in practice [89]. In particular, we adopt the `Online Marketplace` benchmark and target exploiting the overhead of enforcing constraints cutting actors across, including foreign keys, data replication, and functional dependencies.

5.4.1 Implementation Variants

Five competing systems (Fig.5.7) are compared to show the implications of our proposed advancements to performance. `NonTxn` applies a non-transactional execution on Orleans. Actors execute operations in arbitrary order; thus, accesses to the state are performed without isolation guarantees. It gives an upper bound of the system's performance, indicating the best throughput the actor system can achieve under a certain workload. `Snapper` [99] is adopted in our experiments as a baseline solution for enforcing application correctness with strong consistency guarantees. To allow further insights, we integrate different data model functionalities in `Snapper` and devise two other variants.

`Snapper+` combines the concept of keys and dependencies with `Snapper`, which facilitates transaction processing with incremental logging. `SnapperX` further adds key-level concurrency control on top of `Snapper+`. Therefore, `SnapperX` is the version that applies all the optimizations. The purpose of having `Snapper+` is to benchmark the effects of the different optimizations. In addition, we also include Orleans Transactions [55](`OrleansTxn`) in our experiments. It is the default transaction management API in Orleans and applies 2PL and 2PC to fulfill ACID multi-actor transactions. That enriches the experiment by confronting competing concurrency control scheduling techniques, namely, locking-based in Orleans and deterministic in Snapper variants.

	Data Model	Logging		Concurrency Control		Access To Actor State
		snapshot	incremental	actor-level	key-level	
NonTxn						Get direct access to actor state.
Snapper		✓		✓		**PACT**: Get direct access to actor state. **ACT**: Get a cloned version of actor state.
Snapper+	✓		✓	✓		**PACT & ACT**: Get cloned version of the acquired keys
SnapperX	✓		✓		✓	
OrleansTxn		✓		✓		Get a cloned version of the actor state.

Figure 5.7: Implementation variants

5.4.2 Experimental Setting

5.4.2.1 Deployment

All experiments are run on Orleans 8.1.0 and .NET SDK 8.0.300. Each experiment is conducted on a Orleans cluster consisting of a master node (`MN`) and several worker nodes (`WN`), each hosting a Orleans server and located in the same region. The `MN` is responsible for coordinating PACT execution on different `WN`s for `Snapper`, `Snapper+` and `SnapperX`. In addition, a group of experiment nodes (`EN`) are spawned in the same region to generate workloads, host Orleans clients, and submit transaction requests to Orleans servers. The same number of `EN`s and `WN`s are deployed to ensure a sufficient amount of requests are generated and dispatched to `WN`s. Each node, regardless of the type, is an AWS EC2 instance (c5n) with a 2-core processor (4 vCPUs). In the scalability experiment (§ 5.4.5 and 5.4.6), we proportionally increase the number of `WN`s and `EN`s, as well as the number of vCPUs of the `MN`.

On the client side, each `EN` initiates one Orleans client thread for PACT and ACT executions, respectively. The client thread submits a pipeline of transaction requests to `WN`s. The pipeline size determines the concurrency level of the workload. More specifically, it limits the maximum number of concurrent requests in the system. Whenever the result of one request is returned, a new request is replenished. In our experiments, the pipeline sizes are tuned for different implementation variants so that they all achieve a

good performance while the system's computing resources are near saturation. Fig.5.8 shows how the pipeline size is configured.

pipeline size							
NonTxn			128				
PACT	Snapper		128				
	Snapper+		128				
ACT	Snapper		2	4	16		
	Snapper+		2	4	16		
OrleansTxn			2	4	16		
# actor per WN			10	100	1000		
PACT	SnapperX		128				
ACT	SnapperX		4	16	128	512	512
# keys per WN			100	1K	10K	100K	1000K

Figure 5.8: Pipeline size

5.4.2.2 `SmallBank` Benchmark

We adopt the `SmallBank` ben- chmark [11] in the first part of the experiments to gain insights about the features of `SnapperX`. In this benchmark, users' bank accounts are partitioned across many account actors, and operations such as `Deposit` and `Transfer` are applied to one or multiple account actors. `SmallBank` approximates an OLTP actor-oriented workload. Note that this benchmark shows no cross-actor dependencies, and we use it for investigating the performance of fundamental building blocks of `SnapperX`, including incremental logging and the key-level concurrency control. In our experiments, we only employ the `MultiTransfer` transaction [155], which withdraws money from accounts on one actor and deposits money to several accounts on other actors. This transaction gives us the flexibility to control the transaction size and the access pattern to different actors and accounts. More specifically, we adopt five parameters to configure a `SmallBank` workload. $numActor$ is the total number of account actors located in each `WN`. $actorSize$ determines the number of bank accounts, i.e., the number of keys, stored in each account actor. $txnSize$ specifies the number of keys a transaction will access on each selected actor. In the experiments, we fix the number of accessed actors for each transaction as 4. Thus, $txnSize = 4$ means the transaction will access a total of 16 keys across four actors. $actorSkew$ determines the number of hot actors on each `WN`. Each transaction selects a set of actors to access based on the following rule: there is a 75% chance that an actor is selected from the set of hot actors. For example, when $numActor = 1000$ and $actorSkew = 1\%$, the hot set contains ten actors. And when $actorSkew = 100\%$, every actor has an equal chance to be chosen. The smaller $actorSkew$, the higher contention at the actor level. Similarly, $keySkew$ decides the number of hot keys on each actor and controls the contention at the key level.

5.4.2.3 `Online Marketplace` Benchmark

We also run experiments with the `Online Marketplace` [88] benchmark, which mainly focuses on data management challenges in an event-driven and micro-service-like system architecture. It simulates a multi-tenant web-scale application where sellers manage their products and associated inventory, and customers interact with the system by managing their carts (i.e., adding products) and submitting them for checkout. The reason for adopting this benchmark in our experiments is threefold. First, it covers several scenarios of cross-component data integrity constraints, which provides a perfect use case for our data model where the dependency between keys across actors is automatically handled. Existing commonly adopted benchmarks, such as YCSB [42] and TPCC [174], do not model these types of correctness criteria, thus unfitting the goals of our experiments. Second, it can be easily mapped to the actor model by partitioning each component into multiple actors. Meanwhile, the application state can be easily modeled as key-value collections across different types of actors. Third, it forms a realistic and complex workload where some transactions can be implemented as PACT and some as ACT. Its business logic is carried out among eight different components as listed in Fig.5.14a. § 5.4.6 discussed more details about how the experiment is set up with this benchmark.

5.4.3 Characteristics of `SnapperX`

In this section, we investigate the impact of `SnapperX` with the `SmallBank` benchmark. We measure the overhead of concurrency control and logging by presenting the relative throughput of `Snapper`, `Snapper+`, and `SnapperX` with regards to the throughput of `NonTxn`. The differences between these three `Snapper` variants are supposed to reflect the trade-offs of a state management layer, namely, the overhead to maintain the key-value collection on each actor and the benefits it brings to the system performance. In this group of experiments, workloads are generated with a uniform distribution ($actorSkew = 100\%$ and $keySkew = 100\%$), and we vary the other three parameters, $numActor$, $actorSize$, and $txnSize$.

5.4.3.1 Vary $actorSize$

In this experiment, we fix $numActor = 10$, $txnSize = 1$, and vary $actorSize$, Fig.5.9a shows the results. For PACT, when logging is disabled, the throughput of `Snapper` and `Snapper+` is not affected by $actorSize$, because they both apply actor-level concurrency control, in which the number of keys in an actor does not affect the contention at the actor level. Besides, the throughput of `Snapper+` is slightly lower than `Snapper`. It is because `Snapper` allows each PACT to access the actor state directly, while `Snapper+` needs to copy the accessed keys and operate on the cloned version. This is to fulfill the func-

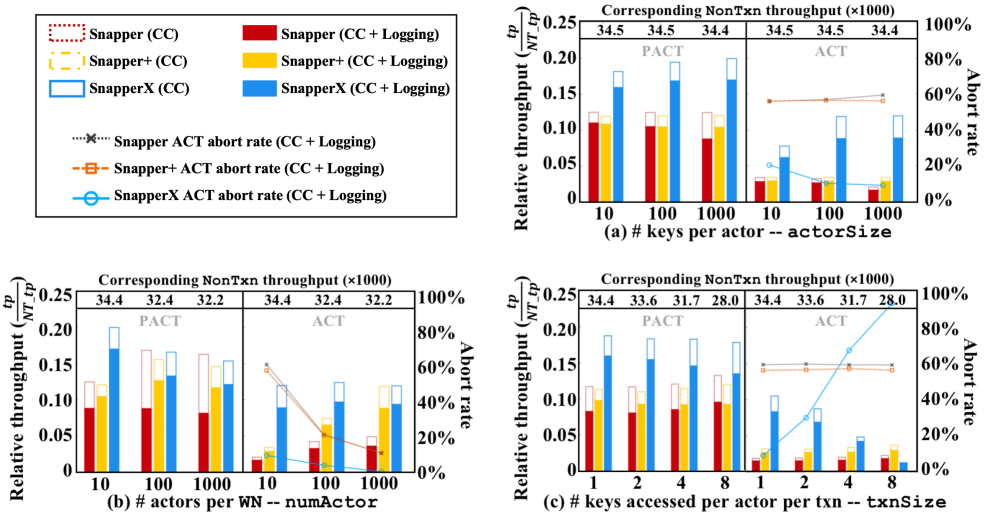


Figure 5.9: Characteristics of SnapperX

tionality of the data model that the before- and after-image of the modified keys are both captured. In contrast, when adding more keys to each actor, the throughput of SnapperX increases because the key-level concurrency control benefits from the reduced contention on keys. And the gain from the finer-grained scheduling offsets the overhead of key cloning. When logging is enabled in PACT, Snapper throughput decreases largely, especially from 100 to 1000 keys per actor, while Snapper+ and SnapperX rarely change. This shows the advantage of incremental logging that Snapper+ and SnapperX only need to persist the changes on specific keys, but Snapper has to persist the whole actor state.

For ACT, Snapper and Snapper+ show a similar trend as PACT – their throughput and abort rate remains the same regardless of *actorSize*. Differently, SnapperX throughput increases and the abort rate decreases. when $actorSize = 1000$, there is a significant gap between Snapper and Snapper+. This is because, in Snapper, every ACT has to make a copy of the whole actor state and apply read or write operations on the cloned state. Given that Snapper guarantees PACTs do not abort due to transaction conflicts, it is safe to have PACT modifying the actor state in place. However, every single ACT can be aborted; thus Snapper applies the updates of an ACT to the actor state only when the ACT commits.

5.4.3.2 Vary *numActor*

In this section, we vary *numActor* and fix $txnSize = 1$, $actorSize = 1000$. Fig.5.9b shows the results. For PACT, Snapper and Snapper+'s throughput significantly increases from 10 to 100, then slightly decrease from 100 to 1000. When there are only 10 actors in each WN, the contention on each actor is

extremely high. In `Snapper` and `Snapper+`, each actor executes PACTs one by one in the ascending order of transaction ID (*tid*) and batch ID (*bid*). In addition, each actor is also responsible for tasks, including receiving batch messages and committing batches, which further extends the critical path of transaction processing. When more actors are added to the system, more transactions can be processed in parallel on different actors, therefore leading to a higher throughput. The decreased contention on actors also benefits their ACT execution – the ACT throughout increases, and the abort rate decreases.

However, when we keep adding more actors, e.g. to 1000 actors, the parallelism is not improved further due to the limited number of vCPUs. Meanwhile, the throughput drops when there are too many actors because the batching of PACTs becomes less efficient when the set of actors accessed by each PACT is less likely to overlap. We define the overlap rate r as the number of PACTs included in a batch divided by the number of actors accessed by the batch. According to the collected experimental data, we get $r = 1.0, 0.4, 0.2$ for $numActor = 10, 100, 1000$ respectively. This overlap rate has a more obvious impact on `SnapperX` that its throughput continuously decreases. In this experiment, even if more keys are added while adding more actors, the contention on keys remains low, so the performance of `SnapperX` is not improved so much. The ACT abort rate of `SnapperX` also keeps at a low value.

In terms of the difference between with and without logging, similar to the results observed in § 5.4.3.1, the throughput of `Snapper` drops significantly when logging is enabled, such that its PACT throughput goes below `Snapper+` and `SnapperX`.

5.4.3.3 Effect of *txnSize*

In this part, we fix $actorSize = 1000$, $numActor = 10$ and vary $txnSize$. Fig.5.9c shows the results. For `NonTxn`, its throughput obviously decreases with larger $txnSize$ because $txnSize$ determines the complexity of the transaction logic and the transaction execution latency. For both PACT and ACT, the absolute throughput of `Snapper` and `Snapper+` decreases, but the relative throughput slightly increases when $txnSize$ grows. It indicates they are not as sensitive to the change of $txnSize$ as `NonTxn`. PACT is mainly affected by actor-level transaction scheduling. Each PACT spends $\sim 80\%$ time waiting for the turn to start execution on the first actor, and only 3% time executing the transaction logic. For ACT, the dominant factors are the concurrency control (2PL) and the commit protocol (2PC). In addition, since both `Snapper` and `Snapper+` apply actor-level concurrency control, their abort rate remains the same even if $txnSize$ grows.

In contrast, `SnapperX` is a lot more sensitive to the change of $txnSize$ compared to `Snapper` and `SnapperX`. Its PACT throughput decreases due to the increased overhead of generating and maintaining the key-level transaction execution schedules, as well as persisting key modifications. The ACT through-

put drops significantly, and the abort rate increases largely. In `SnapperX`, an actor maintains a lock instance for every key stored on the actor, and each ACT needs to acquire the corresponding lock for every accessed key. Compared to PACT, ACT suffers more from contention. When $txnSize = 8$, the ACT throughput of `SnapperX` drops below `Snapper` and `Snapper+`. For one thing, the number of concurrent transactions submitted to `SnapperX` is much higher than `Snapper` and `Snapper+` (128 vs 2). As is explained in § 5.4.2.1, for `SnapperX`, this number is tuned based on the total number of keys in each `WN`, while for `Snapper` and `Snapper+`, it is based on the total number of actors. In this experiment, when $txnSize$ grows from 1 to 8, the contention at the key level largely increases; thus, the ACT throughput of `SnapperX` experienced a significant drop.

5.4.3.4 Conclusion

In conclusion, the data model is most effective when there are a small number of large actors, and the data model benefits more for workloads that access fewer keys. First, the data model helps the system capture changes performed on specific keys, thus largely reducing the logging overhead for `Snapper+` and `SnapperX`. Second, the data model can further exploit concurrency in every single actor; therefore, `SnapperX` performs significantly better than the other two when the contention is high at the actor level and low at the key level. Third, the overhead brought by the data model, such as the key cloning and the key-level transaction schedule maintenance, can be completely offset by the benefits brought by the finer-grained logging and higher concurrency level.

5.4.4 Skewed workload

This section presents how performance is affected by skewed workloads. We fix $numActor = 1000$, $actorSize = 1000$, $txnSize = 1$ and vary $actorSkew$, $keySkew$. Logging is enabled for all implementation variants from now onwards. In this section, we present not only the throughput of PACT and ACT (Fig.5.10 a-c) but also the breakdown latency of PACT (Fig.5.10 d-f). We divide the latency of each PACT into 7 time intervals. The breakpoints are set according to the progress of a PACT on the first accessed actor. The time intervals I_1, I_2, I_3, I_4, I_6 and I_7 represent the time spent on steps 2, 6, 7, 8, 21 and 23 in Fig.5.5b, respectively. Note that I_5 begins when A starts to execute the transaction logic and ends when the whole transaction completes. I_5 includes the time to forward calls to other actors and execute these transactional invocations on other actors.

5.4.4.1 Skew on actors

Here, we investigate the impact of $actorSkew$, while fixing $keySkew = 100\%$. As explained in § 5.4.2.2, a smaller $actorSkew$ indicates higher contention at

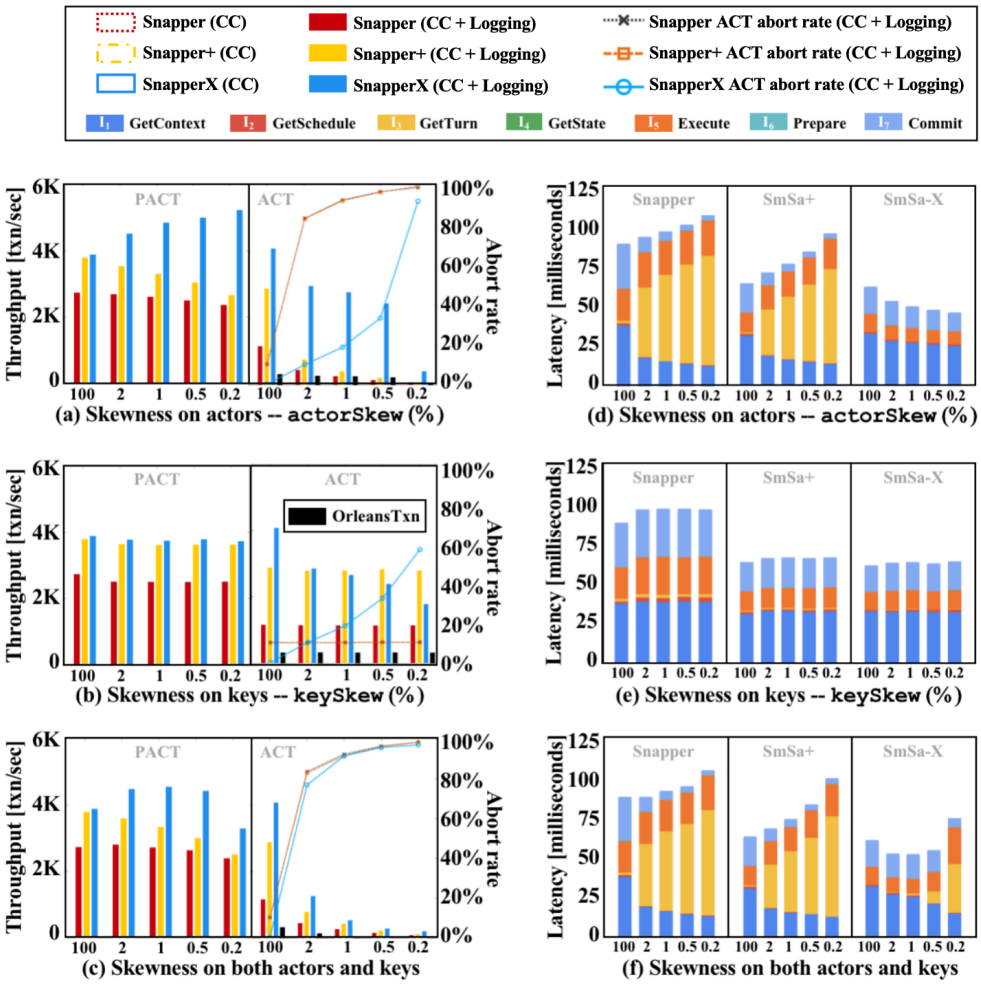


Figure 5.10: Effect of skewed workload

the actor level. Fig.5.10a and 5.10d show the throughput and breakdown latency, respectively. For PACT, when *actorSkew* decreases, `SnapperX` benefits greatly from the growing contention on actors. As is discussed in § 5.4.3.2, when the workload is concentrated on a smaller set of actors, the batching of PACTs becomes more efficient due to a higher overlap rate. On the contrary, the contention on actors has a negative impact on `Snapper` and `Snapper+`. According to their breakdown latency, I_3 of `Snapper` and `Snapper+` grows obviously with decreasing *actorSkew*, indicating that each PACT is blocked for a longer time while waiting for previous PACTs to complete. Even if `Snapper` and `Snapper+` also benefit from the higher batching efficiency (I_1, I_2, I_6 and I_7 decrease), the increased I_3 dominates the transaction latency. Therefore, their throughput shows a decreasing trend. I_3 of `SnapperX` remains very low because the contention on keys is low, and a PACT only needs to wait for the

completion of a previous PACT that accesses the same key.

For ACT, the throughput of all three `Snapper` variants decreases. `Snapper` and `Snapper+` are affected by the contention at the actor level. However, `SnapperX` is also affected by *actorSkew* because it indirectly influences the contention at the key level. In addition, we measure the throughput of `OrleansTxn`. Given that `OrleansTxn` is reported to be extremely vulnerable to contention [99, 34, 36, 98], here we use a workload with no deadlocks but under the same actor and key distribution (*actorSkew* and *keySkew*). Deadlocks are removed by letting each transaction access the selected actors always in the ascending order of actor IDs. In our result, `OrleansTxn` remains a very low throughput under different *actorSkew* values.

5.4.4.2 Skew on keys

In this part, we present the impact of *keySkew*, which determines the contention at the key level. We fix *actorSkew* as 100%. Fig.5.10b and 5.10e show the throughput and breakdown latency, respectively. The PACT and ACT throughput of `Snapper` and `Snapper+` rarely change while varying *keySkew*, because the contention on keys does not affect how transactions are scheduled at the actor level. For `SnapperX`, its PACT throughput remains unchanged. The growing contention on keys does not cause a growing blocking time (I_3) for `SnapperX` because the skewness is not high enough with $actorSkew = 100\%$. In contrast, its ACT throughput decreases largely. This validates that the ACT execution of `SnapperX` is more sensitive to contention on keys than PACT. Again, `OrleansTxn` has a much lower throughput than the ACT execution of all three variants.

5.4.4.3 Skew on both actors and keys

Here, we present the combined effects of *actorSkew* and *keySkew* by changing their values from 100% to 0.2% simultaneously. Fig.5.10c and 5.10f show the result. For `Snapper` and `Snapper+`, their PACT and ACT throughput show the same pattern as in § 5.4.4.1 that they are not affected by contention on keys. For `SnapperX`, its PACT throughput increases, then decreases. Compared to Fig.5.10b, the contention on keys grows much faster in Fig.5.10c. I_3 of `SnapperX` also increases a lot from 1% to 0.2%. For its ACT execution, the throughput has dropped significantly already from 100% to 2% because ACT is more sensitive to contention.

5.4.4.4 Conclusion

In conclusion, the PACT execution of `SnapperX` greatly benefits from skewness on actors because of the improved batching efficiency. When there are a large number of actors in the system and the workload follows a uniform distribution, `SnapperX` does not have an obvious advantage over `Snapper` and

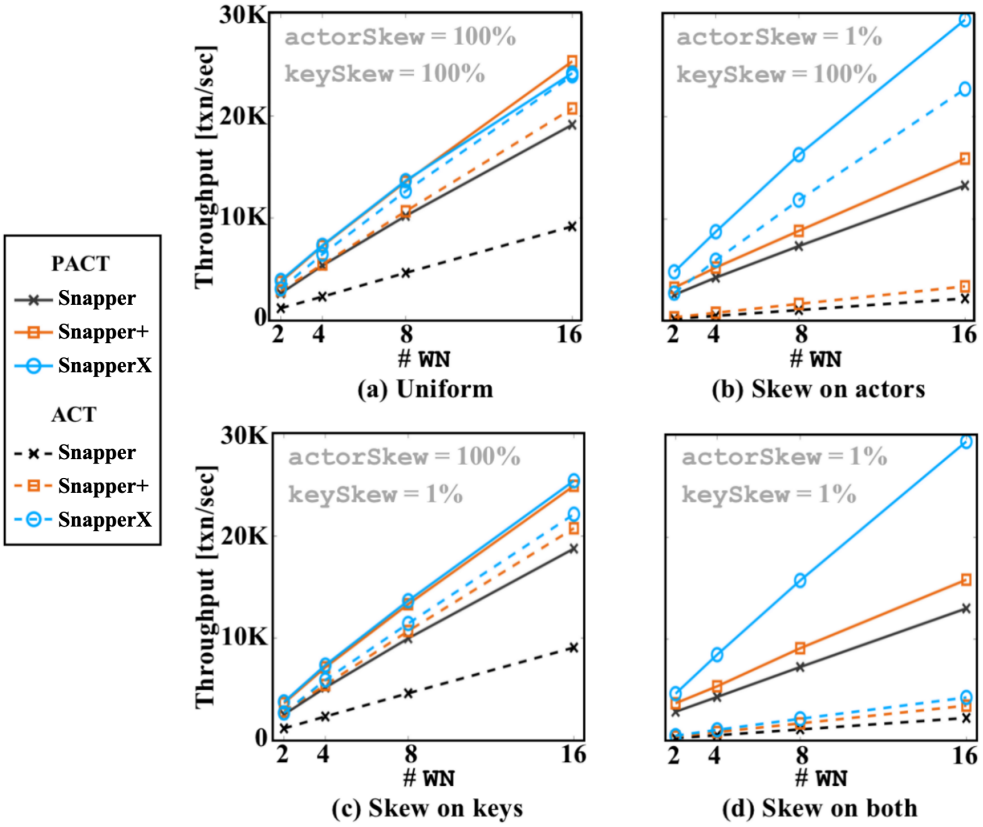


Figure 5.11: Scalability of `SmallBank`

`Snapper+`. When the workload is skewed on actors, `SnapperX` outperforms `Snapper+` and `Snapper`. In addition, the PACT execution of `SnapperX` is less sensitive to contention on keys than its ACT execution. Its PACT throughput is affected only when the skewness on keys is at a very high level. the ACT execution of `SnapperX` performs better than `Snapper` and `Snapper+` in most cases, except when the contention on keys is extremely high.

5.4.5 Scalability

In this section, we validate the scalability of `SnapperX`. We measure the throughput of PACT and ACT of three variants (`Snapper`, `Snapper+`, and `SnapperX`) under different $actorSkew$ and $keySkew$, with $numActor = 1000$, $actorSize = 1000$, $txnSize = 1$. Fig.5.11 shows that all cases scale linearly. The same as in § 5.4.4, `Snapper` and `Snapper+` are only affected by $actorSkew$, not $keySkew$. When $actorSkew$ changes from 100% to 1%, the contention at the actor level increases. The PACT throughput of `Snapper+` decreases from $25K$ to $16K$, and `Snapper` from $19K$ to $13K$. The ACT throughput of `Snapper+` decreases from $20K$ to $3K$, and `Snapper` from $9K$ to $2K$. Their

ACT throughput decreases more significantly than the PACT because ACT execution is more sensitive to contention at the actor level. Differently, for `SnapperX`, its PACT throughput even increases from $25K$ to $30K$ when the contention on actors grows. And its ACT throughput only drops when the contention on keys is extremely high (Fig.5.11d).

5.4.6 Online Marketplace

In this section, we measure the performance of different implementation variants under `Online Marketplace` benchmark [88]. For `Snapper`, we adopt conventional actor state manipulation and encode all dependencies in the application logic. `Snapper+` benefits from the fine-grained state manipulation, which ought to decrease logging overhead. For `SnapperX`, we exploit the full contributions of this work, providing a sophisticated actor state manipulation, transparent constraint enforcement, and novel logging and concurrency control techniques.

5.4.6.1 Modeling

The `Online Marketplace` contains eight different components, each encapsulating corresponding application logic and maintaining a set of relations. In our implementation, as is shown in Fig.5.13a, each component is mapped to a group of actors. Each seller maintains 1000 products, and 100 sellers make a total of $100K$ products in the system. Besides, three dependency constraints are modeled. First, each item in a cart actor is essentially a replica of the product in the corresponding product actor; thus, an update dependency is built between the original and the copied keys. Second, the stock of a product in the stock actor is a foreign key of the product in the product actor, which can be interpreted as a delete dependency. Third, each seller actor maintains an up-to-date materialized view of the total number of orders created. We implement it by using a functional update dependency.

actor type	# actor per WN	key-value pairs per actor	# entries
Customer	100	CustomerID => CustomerInfo	1
Cart	100	**ProductID** => ProductInfo	varies
Payment	100	CustomerID => PaymentMethod	1
Order	100	NextOrderID => long	1
Seller	100	**SellerID** => AggregatedOrderInfo	1
Product	100	**ProductID** => ProductInfo	1000
Stock	100	**ProductID** => StockInfo	1000
Shipment	100	NextPackageID => long	1
		SellerID => LastCreatedPackage	1

Figure 5.12: `Online Marketplace` modeling

5.4.6.2 Transactions

In our experiments, four types of transactions are adopted (Fig.5.13b). The `Checkout` transaction crosses seven types of actors, and the total number of actors involved depends on the number of items that are bought. Steps 6 and 7 represent tasks that happen while resolving the dependency constraints. In our experiment, a workload consists of 30% `AddItemToCart`, 20% `DeleteItemInCart`, 10% `Update- Price` and 40% `Checkout` transactions. Each transaction is generated by selecting a customer or a seller under the *actorSkew* and selecting a product under the *keySkew*. For a `Checkout` transaction, the number of items to checkout varies from 1 to 5. In addition, based on the observation from previous experiments that ACT is vulnerable to contention, we implement transactions as PACT whenever we can in this experiment. Therefore, only `UpdatePrice` transaction is executed as ACT because it is unknown which cart actors have dependencies on the product when a `UpdatePrice` transaction is submitted.

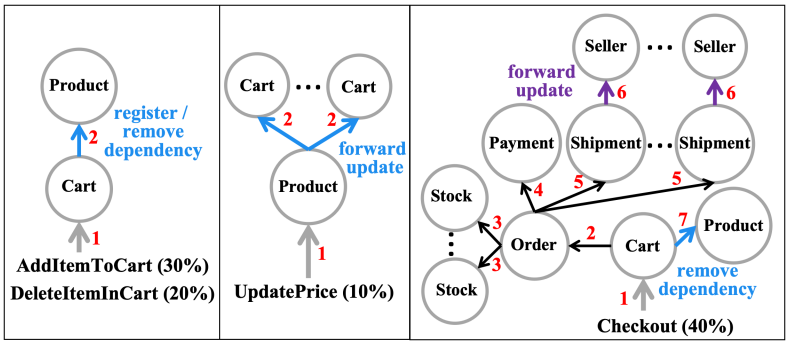


Figure 5.13: `Online Marketplace` transactions

5.4.6.3 Scalability under Skewed Workloads

Fig.5.14a shows the result of a scalability experiment on `Online Marketplace` workload. Fig.5.14b shows the throughput of each type of transaction when $\#WN = 16$. Here, the PACT throughput represents the deterministic execution of all transactions, except for `UpdatePrice`.

For PACT, under the uniform distribution, `Snapper+` throughput is slightly higher than `SnapperX`. When *actorSkew* decreases to 2%, their PACT throughput both increases; however, `SnapperX` surpasses `Snapper+`. It is because they both benefit from higher batching efficiency, but `Snapper+` is further affected by the increased blocking on each actor, while `SnapperX` remains at a higher concurrency level, benefiting from key-level concurrency control. When *keySkew* is decreased to 2%, the throughput of `Snapepr+` and `SnapperX` both decrease moderately, and the impact on `UpdatePrice` transaction is more severe than on other transactions.

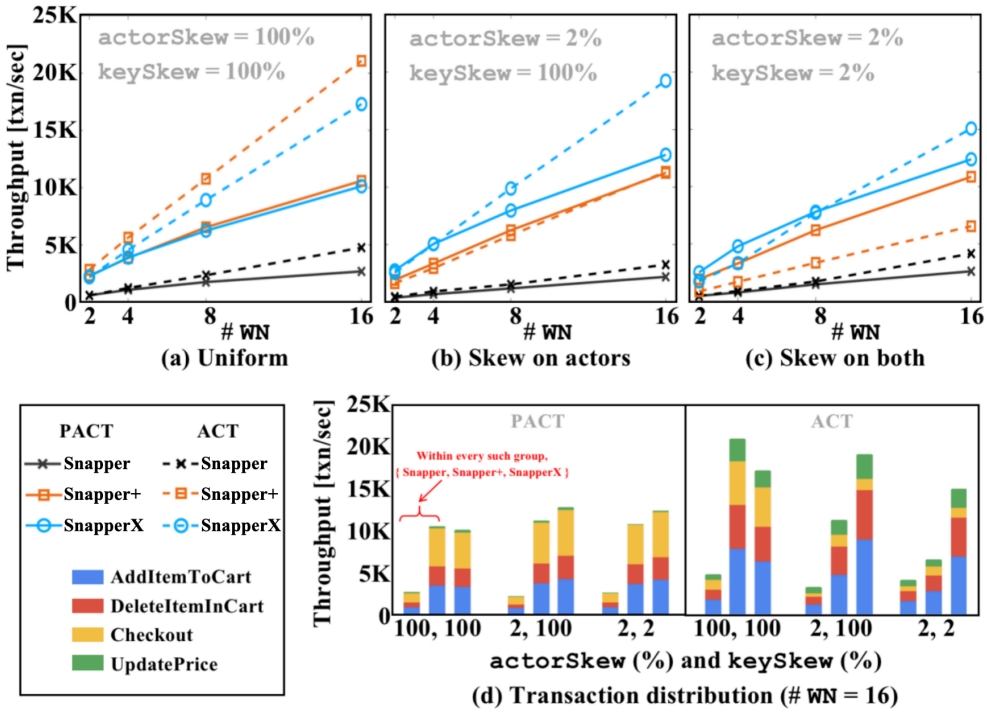


Figure 5.14: Performance of `Oneline MarketPlace`

The ACT throughput of all three variants is higher than the corresponding PACT throughput. As is shown in Fig.5.14b, the transaction distribution of ACT differs from PACT – the throughput of smaller transactions such as `AddItemToCart` and `DeleteItemInCart` accounts for a much higher proportion in ACT execution. This is because ACT has the flexibility to quickly abort large transactions, in exchange for the commit of smaller-sized transactions. For `SnapperX`, its ACT throughput increases when *actorSkew* changes to 2%, because it gains throughput from smaller transactions by aborting larger transactions. When further changing *keySkew* to 2%, its ACT throughput decreases. It is because the contention on keys becomes very high, such that smaller transactions also suffer from the high contention. Thus, all types of transactions experience a higher abort rate and result in a lower throughput.

For `Snapper+`, its ACT throughput decreases from $21K$ to $11K$ when *actorSkew* changes from 100% to 2%. It is because `Snapper+` suffers from contention on actors. However, when further varying *keySkew* from 100% to 2%, `Snapper+` throughput drops from $11K$ to $6K$. Note that when *keySkew* decreases, it raises the possibility that the same product is added by many carts and further causes a larger `UpdatePrice` transaction (Fig.5.15), therefore bringing higher contention at the actor level. As for `Snapper`, its PACT and ACT throughput remain at a low level in all three groups due to the high logging overhead.

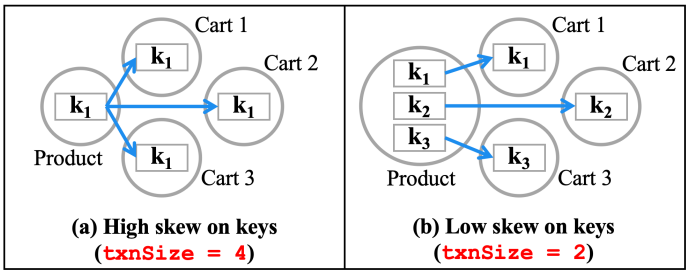


Figure 5.15: Different skewness on keys

In conclusion, under the `Online Marketplace` benchmark, where the dependency constraints are broadly applied, `SnapperX` shows its advantage over the other two variants in both PACT and ACT execution, and it reacts to different skewed workloads in a similar way as in § 5.4.4.

5.5 Related Work

Data replication is a mechanism widely adopted in distributed systems to leverage data locality, decrease data access latency, enhance fault tolerance, and achieve higher system availability. Existing actor systems adopt methods like event sourcing [5, 122] and geo-distributed caching [27] to replicate the actor state, achieving eventual consistency and linearizability, respectively. However, actor replication differs from replicating data items, since the technique involves actor metadata (e.g., type), a possibly large actor state and actor functionalities, thus introducing a higher overhead. In this work, an actor state abstraction is designed to facilitate the identification and replication of data items across actors while capturing and maintaining the primary-replica relations.

The actor system, Akka [5], relies on the conflict-free replicated data types (CRDTs) [156, 92] to replicate data across nodes with eventual consistency guarantee. Although not incurring the high overhead of actor replication, having data types as the abstraction level inherits the same impedance found in Orleans, updates to single entities are treated as an update to the whole actor state, preventing optimizations. As a result, such method is oblivious to the possible data dependencies across actors.

Anna [180] is a distributed key-value store that employs the actor model transparently to developers in order to process and possibly merge concurrent updates. Different from our work, Anna does not expose an actor programming model nor offer data dependency management and constraint enforcement. Besides, Anna focus on eventual consistency scenarios, contrasting with `SnapperX` support for ACID guarantees. DPA [84] is a data platform for OLAP workloads where developers utilize an actor-based programming model to abstract and advance an underlying data analytics system. DPA is designed for

bulk data updates, which inhibits its use in event-driven, highly-transactional scenarios as found in microservices.

Data dependency is another crucial aspect of data management, which specifies how data in one object depends on data in others. To enforce data dependency constraints in the context of actor systems, recent research works [3, 155, 178] have focused on connecting actors to relational database systems so as to regain their support for declarative querying and extensive data management functionalities. In this approach, operations on data are forwarded through actors to the backend storage and handled there. However, this model conflicts with the function shipping paradigm, which decouples the fast computation from the slow storage [30]. In contrast, dependencies between different data items across actors are captured by `SnapperX` in the application layer. Meanwhile, dependency constraints are enforced with transactional guarantees.

5.6 Conclusion

The actor model emerged as a promising concurrency model for facilitating distributed application design. However, exposing an opaque state management abstraction limits developers' ability to express complex relationships that cut across actors, inhibiting further adoption of the actor model.

To fill this gap, we propose `SnapperX`, a state management layer that advances the actor model with rich state management abstractions and novel logging and concurrency control algorithms. Our experimental study shows `SnapperX` enhances the design of complex relationships in actor systems and improves by 2X the performance of state-of-the-art deterministic concurrent control methods. As a result, `SnapperX` will facilitate designing highly partitioned and distributed data-intensive applications based on the actor model. As for future work, we identify the potential to further increase `SnapperX`'s performance when there is high contention on keys. Though there is little space to further exploit concurrency on an individual actor, techniques like actor re-balancing or key re-partitioning can be adopted.

Chapter 6

Conclusion

To bridge the gap between the actor model and the unique requirements of modern applications, this dissertation builds a scalable and transactional AODB, which provides a state management solution under a strong consistency guarantee for applications with actor-based middle tiers. Our solution advances the state-of-the-art implementation of AODB by introducing novel techniques. This dissertation achieves the goal in three steps. `Snapper` proposes and validates the effectiveness of an efficient transaction processing technique, which serves as a fundamental building block for the AODB. `SnapperD` extends the architecture of `Snapper` to a distributed environment while preserving its good performance. `SnapperX` advances the actor model with a dedicated data model, which exploits the potential of individual actors and further improves system performance.

6.1 Ongoing and Future Work

Here, we outline several directions for future research that build upon the findings of this dissertation.

- **Vulnerability to incorrect transaction inputs**. One critical limitation of deterministic transaction execution is its correctness heavily relies on the user input. The current scheduling mechanism is vulnerable to inaccurate actor access information. If a transaction accesses an actor that is not declared when the transaction is submitted, or if a transaction does not access all the actors that have been declared, the system will fail to fulfill the deterministic schedule and be blocked forever. In addition, when plugging in the dedicated data model, the system requires each transaction to specify the set of keys to access, which is more detailed information and more likely to be error-prone. Therefore, we think two improvements should be explored for our system: (1) the support for developers to run their transactions in a debug

mode and check the correctness of the input data, which can help to correctly develop applications on our system with higher productivity, and (2) the ability to detect such blocking situations and proactively abort suspicious transactions to allow the system to continue.

- **Limited ability to abort deterministically scheduled transactions**. Enabling the ability to abort deterministically scheduled transactions not only helps to deal with incorrect transaction inputs but also loosens the strictness for developers to program transaction logic. To abort these transactions in our system, the current solution relies on the logged information and rolls back the system to the latest committed state. This process can be time-consuming and requires extra design for the log file to support efficient log scanning. The primary challenge to aborting such transactions is handling cascading abort. Making extra assumptions about transactions and performing more controlled speculative execution can help to solve this challenge. Similar mechanisms can be found in systems that adopt deterministic scheduling. For example, allowing the definition of partially abortable transactions can avoid cascading abort or restrict its impact within a smaller scope. These mechanisms can be explored in the future to improve our system.

- **High abort rate under hybrid scheduling**. A drawback of the novel hybrid concurrency control proposed in this dissertation is that it leads to a very high abort rate for non-deterministically executed transactions, especially under high contention. In this method, we always prioritize the progress of PACTs, optimistically schedule ACTs, and abort ACTs for any found or potential serializability issues. This may lead to false abort of ACTs. We conjecture there are opportunities to reduce the abort rate by giving more control to the hybrid scheduling or retrieving more information about the generated deterministic schedules. This is a non-trivial goal due to the novel concept of this hybrid approach. Extensive and in-depth research works are needed to find a hybrid protocol that guarantees serializability or weaker isolation levels while achieving a lower abort rate.

- **A proposal for TAODB (Transactional AODB)**. Throughout this dissertation, the support for multi-actor transactions serves as a fundamental component of our system. In many existing database systems, transactional properties are required not only for transactions but also for features like indexing management, data replication, and data migration. These features gain transactional guarantees by incorporating their operations within the context of a transaction. Therefore, we propose a novel concept, TAODB (Transactional Actor-Oriented Database), which prioritizes comprehensive transactional support across the entire system. Unlike AODB, which primarily focuses on dynamically inte-

grating different database features with actor systems, TAODB extends this approach by ensuring that all system features are built directly on top of transactional guarantees. This approach not only enhances consistency and reliability but also simplifies the development of complex, distributed actor-based applications, ensuring that system-wide operations adhere to ACID properties. In addition, Orleans has proven to be a user-friendly and high-performance actor framework, which enables us to develop complex functionalities on top of the actor abstraction and simplifies the development process. We envision the potential to implement more features on virtual actors while gaining good performance.

- **A novel log replication mechanism**. A planned future project is to build a read replica of the system by subscribing to the logs generated by the write transactions. This approach differs from the state machine replication (SMR) that is commonly adopted in deterministic databases. In SMR, an ordered sequence of transactions is replicated into replicas, which can form an up-to-date state by replaying the transactions in order. However, it is not suitable for our system because (1) We do not assume determinism in transaction logic, i.e., replaying a transaction may not end with an identical result. (2) It can be difficult to replicate transaction schedules under the hybrid approach. Therefore, we tend to adopt a novel log replication mechanism that replicates transactional logs of both PACT batches and ACTs from the primary to replicas. However, further research is required to develop and refine this mechanism fully. For example, it is not straightforward to rebuild a consistent system state using logs delivered across the network, which can be disordered and asynchronous.

- **Exploring data persisting methods**: In our implementation, transactional logs are always written into the local file system, where (1) the log record can only be retrieved by scanning the whole file line by line, (2) the log files are distributed across servers, (3) each log file is shared by a group of actors. This method brings difficulties for the system in doing garbage collection (clearing stale log records), failure recovery, or log retrieval. In the future, it is worth exploring different methods to persist data for our system that can better suit the actor model and bring less overhead.

Bibliography

[1] Why streams in orleans? https://learn.microsoft.com/en-us/dotnet/orleans/streaming/streams-why, October 2024.

[2] Actix. Actix documentation. https://actix.rs/docs/actix/actor, September 2024.

[3] ActorDB. Actordb documentation. https://www.actordb.com/docs-about.html, May 2024.

[4] Agha, G. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[5] Akka. Akka documentation. https://akka.io/, July 2021.

[6] Akka. Migration guide 2.3.x to 2.4.x. https://doc.akka.io/docs/akka/2.4/project/migration-guide-2.3.x-2.4.x.html, July 2021.

[7] Akka. Transactors (java). https://doc.akka.io/docs/akka/2.0.5/java/transactors.html, July 2021.

[8] Akka. Cluster sharding. https://doc.akka.io/docs/akka/current/typed/cluster-sharding.html#cluster-sharding, September 2024.

[9] Akka. Resolving conflicting updates. https://doc.akka.io/docs/akka/current/typed/replicated-eventsourcing.html#resolving-conflicting-updates, September 2024.

[10] Akka. Why modern systems need a new programming model. https://doc.akka.io/docs/akka/current/typed/guide/actors-motivation.html, October 2024.

[11] Alomari, M., Cahill, M., Fekete, A., and Rohm, U. The cost of serializability on platforms that use snapshot isolation. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering* (2008), pp. 576–585.

[12] Armstrong, J. A history of erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages* (2007).

[13] Athanassoulis, M., Johnson, R., Ailamaki, A., and Stoica, R. Improving oltp concurrency through early lock release. Tech. rep., EPFL, 2009.

[14] Athanassoulis, M., Johnson, R., Ailamaki, A., and Stoica, R. Improving oltp concurrency through early lock release. Tech. rep., EPFL, 2009.

[15] AWS. Amazon aurora pricing. https://aws.amazon.com/rds/aurora/pricing/, July 2021.

[16] Azure. Azure sql database pricing. https://azure.microsoft.com/en-us/pricing/details/azure-sql-database/single/, July 2021.

[17] Bagherzadeh, M., Fireman, N., Shawesh, A., and Khatchadourian, R. Actor concurrency bugs: a comprehensive study on symptoms, root causes, api usages, and differences. In *Proceedings of the ACM on Programming Languages* (2020), pp. 1–32.

[18] Bahssas, D. M., AlBar, A. M., and Hoque, M. R. Enterprise resource planning (erp) systems: Design, trends and deployment. *The International Technology Management Review* (2015), 72–81.

[19] Bailis, P., and Ghodsi, A. Eventual consistency today: Limitations, extensions, and beyond. *Communications of the ACM* (2013), 55–63.

[20] Bailis, P., Ghodsi, A., Hellerstein, J. M., and Stoica, I. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), pp. 761–772.

[21] Baker, J., Bond, C., Corbett, J. C., Furman, J., Khorlin, A., Larson, J., Leon, J.-M., Li, Y., Lloyd, A., and Yushprakh, V. Megastore: Providing scalable, highly available storage for interactive services. In *Fifth Biennial Conference on Innovative Data Systems Research (CIDR)* (2011), pp. 223–234.

[22] Barthels, C., Müller, I., Taranov, K., Alonso, G., and Hoefler, T. Strong consistency is not hard to get: Two-phase locking and two-phase commit on thousands of cores. In *Proceedings of the VLDB Endowment* (2019), pp. 2325–2338.

[23] Bauer, D. A., and Mäkiö, J. Actor4j: A software framework for the actor model focusing on the optimization of message passing. In *The Fourteenth Advanced International Conference on Telecommunications* (2018), pp. 125–134.

[24] Bernstein, P. A. Middleware: a model for distributed system services. *Communications of the ACM* (1996), 86–98.

[25] Bernstein, P. A. Actor-oriented database systems. In *IEEE 34th International Conference on Data Engineering (ICDE)* (2018), pp. 13–14.

[26] Bernstein, P. A. Resurrecting middle-tier distributed transactions. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* (2019), 3–6.

[27] Bernstein, P. A., Burckhardt, S., Bykov, S., Crooks, N., Faleiro, J. M., Kliot, G., Kumbhare, A., Rahman, M. R., Shah, V., Szekeres, A., and Thelin, J. Geo-distribution of actor-based services. In *Proceedings of the ACM on Programming Languages* (2017), pp. 1–26.

[28] Bernstein, P. A., and Bykov, S. Developing cloud services using the orleans virtual actor model. *IEEE Internet Computing* (2016), 71–75.

[29] Bernstein, P. A., Bykov, S., Geller, A., Kliot, G., and Thelin, J. Orleans: Distributed virtual actors for programmability and scalability. Tech. rep., Microsoft Research, 2014.

[30] Bernstein, P. A., Dashti, M., Kiefer, T., and Maier, D. Indexing in an actor-oriented database. In *Conference on Innovative Database Research (CIDR)* (2017).

[31] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency control and recovery in database systems.* Addison-Wesley Longman Publishing Co., Inc., 1987.

[32] Bernstein, P. A., and Newcomer, E. *Principles of transaction processing: for the systems professional.* Morgan Kaufmann Publishers Inc., 1996.

[33] Betts, D., Dominguez, J., Melnik, G., Simonazzi, F., and Subramanian, M. *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure.* Microsoft patterns practices, 2013.

[34] Bond, R. Solving a transactions performance mystery. https://dotnet.github.io/orleans/blog/solving-a-transactions-performance-mystery.html, July 2018.

[35] Bond, R. Grain migration, April 2022.

[36] Bragg, J. Orleans transaction deadlocks. https://github.com/dotnet/orleans/issues/5297, January 2019.

[37] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns.* Wiley Publishing, 1996.

[38] Bykov, S., Geller, A., Kliot, G., Larus, J. R., Pandya, R., and Thelin, J. Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (2011), pp. 1–14.

[39] CAF. The c++ actor framework. https://www.actor-framework.org/, July 2024.

[40] case study, A. Walmart boosts conversions by 20% with lightbend reactive platform. https://www.lightbend.com/case-studies/walmart-boosts-conversions-by-20-with-lightbend-reactive-platform, July 2021.

[41] Clebsch, S. W. *Pony: Co-designing a Type System and a Runtime*. PhD thesis, Imperial College London, 2017.

[42] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010).

[43] Cowling, J., and Liskov, B. Granola: Low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference* (2012), pp. 223–235.

[44] Crooks, N., Burke, M., and Cecchetti, E. Obladi: Oblivious serializable transactions in the cloud. In *Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation* (2018), pp. 727–743.

[45] Curino, C., Jones, E. P. C., Zhang, Y., and Madden, S. R. Schism: a workload-driven approach to database replication and partitioning.

[46] Dapr. Dapr documentation. https://docs.dapr.io/developing-applications/building-blocks/actors/, September 2024.

[47] Das, S., Agrawal, D., and Abbadi, A. E. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Transactions on Database Systems (TODS)* (2013), 1–45.

[48] Dayarathna, M., and Perera, S. Recent advancements in event processing. *ACM Computing Surveys (CSUR)* (2018), 1–36.

[49] dbyrne. Looking for alternatives of transactor. https://stackoverflow.com/questions/29154913/scala-replacement-for-akka-transactors, March 2015.

[50] Ding, B., Kot, L., and Gehrke, J. Improving optimistic concurrency control through transaction batching and operation reordering. In *Proceedings of the VLDB Endowment* (2018), pp. 169–182.

[51] Dong, Z., Wang, Z., Zhang, X., Xu, X., Zhao, C., Chen, H., Panda, A., and Li, J. Fine-grained re-execution for efficient batched commit of distributed transactions. In *Proceedings of the VLDB Endowment* (2023), pp. 1930–1943.

[52] Dong, Z.-Y., Tang, C.-Z., Wang, J.-C., Wang, Z.-G., Chen, H.-B., and Zang, B.-Y. Optimistic transaction processing in deterministic database. *Journal of Computer Science and Technology* (2020), 382–394.

[53] Dong, Z.-Y., Tang, C.-Z., Wang, J.-C., Wang, Z.-G., Chen, H.-B., and Zang, B.-Y. Optimistic transaction processing in deterministic database. *Journal of Computer Science and Technology* (2020), 382–394.

[54] Elbagir, F. A., and Khanfar, A. K. A survey of commit protocols in distributed real time database systems. *International Journal of Emerging Trends Technology in Computer Science 31* (2016), 61–66.

[55] Eldeeb, T., and Bernstein, P. A. Transactions for distributed actors in the cloud. Tech. rep., Microsoft Research, 2016.

[56] Eldeeb, T., Burckhardt, S., Bond, R., Cidon, A., Yang, J., and Bernstein, P. A. Cloud actor-oriented database transactions in orleans. In *Proceedings of the VLDB Endowment* (2024), pp. 3720–3730.

[57] Eldeeb, T., Xie, X., Bernstein, P. A., Cidon, A., and Yang, J. Chardonnay: Fast and general datacenter transactions for on-disk databases. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (2023), pp. 343–360.

[58] Elixir. Elixir documentation. https://elixir-lang.org/, July 2021.

[59] Erlang. Erlang documentation. https://www.erlang.org/, July 2021.

[60] Erlang. Who uses erlang for product development? http://erlang.org/faq/introduction.html, July 2021.

[61] Erlang. Who uses erlang for product development? http://erlang.org/faq/introduction.html, July 2021.

[62] Erlang. Erlang/otp system documentation. https://www.erlang.org/doc/system/readme.html, September 2024.

[63] Faleiro, J. M., and Abadi, D. J. Rethinking serializable multiversion concurrency control. In *Proceedings of the VLDB Endowment* (2015), pp. 1190–1201.

[64] Faleiro, J. M., Abadi, D. J., and Hellerstein, J. M. High performance transactions via early write visibility. In *Proceedings of the VLDB Endowment* (2017), pp. 613–624.

[65] Faleiro, J. M., Abadi, D. J., and Hellerstein, J. M. High performance transactions via early write visibility. In *Proceedings of the VLDB Endowment* (2017), pp. 613–624.

[66] Floratos, S., Zhang, Y., Yuan, Y., Lee, R., and Zhang, X. Sqloop: High performance iterative processing in data management. In *IEEE 38th International Conference on Distributed Computing Systems* (2018), pp. 1039–1051.

[67] Fox, A., and Brewer, E. A. Harvest, yield, and scalable tolerant systems. In *Proceedings of the seventh workshop on hot topics in operating systems* (1999), pp. 174–178.

[68] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[69] Garcia-Molina, H., and Salem, K. Sagas. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data* (1987), pp. 249–259.

[70] Gear. Actor model. https://wiki.gear-tech.io/docs/gear/technology/actor-model, September 2023.

[71] Gos, K., and Zabierowski, W. The comparison of microservice and monolithic architecture. In *IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)* (2020), pp. 150–153.

[72] Guerraoui, R., and Wang, J. How fast can a distributed transaction commit? In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (2017), pp. 107–122.

[73] Guo, Z., Wu, K., Yan, C., and Yu, X. Releasing locks as early as you can: Reducing contention of hotspots by violating two-phase locking. In *Proceedings of the 2021 International Conference on Management of Data* (2021), pp. 658–670.

[74] H2. In-memory databases. https://www.h2database.com/html/features.html#in_memory_databases, September 2024.

[75] Harding, R., Aken, D. V., Pavlo, A., and Stonebraker, M. An evaluation of distributed concurrency control. In *Proceedings of the VLDB Endowment* (2017), pp. 553–564.

[76] Hasselbring, W., and Steinacker, G. Microservice architectures for scalability, agility and reliability in e-commerce. In *IEEE International Conference on Software Architecture Workshops (ICSAW)* (2017), pp. 243–246.

[77] Helland, P. Life beyond distributed transactions: An apostate's opinion. *Queue* (2016), 69–98.

[78] Hewitt, C., Bishop, P., and Steiger, R. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence* (1973), pp. 235–245.

[79] Irby, B. Orleans slow with minimalistic use case. https://stackoverflow.com/questions/58308230/orleans-akka-net-problem-with-understanding-the-actor-model, October 2024.

[80] Ireland, C., Bowers, D., Newton, M., and Waugh, K. A classification of object-relational impedance mismatch. In *2009 First International Conference on Advances in Databases, Knowledge, and Data Applications* (2009), pp. 36–43.

[81] Issa, S., Viegas, M., Raminhas, P., Machado, N., Matos, M., and Romano, P. Exploiting symbolic execution to accelerate deterministic databases. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)* (2020), pp. 678–688.

[82] Katsarakis, A., Ma, Y., Tan, Z., Bainbridge, A., Balkwill, M., Dragojevic, A., Grot, B., Radunovic, B., and Zhang, Y. Zeus: Locality-aware distributed transactions. In *Proceedings of the Sixteenth European Conference on Computer Systems* (2021), pp. 145–161.

[83] Kleppmann, M., Beresford, A. R., and Svingen, B. Online event processing. *Communications of the ACM* (2019), 43–49.

[84] Kraft, P., Kazhamiaka, F., Bailis, P., and Zaharia, M. Data-parallel actors: A programming model for scalable query serving systems. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation* (2022), pp. 1059–1074.

[85] Lai, Z., Fan, H., Zhou, W., Ma, Z., Peng, X., Li, F., and Lo, E. Knock out 2pc with practicality intact: a high-performance and general distributed transaction protocol. In *International Conference on Data Engineering (ICDE)* (2023), pp. 2317–2331.

[86] Laigner, R., Kalinowski, M., Diniz, P., Barros, L., Cassino, C., Lemos, M., Arruda, D., Lifschitz, S., and Zhou, Y. From a monolithic big data system to a microservices event-driven architecture. In *46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (2020), pp. 213–220.

[87] Laigner, R., Zhang, Z., Liu, Y., Gomes, L. F., and Zhou, Y. A benchmark for data management challenges in microservices. In *Arxiv* (2024).

[88] Laigner, R., and Zhou, Y. Benchmarking data management systems for microservices. In *IEEE 40th International Conference on Data Engineering (ICDE)* (2024).

[89] Laigner, R., Zhou, Y., Salles, M. A. V., Liu, Y., and Kalinowski, M. Data management in microservices: State of the practice, challenges, and research directions. In *Proceedings of the VLDB Endowment* (2021), pp. 3348–3361.

[90] Lampson, B., and Sturgis, H. E. Crash recovery in a distributed data storage system. Tech. rep., Microsoft Research, 1979.

[91] Li, Z., Romano, P., and Roy, P. V. Sparkle: Speculative deterministic concurrency control for partially replicated transactional stores. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2019).

[92] Lightbend. Distributed data. https://doc.akka.io/docs/akka/current/typed/distributed-data.html#distributed-data, May 2024.

[93] Lin, Q., Chang, P., Chen, G., Ooi, B. C., Tan, K.-L., and Wang, Z. Towards a non-2pc transaction management in distributed database systems. In *Proceedings of the 2016 International Conference on Management of Data* (2016), pp. 1659–1674.

[94] Lin, Y.-S., Pi, S.-K., Liao, M.-K., Tsai, C., Elmore, A., and Wu, S.-H. Mgcrab: Transaction crabbing for live migration in deterministic database systems. In *Proceedings of the VLDB Endowment* (2019), pp. 597–610.

[95] Lin, Y.-S., Tsai, C., Lin, T.-Y., Chang, Y.-S., and Wu, S.-H. Don't look back, look into the future: Prescient data partitioning and migration for deterministic database systems. In *Proceedings of the 2021 International Conference on Management of Data* (2021).

[96] Liskov, B., and Shrira, L. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI)* (1988), pp. 260–267.

[97] Liu, X., Heo, J., and Sha, L. Modeling 3-tiered web applications. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (2005).

[98] Liu, Y. Why i get low throughput with orleanstransaction. https://github.com/dotnet/orleans/issues/6912, January 2021.

[99] Liu, Y., Su, L., Shah, V., Zhou, Y., and Salles, M. A. V. Hybrid deterministic and nondeterministic execution of transactions in actor systems. In *Proceedings of the 2022 International Conference on Management of Data* (2022), pp. 65–78.

[100] Lu, Y., Yu, X., Cao, L., and Madden, S. Aria: a fast and practical deterministic oltp database. In *Proceedings of the VLDB Endowment* (2020), pp. 2047–2060.

[101] Lu, Y., Yu, X., Cao, L., and Madden, S. Epoch-based commit and replication in distributed oltp databases. In *Proceedings of the VLDB Endowment* (2021), pp. 743–756.

[102] Lu, Y., Yu, X., Cao, L., and Madden, S. Epoch-based commit and replication in distributed oltp databases. In *Proceedings of the VLDB Endowment* (2021), pp. 743–756.

[103] Luo, Q., Krishnamurthy, S., Mohan, C., Pirahesh, H., Woo, H., Lindsay, B. G., and Naughton, J. F. Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data* (2002), pp. 600–611.

[104] Martin, J. *Managing the Data Base Environment*. Prentice Hall PTR, 1983.

[105] mathdotnet. Zipf. https://numerics.mathdotnet.com/api/MathNet.Numerics.Distributions/Zipf.htm, 2021.

[106] Mattsson, H., Nilsson, H., and Wikström, C. Mnesia - a distributed robust dbms for telecommunications applications. In *Practical Aspects of Declarative Languages* (1998), pp. 152–163.

[107] Mehdi, S. A., Hwang, D., Peter, S., and Alvisi, L. Scaledb: A scalable, asynchronous in-memory database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (2023), pp. 361–376.

[108] Memcached. Memcached wiki. https://github.com/memcached/memcached/wiki, September 2024.

[109] Mertz, J., and Nunes, I. Understanding application-level caching in web applications: A comprehensive introduction and survey of state-of-the-art approaches. *ACM Computing Surveys (CSUR)* (2017), 1–34.

[110] Microsoft. Async query and save. https://learn.microsoft.com/en-us/ef/ef6/fundamentals/async?source=recommendations, September 2024.

[111] Mohan, C., and Lindsay, B. Efficient commit protocols for the tree of processes model of distributed transactions. *ACM SIGOPS Operating Systems Review* (1985), 40–52.

[112] Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., and Stoica, I. Ray: A distributed framework for emerging ai applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation* (2018), pp. 561–577.

[113] Mu, S., Cui, Y., Zhang, Y., Lloyd, W., and Li, J. Extracting more concurrency from distributed transactions. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation* (2014), pp. 479–494.

[114] Murdock. Orleans slow with minimalistic use case. https://stackoverflow.com/questions/74310628/orleans-slow-with-minimalistic-use-case, October 2024.

[115] Newell, A., Kliot, G., Menache, I., Gopalan, A., Akiyama, S., and Silberstein, M. Optimizing distributed actor systems for dynamic interactive services. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), pp. 1–15.

[116] Nguyen, C. D. T., Miller, J. K., and Abadi, D. J. Detock: High performance multi-region transactions at scale. *Proceedings of the ACM on Management of Data* (2023), 1–27.

[117] on Serverless Stateful Functions, D. T. Martijn de heus and kyriakos psarakis and marios fragkoulis and asterios katsifodimos. In *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems* (2021), pp. 31–42.

[118] Orbit. Orbit documentation. https://www.orbit.cloud/orbit/, July 2021.

[119] Orleans. Orleans transactions. https://dotnet.github.io/orleans/docs/grains/transactions.html, July 2021.

[120] Orleans. Grain placement. https://learn.microsoft.com/en-us/dotnet/orleans/grains/grain-placement, September 2024.

[121] Orleans. Replicated grains. https://learn.microsoft.com/en-us/dotnet/orleans/grains/event-sourcing/replicated-instances, September 2024.

[122] Orleans, M. Orleans documentation. https://learn.microsoft.com/en-us/dotnet/orleans/overview, September 2024.

[123] Orleans, M. Reentrancy. https://learn.microsoft.com/en-us/dotnet/orleans/grains/request-scheduling#reentrancy, September 2024.

[124] Pavlo, A., Curino, C., and Zdonik, S. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), pp. 61–72.

[125] Perez-Sorrosal, F., Patiño-Martinez, M., Jimenez-Peris, R., and Kemme, B. Elastic si-cache: consistent and scalable caching in multi-tier architectures. *The VLDB Journal* (2011), 841–865.

[126] Persson, P., and Angelsmark, O. Calvin–merging cloud and iot. *Procedia Computer Science 52* (2015), 210–217.

[127] Ports, D. R. K. *Application-Level Caching with Transactional Consistency*. Ph.D., MIT, Cambridge, MA, USA, 2012.

[128] Prasaad, G., Cheung, A., and Suciu, D. Handling highly contended oltp workloads using fast dynamic partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), pp. 527–542.

[129] Preguiça, N., Baquero, C., and Shapiro, M. *Conflict-Free Replicated Data Types CRDTs*. Springer International Publishing, 2018, pp. 1–10.

[130] Prickett, S. What is redis?: An overview. https://redis.io/learn/develop/node/nodecrashcourse/whatisredis, February 2024.

[131] Proto.Actor. Proto.actor documentation. https://proto.actor/docs/, September 2024.

[132] Ptaszek, M. Chat service architecture: Servers. https://technology.riotgames.com/news/chat-service-architecture-servers, September 2015.

[133] Pulsar. Pulsar documentation. https://quantmind.github.io/pulsar/design.html#actors, September 2024.

[134] Pyactor. Pyactor documentation. https://pyactor.readthedocs.io/en/master/, September 2024.

[135] Pykka. Pykka documentation. https://pykka.readthedocs.io/stable/api/actors/, September 2024.

[136] Qadah, T. Q-store: Distributed, multi-partition transactions via queue-oriented execution and communication, March 2020.

[137] Qadah, T. M., Gupta, S., and Sadoghi, M. Q-store: Distributed, multi-partition transactions via queue-oriented execution and communication. In *EDBT* (2020), pp. 73–84.

[138] Qadah, T. M., and Sadoghi, M. Quecc: A queue-oriented, control-free concurrency architecture. In *Proceedings of the 19th International Middleware Conference* (2018), pp. 13–25.

[139] Qin, D., Brown, A. D., and Goel, A. Caracal: Contention management with deterministic concurrency control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 180–194.

[140] Rafiq. How to handle multiple update event when there is more then one replica of a pod. https://stackoverflow.com/questions/67612615/how-to-handle-multiple-update-event-when-there-is-more-then-one-replica-of-a-pod, May 2021.

[141] Ramachandra, K., Chavan, M., Guravannavar, R., and Sudarshan, S. Program transformations for asynchronous and batched query submission. *IEEE Transactions on Knowledge and Data Engineering* (2015), 531–544.

[142] Redis. Write-behind, write-through, and read-through caching. https://redis.io/docs/latest/operate/oss_and_stack/stack-with-enterprise/gears-v1/jvm/recipes/write-behind/, September 2024.

[143] Reiko, H., Rui, C., Carlos, M., Mohammad, E.-R., Georgios, K., and Luis, A. *Software Evolution*. Springer Berlin Heidelberg, 2008, ch. Architectural Transformations: From Legacy to Three-Tier and Services.

[144] Ren, K., Diamond, T., Abadi, D. J., and Thomson, A. Low-overhead asynchronous checkpointing in main-memory database systems. In *Proceedings of the 2016 International Conference on Management of Data* (2016), pp. 539–1551.

[145] Ren, K., Li, D., and J.Abadi, D. Slog: Serializable, low-latency, geo-replicated transactions. In *Proceedings of the VLDB Endowment* (2019), pp. 1747–1761.

[146] Ren, K., Thomson, A., and Abadi, D. J. An evaluation of the advantages and disadvantages of deterministic database systems. In *Proceedings of the VLDB Endowment* (2014), pp. 821–832.

[147] Riker. Riker documentation. https://riker.rs/actors/#actors, September 2024.

[148] Rosenkrantz, D. J., Stearns, R. E., and Lewis, P. M. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems* (1978), 178–198.

[149] saeedakhter. Ensuring data consistency when a transaction spans state in two grains. https://github.com/dotnet/orleans/issues/1090, November 2015.

[150] Sang, B., Petri, G., Ardekani, M. S., Ravi, S., and Eugster, P. Programming scalable cloud services with aeon. In *Proceedings of the 17th International Middleware Conference* (2016), pp. 1–14.

[151] Sang, B., Roman, P.-L., Eugster, P., Lu, H., Ravi, S., and Petri, G. Plasma: Programmable elasticity for stateful cloud computing applications. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–15.

[152] Schantz, R. E., and Schmidt, D. C. Middleware for distributed systems: Evolving the common structure for network-centric applications. *Encyclopedia of Software Engineering* (2001), 1–9.

[153] Schmidl, S., Schneider, F., and Papenbrock, T. An actor database system for akka. In *Datenbanksysteme für Business, Technologie und Web* (2019).

[154] Shah, V., and Salles, M. A. V. Actor-relational database systems: A manifesto. *arXiv* (2017).

[155] Shah, V., and Salles, M. A. V. Reactors: A case for predictable, virtualized actor database systems. In *Proceedings of the 2018 International Conference on Management of Data* (2018), pp. 259–274.

[156] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems* (2011), pp. 386–400.

[157] Shasha, D., Llirbat, F., Simon, E., and Valduriez, P. Transaction chopping: algorithms and performance studies. *ACM Transactions on Database Systems* (1995), 325–363.

[158] Soethout, T., Vinju, J. J., and van der Storm, T. Path-sensitive atomic commit: Local coordination avoidance for distributed transactions (technical report). *CoRR* (2019).

[159] Soisalon-Soininen, E., and Ylönen, T. Partial strictness in two-phase locking. In *Proceedings of the 5th International Conference on Database Theory* (1995), pp. 139—-147.

[160] Somuah, H. Using project "orleans" in halo. https://hoopsomuah.com/2014/04/06/using-project-orleans-in-halo/, April 2014.

[161] SQLite. In-memory databases. https://www.sqlite.org/inmemorydb.html, September 2024.

[162] Srinivasan, S., and Mycroft, A. Kilim: Isolation-typed actors for java. In *ECOOP 2008 – Object-Oriented Programming* (2008), pp. 104–128.

[163] Srinivasan, V., Bulkowski, B., Chu, W.-L., Sayyaparaju, S., Gooding, A., Iyer, R., Shinde, A., and Lopatic, T. Aerospike: architecture of a real-time operational dbms. *Proceedings of the VLDB Endowment* (2016), 1389—1400.

[164] Stefanko, M., Chaloupka, O., and Rossi, B. The saga pattern in a reactive microservices environment. In *Proceedings of the 14th International Conference on Software Technologies* (2019), pp. 483–490.

[165] Stonebraker, M., and Weisberg, A. The voltdb main memory dbms. *IEEE Data Eng. Bull.* (2013), 21–27.

[166] Swalens, J., Koster, J. D., and Meuter, W. D. Transactional actors: Communication in transactions. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems* (2017), pp. 31–41.

[167] Taft, R., Sharif, I., Matei, A., VanBenschoten, N., Lewis, J., Grieger, T., Niemi, K., Woods, A., Birzin, A., Poss, R., Bardea, P., Ranade, A., Darnell, B., Gruneir, B., Jaffray, J., Zhang, L., and Mattis, P. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), pp. 1493–1509.

[168] Tanenbaum, A. S., and van Steen, M. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., 2006.

[169] @theotown. How reactive systems help paypal's squbs scale to billions of transactions daily. https://www.lightbend.com/blog/how-reactive-systems-help-paypal-squbs-scale-to-billions-of-transactions-daily, June 2016.

[170] Thespian. Thespian documentation. https://thespianpy.com/doc/, September 2024.

[171] Thomson, A., and Abadi, D. J. The case for determinism in database systems. In *Proceedings of the VLDB Endowment* (2010), pp. 70–80.

[172] Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P., and Abadi, D. J. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), pp. 1–12.

[173] Torres, A., Galante, R., Pimenta, M. S., and Martins, A. J. B. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Information and Software Technology* (2017), 1–18.

[174] TPCC. Tpc-c is an on-line transaction processing benchmark. http://www.tpc.org/tpcc/, July 2021.

[175] user, A. Transactors and stm are gone. what conception to use instead? https://groups.google.com/g/akka-user/c/XS-Pk3SOzbw?pli=1, July 2021.

[176] Viennot, N., Lécuyer, M., Bell, J., Geambasu, R., and Nieh, J. Synapse: A microservices architecture for heterogeneous-database web applications. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), pp. 1–16.

[177] Wang, Y. *Scalable and Reactive Data Management for Mobile Internet-of-Things Applications with Actor-Oriented Databases*. PhD thesis, University of Copenhagen, 2021.

[178] Wang, Y., Reis, J. C. D., Borggren, K. M., Salles, M. A. V., Medeiros, C. B., and Zhou, Y. Modeling and building iot data platforms with actor-oriented databases. In *Proceedings of the 22nd International Conference on Extending Database Technology* (2019), pp. 512–523.

[179] Wang, Z., Mu, S., Cui, Y., Yi, H., Chen, H., and Li, J. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data* (2016), pp. 1643–1658.

[180] Wu, C., Faleiro, J. M., Lin, Y., and Hellerstein, J. M. Anna: A kvs for any scale. In *IEEE Transactions on Knowledge and Data Engineering* (2021), pp. 344–358.

[181] Wu, S.-H., Feng, T.-Y., Liao, M.-K., Pi, S.-K., and Lin, Y.-S. T-part: Partitioning of transactions for forward-pushing in deterministic database systems. In *Proceedings of the 2016 International Conference on Management of Data* (2016).

[182] Yao, C., Agrawal, D., Chen, G., Lin, Q., Ooi, B. C., Wong, W.-F., and Zhang, M. Exploiting single-threaded model in multi-core in-memory systems. *IEEE Transactions on Knowledge and Data Engineering* (2016), 2635–2650.

[183] You, J., Wu, J., Jin, X., and Chowdhury, M. Ship compute or ship data? why not both? In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (2021), pp. 633–651.

[184] Zamanian, E., Binnig, C., Harris, T., and Kraska, T. The end of a myth: Distributed transactions can scale. In *Proceedings of the VLDB Endowment* (2017), pp. 685–696.

[185] Zhang, J., Huang, K., Wang, T., and Lv, K. Skeena: Efficient and consistent cross-engine transactions. In *Proceedings of the 2022 International Conference on Management of Data* (2022), pp. 34–48.

[186] Zhang, Y., Power, R., Zhou, S., Sovran, Y., Aguilera, M. K., and Li, J. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), pp. 276–291.

[187] Zhou, X., Yu, X., Graefe, G., and Stonebraker, M. Lotus: Scalable multi-partition transactions on single-threaded partitioned databases. In *Proceedings of the VLDB Endowment* (2022), pp. 2939–2952.

Appendix A

Proof for Hybrid Execution Correctness

In `Snapper`, the system state is a collection of the state of every named actor, and a transaction is a series of method invocations performed on one or more actors. Each method invocation can perform one or more `Read` or `ReadWrite` operations on an actor and invoke methods on some other actors via asynchronous RPCs. A transaction can invoke methods on the same actor multiple times.

We note that `Read` or `ReadWrite` operations on an actor always touch the whole of its state, so in the following, we make no distinction between the notion of an actor and the data in its state. We use $r_i[x]$ or $w_i[x]$ to denote a `Read` or a `ReadWrite`, respectively, operation performed by transaction T_i on actor x. We focus on the concept of conflict serializability [31], so the `ReadWrite` operation can be regarded as a blind `Write` operation without being distinguished from a read-modify-write. We say any two read or write operations are conflicting if they are applied on the same actor, and one of them is a write.

Besides, in `Snapper`, PACTs are committed in batches, thus a PACT batch can be considered as one large transaction. We denote an ACT as T_i^a, where i corresponds to the ACT's *tid* (a non-negative integer, $tid \in Z^*$), and a batch as T_j^b, where j corresponds to the batch's *bid* (similarly $bid \in Z^*$). `Snapper` guarantees that no two ACTs have the same *tid* and no two batches have the same *bid*. When we refer to T_i without further qualification, we denote either an ACT or a batch. We assume, without loss of generality, that the transaction identifiers corresponding to *tid*s or *bid*s be taken from disjoint subsets of Z^*.

Before we prove the correctness of `Snapper`'s hybrid processing, we first define some key concepts similar to the formalism introduced by [31] for the classic transactional model. Definitions 2, 3, and 4 reuse the definitions of transaction, serialization graph, and history introduced in [31] with necessary

adaptations to Snapper's context and introduce important notation. We state these definitions here to make this appendix self-contained.

Definition 2 *In Snapper, a transaction T_i is a partial order with ordering relation $<_i$ where:*

1. *$T_i \subseteq \{r_i[x], w_i[x] \mid x \text{ is an actor}\} \cup \{a_i, c_i\}$, where a_i and c_i denote abort or commit, respectively;*
2. *$a_i \in T_i$ iff $c_i \notin T_i$;*
3. *if t is c_i or a_i (whichever is in T_i), for any other operation $p \in T_i$, $p <_i t$;*
4. *if $r_i[x]$, $w_i[x] \in T_i$, then either $r_i[x] <_i w_i[x]$ or $w_i[x] <_i r_i[x]$.*

To simplify the following discussion, we assume that a transaction T_i does not contain multiple operations of the same type on the same actor as in [31, page 27]. All the following results we get do not depend on this assumption.

Definition 3 (from [31]) *Let $T = \{T_1, T_2, ..., T_n\}$ be a set of transactions. A complete history H over T is a partial order with ordering relation $<_H$ where:*

1. *$H = \cup_{i=1}^{n} T_i$;*
2. *$\cup_{i=1}^{n} <_i \subseteq <_H$;*
3. *for any two conflicting operations p, $q \in H$, either $p <_H q$ or $q <_H p$.*

Definition 4 *The serialization graph SG for H, denoted $SG(H)$, is a directed graph, including a set of nodes V and edges E:*

1. *$V = \{T_i | T_i \subseteq H \text{ is a transaction} \wedge c_i \in T_i\}$*
2. *$E = \{T_i \rightarrow T_j | T_i \text{ and } T_j \text{ are different transactions, and there exist } o_i \in T_i, o_j \in T_j \text{ such that } o_i <_H o_j\}$*

In the following, we slightly abuse notation by referring to $T_i \in SG(H)$ as a transaction in the set of nodes V of $SG(H)$. When the context is clear, we also refer to $T_i \rightarrow T_j$ without further qualification to denote an edge in the set of edges of $SG(H)$. Furthermore, we assume that histories generated by Snapper's hybrid processing always include at least one ACT transaction and one PACT batch.

To check global serializability under hybrid transaction processing, Snapper introduces the concepts of BeforeSet and AfterSet, which contain the scheduling information of each ACT.

Definition 5 *Given a history H generated by* ***Snapper****'s hybrid processing and the corresponding serialization graph $SG(H)$, $\forall T_i^a \in SG(H)$, its* `BeforeSet` *($BS_{T_i^a}$) and* `AfterSet` *($AS_{T_i^a}$) are defined as:*

1. $BS_{T_i^a} = \{j|$ *there exists a path* $T_j^b \rightarrow ... \rightarrow T_i^a\}$
2. $AS_{T_i^a} = \{j|T_i^a \rightarrow T_j^b\}$

In addition, $max(BS_{T_i^a})$ and $min(AS_{T_i^a})$ are the maximum and minimum numbers (bids) in $BS_{T_i^a}$ and $AS_{T_i^a}$, respectively. If $BS_{T_i^a} =$, $max(BS_{T_i^a}) = -1$. Similarly, if $AS_{T_i^a} =$, $min(AS_{T_i^a}) = -1$.

Given a history H generated by `Snapper`'s hybrid processing and the corresponding serialization graph $SG(H)$, if $T_{i_1}^a \rightarrow T_{i_2}^a$, then $max(BS_{T_{i_1}^a}) \leq max(BS_{T_{i_2}^a})$.

If $BS_{T_{i_1}^a} =$, then $max(BS_{T_{i_1}^a}) = -1 \leq max(BS_{T_{i_2}^a})$. Otherwise, according to definition 5, $\forall T_j^b \in BS_{T_{i_1}^a}$, there exists a path from T_j^b to $T_{i_1}^a$ which can be extended by adding one more edge $T_{i_1}^a \rightarrow T_{i_2}^a$. Thus there is also a path from T_j^b to $T_{i_2}^a$. In another word $BS_{T_{i_1}^a} \subseteq BS_{T_{i_2}^a}$, so $max(BS_{T_{i_1}^a}) \leq max(BS_{T_{i_2}^a})$.

We propose Theorem 3 below to prove that `Snapper`'s hybrid processing preserves conflict serializability for all concurrent transactions. Our proof relies on the serializability theorem (Theorem 2), which has been proven in [31].

Theorem 2 (from [31]) *A history H is conflict serializable iff $SG(H)$ is acyclic.*

Theorem 3 *A history H generated by* ***Snapper****'s hybrid processing is conflict serializable if:*

(1) $\forall T_{j_1}^b \rightarrow T_{j_2}^b$, $j_1 < j_2$;

(2) the execution of all T_i^a is conflict serializable;

(3) $\forall T_i^a \in SG(H)$, $max(BS_{T_i^a}) < min(AS_{T_i^a})$.

Here we prove that when the three conditions are met, then $SG(H)$ can be topologically sorted, which means that $SG(H)$ is acyclic and thus H is conflict serializable. More specifically, we first assign a unique rational number for each transaction T_i by applying a function $\mathcal{N}(T_i)$. Then, we take all transactions in ascending order of the assigned numbers to obtain a topological sort of $SG(H)$. To realize this proposed construction, we need to prove that given the three stated conditions, $\forall T_i \rightarrow T_j$, $\mathcal{N}(T_i) < \mathcal{N}(T_j)$.

Before defining $\mathcal{N}$, we introduce new transaction identifiers to all T_i^a corresponding to a valid serialization order. According to condition (2) above and Theorem 2, if the execution of all T_i^a is conflict serializable, then the induced

sub-graph $SG(H)[\{T_i^a | T_i^a \in SG(H)\}]$ where the vertices consist of all T_i^a is acyclic. Thus, this induced sub-graph can be topologically sorted. Suppose we have $m \in Z^+$ ACT transactions T_i^a, and $T_{i_1}^a$, $T_{i_2}^a$, ..., $T_{i_m}^a$ is such a topological sort. We can relabel the T_i^a with the identifiers in this topological sort so that now we know that $\forall T_{i_{k_1}} \rightarrow T_{i_{k_2}}$, $k_1 < k_2$, where $k_1, k_2 \in [1, m]$.

The function $\mathcal{N} : T \mapsto Q$ is now defined as follows:

- $\forall T_j^b \in SG(H)$, $\mathcal{N}(T_j^b) = j$
- $\forall T_{i_k}^a \in SG(H)$, $k \in [1, m]$, $\mathcal{N}(T_{i_k}^a) = max(BS_{T_{i_k}^a}) + \frac{k}{m+1}$

Now we prove that $\forall T_i \rightarrow T_j$, $\mathcal{N}(T_i) < \mathcal{N}(T_j)$. We divide all $\rightarrow$ edges into four cases:

1. $\forall T_{j_1}^b \rightarrow T_{j_2}^b$,
$$\mathcal{N}(T_{j_1}^b) = j_1, \mathcal{N}(T_{j_2}^b) = j_2$$
With condition (1) above, $j_1 < j_2$, so $\mathcal{N}(T_{j_1}^b) < \mathcal{N}(T_{j_2}^b)$.

2. $\forall T_{i_{k_1}}^a \rightarrow T_{i_{k_2}}^a$,
$$\mathcal{N}(T_{i_{k_1}}^a) = max(BS_{T_{i_{k_1}}^a}) + \frac{k_1}{m+1}$$
$$\mathcal{N}(T_{i_{k_2}}^a) = max(BS_{T_{i_{k_2}}^a}) + \frac{k_2}{m+1}$$

In the topological sort of $SG(H)[\{T_i^a | T_i^a \in SG(H)\}]$ discussed above, $k_1 < k_2$, thus $\frac{k_1}{m+1} < \frac{k_2}{m+1}$, and according to Lemma A, $max(BS_{T^a_{i_{k_1}}}) \leq max(BS_{T^a_{i_{k_2}}})$, so

$$\mathcal{N}(T^a_{i_{k_1}}) < \mathcal{N}(T^a_{i_{k_2}})$$

3. $\forall T_j^b \to T_{i_k}^a$,

$$\mathcal{N}(T_j^b) = j$$

$$\mathcal{N}(T_{i_k}^a) = max(BS_{T^a_{i_k}}) + \frac{k}{m+1}$$

According to Definition 5, $j \in BS_{T^a_{i_k}}$, $j \leq max(BS_{T^a_{i_k}})$, thus

$$\mathcal{N}(T_j^b) < \mathcal{N}(T_{i_k}^a)$$

4. $\forall T_{i_k}^a \to T_j^b$,

$$\mathcal{N}(T_{i_k}^a) = max(BS_{T^a_{i_k}}) + \frac{k}{m+1}$$

$$\mathcal{N}(T_j^b) = j$$

According to Definition 5, $j \in AS_{T^a_{i_k}}$, $min(AS_{T^a_{i_k}}) \leq j$. And according to condition (3) above, $max(BS_{T^a_{i_k}}) < min(AS_{T^a_{i_k}})$, then $max(BS_{T^a_{i_k}}) < j$. And $\frac{k}{m+1} < 1$, so

$$\mathcal{N}(T_{i_k}^a) < \mathcal{N}(T_j^b)$$