# TTCC : Transactional-Turn Causal Consistency

Benoît Martin



Thèse présentée pour l'obtention du grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Spécialité

Ingénierie / Systèmes Informatiques

École doctorale

Informatique, Télécommunication et Électronique Paris (ED130)

# TTCC: Transactional-Turn Causal Consistency

---

Benoît Martin

Soutenue publiquement le : *21 Avril 2023*

Devant un jury composé de :

| | |
|---|---|
| **Achour Mostefaoui**, Professeur, Université de Nantes | *Rapporteur* |
| **Gaël Thomas**, Professeur, Telecom SudParis | *Rapporteur* |
| **Bernd Amann**, Professeur, Sorbonne Université, LIP6 | *Examinateur* |
| **Annette Bieniusa**, Professeure, Université Technique de Kaiserslautern | *Examinateur* |
| **Carla Ferreira**, Maîtresse de Conférences, Université NOVA de Lisbon | *Examinateur* |
| **Peter Van Roy**, Professeur, Université Catholique de Louvain | *Examinateur* |
| **Marc Shapiro**, Directeur de Recherche Emérite, SU, LIP6, Inria | *Encadrant de thèse* |
| **Mesaac Makpangou**, Directeur de Recherche, SU, LIP6, Inria | *Directeur de thèse* |

*À Maïa et Mathilde*

# Remerciements

Ce fut une longue aventure. Après un parcours non-standard, je suis extrêmement heureux d'avoir réussi à terminer cette thèse. Ces années n'auraient pu se faire sans toutes les personnes que j'ai pu rencontrer durant mon parcours. Je remercie toutes ces personnes qui m'ont soutenu durant ces années.

Je tiens tout d'abord à remercier mes rapporteurs, Gaël Thomas et Achour Mostefaoui, pour le temps qu'ils ont consacré à la lecture attentive et à l'évaluation de ce rapport. Merci également à tous les autres membres du jury, Bernd Amann, Annette Bieniusa, Carla Ferreira, Peter Van Roy et Mesaac Makpangou. Un grand merci également à Nikolaos Georgantas pour avoir suivi mes avancées tout au long de cette thèse.

Ensuite, cette thèse n'aurait pas été possible sans la supervision de Marc Shapiro. Merci à toi, Marc pour ton enseignement, ta confiance et de m'avoir accompagné toutes ces années. Je suis plus que reconnaissant pour tout ce que j'ai appris à tes côtés, et je m'efforcerai à garder la rigueur que tu m'as enseignée pour faire de la recherche en systèmes.

Je souhaite ensuite remercier les membres du LIP6, en particulier les permanents de l'équipe Delys. Merci à Pierre Sens et Julien Sopena pour avoir su veiller sur l'équipe et pour vos nombreux conseils.

Ces dernières années m'ont fait voir plusieurs générations d'étudiants et d'ingénieurs et de postdocs que je tiens à remercier. Un énorme merci à mes camarades doctorants, Sreeja, Dimitri, Francis, Ilyas et Jonathan avec qui nous avons eu de nombreuses discussions et partagé de bons moments. Sans oublier Laurent, Saalik et Ayush, avec qui j'ai l'honneur de terminer ma thèse.

Je souhaite remercier l'équipe Aramis, avec qui j'ai passé des années incroyables. Que ce soit sur le plan personnel, amical et professionnel. Merci à Stanley, Olivier et Emmanuelle pour m'avoir accueilli et permis d'intégrer une équipe de recherche fantastique. Merci à Alexandre B, Maxime et Benjamin pour avoir partagé votre travail avec moi. Sans votre générosité et votre pédagogie, je n'aurais peut-être pas sauté le pas vers cette thèse. Merci à vous. Merci également aux doctorants, ingénieurs et chercheurs, Alexandre R, Raphaël, Elina, Tiziana, Catalina, Simona, Igor, Manon,

Ninon, Arnaud M, Arnaud V, Vincent, Hoa, Jorge, Jérémy et Mauricio, merci pour votre amitié et pour ces bons moments que nous avons passés ensemble.

Ensuite, je souhaite remercier l'équipe Tredzone. Slim, Adil et Loïc, merci à vous pour votre enseignement. C'est votre amitié, votre rigueur et votre curiosité qui font de moi l'ingénieur que je suis. Merci.

Mes années d'école d'ingénieur n'auraient pas été aussi agréables sans Julien, Mouhédine et Kévin. Merci les copains.

Je tiens également à remercier mes amis de longue date, Alban, Chris, Bruno, Tarana, Cédric, Ramy, Pif, Flore, Alex et Joe. Merci pour votre amitié depuis toutes ces années. Je ne serais pas la personne que je suis aujourd'hui sans vous Sans oublier Nicolas, mon copain depuis toujours, merci pour tout. Merci les copains.

Et enfin, je tiens à remercier ma famille, sans qui cette aventure n'aurait pas été possible. Papa, Maman, merci pour votre amour, votre soutien inconditionnel et l'éducation que vous m'avez apportée. C'est grâce à vous que j'ai pu étudier ce domaine qui me passionne et que j'ai mené à bout mon projet d'étude. Thibaud et Céline, merci d'avoir toujours été là pour moi et pour avoir toujours avoir su me mettre au défi pour toujours me dépasser. Mathilde, Maïa, ce travail est pour vous. Sans votre amour et votre soutien indéfectible, je ne serais pas l'homme que je suis aujourd'hui. Merci.

Merci à tous.

# Abstract

Today, stateful serverless functions are chained together through a message-based infrastructure and store their durable state in a separate database. This separation between storage and compute creates serious challenges that may lead to inconsistency and application crashes. A *unified consistency model* for message passing and shared memory is required to avoid such errors. The model should ensure that multiple pieces of data remain *mutually* consistent, whether data is sent using messages or shared in a distributed memory. Based on a well-known message-based model (actors) and a state model (transactional shared memory), we propose a unified communication and persistence model called Transactional-Turn Causal Consistency (TTCC). TTCC is asynchronous, preserves isolation, and ensures that the message and memory view are mutually causally consistent.

We propose an implementation of TTCC, based on a unified version vector. Our implementation ensures unified causal consistency for messages and shared memory with a response time overhead of up to 4.62×, 1.40× and 4.58× for read, update and message operation respectively.

**Keywords:** Causal Consistency, Actor Model, Message-Passing, Shared Memory, Serverless

# Résumé

Les applications serverless sont construites à l'aide de frameworks asynchrones basés sur des messages qui permettent aux utilisateurs de composer de manière abstraite des fonctions dans le cloud. Les fonctions serverless stockent leur état dans une base de données distribuée, telle que DynamoDB. Ce scénario architectural courant est fragile, car les garanties de cohérence des données pour la composition de la couche de messages et de la couche de base de données ne sont pas bien définies. Cela peut entraîner des incohérences, des pannes et des pertes de données. Cette séparation entre le stockage et le calcul crée de sérieux défis qui peuvent conduire à des incohérences et à des plantages d'applications. Les approches existantes sont ad hoc et ne garantissent pas la cohérence. En se basant sur un modèle bien connu basé sur les messages (acteurs) et un modèle à état (mémoire partagée transactionnelle), nous proposons un modèle de communication unifié, appelé Transactional Turn Causal Consistency (TTCC), ou Cohérence Causal par Tour. TTCC est asynchrone et préserve l'isolation en interne et garantit que les messages et la vue de la mémoire sont mutuellement cohérents.

Notre évaluation montre que TTCC est une solution viable pour du serverless à état. Notre mise en implémentation garantit une cohérence causale unifiée pour les messages et la mémoire partagée avec un surcoût de temps de réponse allant jusqu'à 4,62×, 1,40× et 4,58× pour la lecture, la mise à jour et l'opération de message respectivement.

**Mots-clés:** Cohérence Causal, Modèle Acteur, Message-Passing, Mémoire Partagée, Serverless

# Contents

# 1 Introduction

Serverless computing applications are built using asynchronous message-based frameworks, which enable users to abstractly compose functions in the cloud. Very often, to handle business logic, applications need to store state, such as user authentication, video encoding or collaborative workspaces. Today, stateful serverless computing stores state in a distributed database such as DynamoDB (Figure 1.1). This common architectural scenario is brittle because the data consistency guarantees for the composition of the message layer and of the database layer are not well-defined. This can result in inconsistency, crashes and data loss.

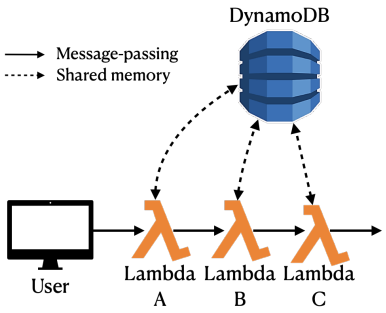



**Fig. 1.1.:** A stateful serverless construct.

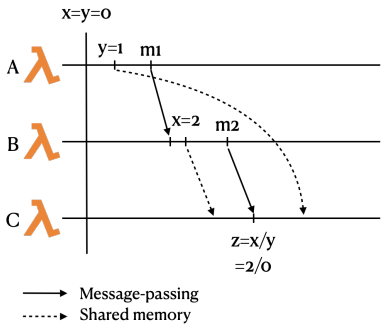



**Fig. 1.2.:** An inconsistency leading to a crash.

Let us illustrate this with the simple example in Figure 1.2, which is a timeline representation of Figure 1.1. Here, $x$ and $y$ are shared variables, initially set to 0, and replicated at all three sites. Node $A$ updates $y$ to 1 and notifies B with a message $m1$. Node $B$ updates $x$ to 2 and sends a message $m2$ to node $C$ requesting to compute $z = x/y$. In the absence of guarantees, $m2$ could be delivered before $y$ is replicated on node $C$, in which case $C$ computes $z = 2/0$, leading to a crash. The issue is that the unconstrained order of message delivery and of database updates violates causality and breaks a fundamental assumption that developers take for granted.

Consistency is the set of rules that constrain the order in which updates (e.g. sending a message or assigns a shared variable) become visible (to receivers or to readers). There are many consistency models, ranging from strict serializability (linearizability), where all updates are visible instantaneously to all processes, to the weakest

consistency models, where updates are delivered to remote nodes in no predictable order.

Most existing systems are designed to communicate either by message or by shared memory, but not both. Therefore, the literature defines message-based and data-based consistency models *separately*. However, serverless computing combines both paradigms. Maintaining *separate* consistency guarantees is non-intuitive and can lead to inconsistencies between message-delivery and shared-object guarantees (as in Figure 1.2).

A *unified consistency model* for message passing and shared memory is required to avoid such errors. The model should ensure that multiple pieces of data remain *mutually* consistent, whether data is sent using messages or shared in a distributed memory.

Serverless computing is asynchronous and thus incompatible with strong consistency (e.g. linearizability or serializability) as it imposes strong synchronicity requirements. On the other end of the spectrum, eventual consistency violates intuition, as our previous example shows. *Causal consistency* is a useful intermediate model, because it is asynchronous, and at the same time provides useful guarantees that ease reasoning [Akk+16; Zaw+15; Llo+11].

Isolation is a constraint that prevents a process from directly altering memory of another process. It is a useful property as it prevents deadlocks while maintaining asynchrony.

Our TTCC model is a transactional, causally consistent, memory-message model, that unifies message passing and shared memory in an asynchronous, and isolated environment.

## 1.1 Overview

In Part I of this thesis, we explore the serverless computing paradigm and the challenges that are created by its architecture. Then, we discuss the compatibility of two communication models: message-passing and shared memory. Finally, we present a comprehensive study of the state of the art, and what has been done in the literature to fulfill the serverless computing requirements.

In Part II, we present our main contribution, TTCC, a transactional, causally consistent model that unifies message-passing and shared memory in an asynchronous and isolated environment. We identify requirements for compatibility between actor-based stateful serverless framework, and shared state between actors. However, a unified memory model is challenging as independent consistency models are not necessarily mutually consistent. To answer this challenge, TTCC takes a hybrid approach, and provides the highest consistency guarantees compatible with asynchrony (TCC+) for shared memory and messages, and integrates them cleanly into a unified transactional memory model. In addition, CRDT data types ensure convergence without rollbacks. A related challenge is the overhead of concurrency metadata (vector clocks).

Finally, in Part III, this thesis addresses a number of design and implementation challenges, including user API choices, user constraints and concurrency metadata types.

## 1.2 Contributions

The main results of this dissertation are as follows:

- The formalization of a unified memory-message model, TTCC, which provides a unified, transactional and causally consistent guarantees that is compatible with stateful serverless frameworks.
- The design of a protocol, which leverages a version vector to maintain causal consistency in a hybrid memory-message environment.
- A reference design and implementation of TTCC, and its experimental evaluation.

Our implementation ensures unified causal consistency for messages and shared memory with a response time overhead of up to 4.62×, 1.40× and 4.58× for read, update and message operation respectively.

## 1.3 Publications

Some of the results presented in this thesis are pending publication:

- Benoît Martin, Laurent Prosperi, and Marc Shapiro. Transactional-Turn Causal Consistency. Euro-Par 2023 - 29th International European Conference on Parallel and Distributed Computing, Aug 2023, Cyprus.
- Benoît Martin and Marc Shapiro. Shared memory for the actor model. COMPAS 2022 - Conférence francophone d'informatique en Parallélisme, Architecture et Système [MS22].

During my thesis, I explored other directions and collaborated on the design of a high level approach to distributed systems' composition, which has helped me to get the insights on the challenges related to serverless frameworks. These efforts have led me to contribute to the following publications:

- Laurent Prosperi, Benoît Martin and Marc Shapiro. [Anonymized]. Submitted for publication.
- Benoît Martin, Laurent Prosperi, and Marc Shapiro. A new environment for composable and dependable distributed computing. EuroDW 2020 - 14th EuroSys Doctoral Workshop, Apr 2020, Heraklion / Virtual, Greece [MPS20].

## 1.4 Organization

This thesis is divided into three parts. The rest of this document is organized as follows:

- Part I introduces the background of our work, formulates the problem, presents the existing solutions, and discusses the use-case requirements. This part is divided into four chapters:
  - Chapter 1 is this introduction.
  - Chapter 2 reviews serverless computing frameworks. We identify the strengths and limits of existing frameworks. Furthermore, we define the consistency and isolation requirements needed to improve these frameworks.

  - Chapter 3 presents message passing as a communication model for the actor programming model. We conclude that isolation, asynchrony and causal delivery are requirements of the actor model.
  - Chapter 4 presents the isolation and causal delivery in a distributed system that communicates by shared memory. We define the requirements that are needed to guarantee clean integration into the actor model.

- Part II, we specify the design and implementation of a unified model that ensures mutual consistency between the message and the state view.
  - Chapter 5 presents our unified model. We formally describe the unification of causal consistency for message-passing and shared memory. Then, we present isolation and a transactional turn.
  - In Chapter 6, we explore the design space of TTCC. We present three protocols that unify message-passing and shared memory using different consistency metadata.
  - Chapter 7 presents our transaction API and our reference implementation using the Akka actor framework.
- Part III provides an experimental evaluation.
- Finally, part IV summarizes our contribution, and present our vision for the future requirements towards more reliable, scalable and easy to use stateful serverless computing frameworks.

# Part I

Background

# 2 Serverless Computing

## 2.1 History of Cloud Computing

The concept of cloud computing has its roots in the 1960s, when computer scientists first began exploring the idea of providing computing resources as a utility, much like electricity or water. However, it wasn't until the late 1990s and early 2000s that the technological advancements and widespread adoption of the Internet made cloud computing a practical reality.

In the early days of cloud computing, companies such as Amazon and Google began offering simple web-based services, such as online storage and email, to consumers. Over time, these services expanded to include more complex offerings, such as virtual machines and databases, that could be rented and accessed over the Internet.

In 2006, Amazon Web Services (AWS) [Ama] was officially launched, offering a suite of cloud-based services to businesses. This marked the beginning of the widespread adoption of cloud computing and the birth of the public cloud computing industry. Over the next decade, major technology companies, such as Microsoft and Google, entered the market with their own cloud computing offerings, and the industry continued to evolve and expand.

More recently, in 2014, AWS introduced Amazon Lambda [Awsa], the first publicly available serverless computing platforms, which allowed developers to run code without having to manage servers. Since then, serverless computing has gained popularity and several other cloud providers have introduced similar services, including Microsoft Azure Functions [Azuc] and Google Cloud Functions [Goo].

In recent years, the growth of serverless computing is driven by the increasing demand for flexible and cost-effective solutions to build and run applications. With the rise of microservices and the need for scalable, event-driven architectures, serverless computing has emerged as a key enabler for modern application development.

Today, cloud computing is a multi-billion dollar industry, serving the computing needs of businesses, governments, and individuals around the world. With its

many benefits, which includes cost savings, scalability, and ease of use, serverless computing has become an essential part of the modern cloud computing landscape.

## 2.2 Concepts

Serverless computing is a cloud computing model where the cloud provider manages the infrastructure and automatically allocates resources as needed to execute and scale an application's code. Customers simply submit their source-code to the serverless provider, who then builds, deploys and runs (i.e. instantiates) the application. With serverless computing, the focus is on writing and deploying code, rather than managing infrastructure, leading to lower costs, increased agility, and improved scalability [HBS21].

Serverless functions, also known as functions-as-a-service (FaaS), are the key concept behind serverless architectures that allow developers to run code without managing the underlying infrastructure. When an event triggers the function, it runs in a containerized environment that is automatically managed by the cloud provider.

Serverless computing offers several advantages over traditional server-based approaches to building and running applications:

**Cost savings** In a serverless model, resources such as computing power and memory are provided on-demand. The user is charged, only when the code is executed, which leads to lower costs and better compute utilization compared to traditional server-based approaches.

**Increased agility** Serverless computing allows customers to focus on writing and deploying code, rather than managing infrastructure, which results in faster time to market for new applications and features.

**Improved scalability** In a serverless architecture, the cloud provider automatically allocates resources as needed to execute and scale an application's code, which leads to improved scalability.

**Event-driven processing** Serverless computing is well-suited for event-driven processing, where code is executed in response to specific events or triggers, making it a good choice for building scalable applications.

Overall, serverless computing offers businesses and developers a flexible, cost-effective, and scalable way to build and run applications and services in the cloud. However, serverless computing also suffers from some limitations:

**Cold start** The initial execution of an application's code can take a long time due to the need to spin up a new instance of the function. This "cold start" issue, results in increased response time, which can be a challenge for applications with strict response time requirements.

**Limited control over infrastructure** In a serverless environment, the cloud provider manages the infrastructure, which limits the control that developers have over the underlying environment. This can make it more difficult to customize the environment to meet the specific needs of an application. For instance, customers cannot choose hardware, which may be limiting for an application that requires a specific CPU or network topology.

**Resource constraints** Serverless computing platforms typically impose limits on the resources that can be used, such as the amount of memory or the run time of a function. These constraints can impact the performance and scalability of an application.

**Cost management** While serverless computing can lead to cost savings, it can also lead to unexpected costs if an application's usage patterns change or if there are spikes in resource usage. Careful cost management and monitoring are important in a serverless environment.

**Vendor lock-in** Users have to choose which cloud provider they want to enroll with, depending on the features and services that the provider has. Most cloud providers use their own in-house serverless framework (and sometimes more than one), which exposes an API that may not be compatible from one cloud provider to another.

## 2.3 Serverless frameworks

Serverless computing functions are triggered by events. An event is an asynchronous input to a function. An event is created when a database updates, with an incoming API request, is scheduled or when serverless functions are chained together. A serverless function is the application logic that a customer wants to deploy to a cloud provider. A serverless computing frameworks is a software platform that helps a cloud provider to build and deploy a user's serverless functions. When a function is triggered by an event, the serverless platform automatically creates an instance of that function to handle the event. The instance is then destroyed when the function has finished executing and outputs a response. An invocation is the act of triggering

a function to run in response to an event. When a function is invoked, the serverless platform automatically instantiates the function, and the function code is executed within the context of that instance.

Serverless computing frameworks can be categorized into two family types, which reflect two different approaches to building serverless applications: stateless and stateful. *Stateless* serverless frameworks are designed to support functions that don't maintain any local state that outlasts a single invocation. Take, for example, a serverless function that computes the sum of numbers that are given as parameters. This function does not store state and only depends on its input parameters.

On the other hand, *stateful* serverless frameworks are designed to maintain local function state and share variables between invocations. State and shared variables may be persisted. Stateful serverless frameworks provide a variety of data storage options, including databases and object storage.

Take for instance Netflix that leverages serverless computing for video encoding [Net]. When a video file is placed in an object store, a serverless function is triggered to split the video into 5 minutes parts that are encoded into 60 different streams. Later, the final video files are re-assembled and deployed into Netflix's content delivery network.

A second example is serverless machine learning applications require a global shared state that is accessible by all spawned instances. The global state is synchronized so that the algorithm can correctly proceed to the next iteration [BP+22].

### 2.3.1 Stateless

Stateless serverless computing refers to the concept of using serverless functions that only relies on its input (Figure 2.1). Take for example a serverless function the converts a text document into a PDF file. The input to this function is the text document and the output is a PDF document containing the input text.

Stateless function scale with ease: converting 100 files to PDF can easily be done by invoking 100 functions in parallel, as converting 1 file does not interfere with the result of another conversions. (Figure 2.2).

Furthermore, a stateless function is idempotent. In other words, no matter how many times the function is run with the same input, it will always produce the same output. This allows for the function processing to retry in case of an error, without risking the corruption of the state (because there is no state).

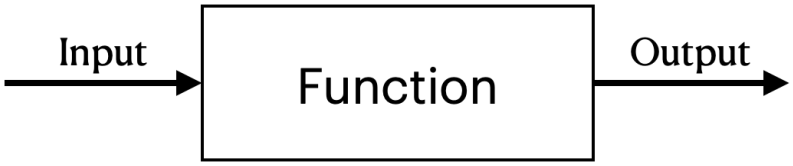



**Fig. 2.1.:** A stateless function processes an input and generates an output.

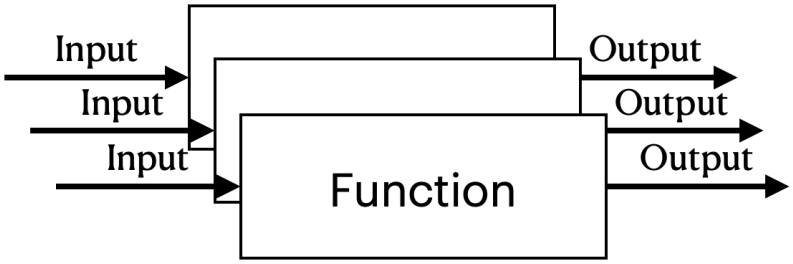



**Fig. 2.2.:** A stateless function can be invoked multiple times for parallel processing.

Overall, stateless functions allow for a cloud application to be elastic (scale up and down) and resilient (retry in case of a failure).

However, even though stateless serverless computing is appealing for certain applications, it remains limited. Stateless serverless functions do not maintain any state between invocations, so they cannot store data for later use. This makes it difficult to build complex applications, as developers need to implement their own state management, which can be time-consuming and error-prone. For example, a website that counts the number of clicks on an item must store what the previous value of the counter is, which is not possible using a stateless function.

### 2.3.2 Stateful

A stateful serverless framework maintains persistent state between invocations, which a function can access or update. Execution of a stateful function relies on external data (i.e. anything that is not contained within the data that is being processed). Usually, a stateful function relies on an external service, such as a database, to store its state (Figure 2.3).

Burckhardt et al. [Bur+21] identifies two types of stateful serverless computing frameworks: workflow-based and actor-based.

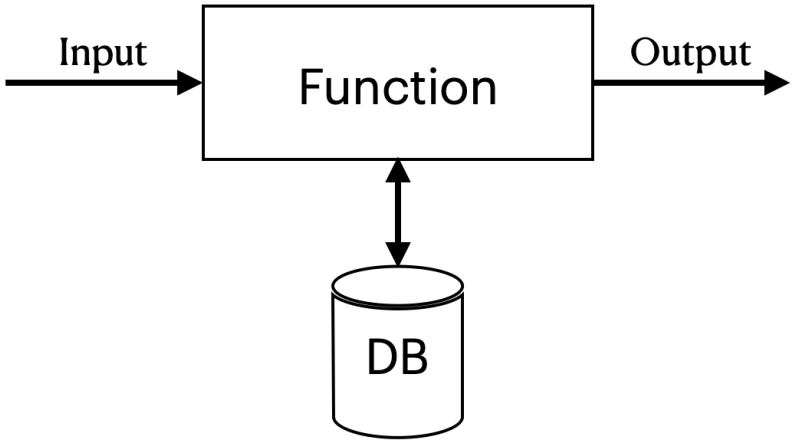



**Fig. 2.3.:** A stateful function that stores its state in an external database (DB).

### Workflow-based frameworks

A serverless workflow-based framework refers to a specific type of serverless architecture, where functions are managed as a series of *steps* that are processed in a specific order, typically following a predefined workflow. A workflow describes the high-level logic of the application.

Workflow-based serverless frameworks provide support for building complex, multistep workflows, allowing developers to model complex business processes and data processing pipelines. Workflow-based frameworks typically provide a graphical interface for designing workflows, as well as the ability to track the progress of workflows and ensure that all steps are executed correctly.

Workflow-based frameworks use a message-queue service to compose serverless functions together. An orchestrator ensure that the right function is executed at the right time, and that the output from one function is passed as the input to the next function in the workflow, as shown in Figure 2.4.

Overall, cloud providers do not offer many configuration options over the queuing and orchestration services, which is limiting as messages cannot be grouped (e.g. process in batch) and timing is imposed. This loss of control forces developers to manually orchestrate their workflow, which is difficult and prone to error. Take for instance the following example [Sta]: a customer wants to start a workflow, but only after having received 10 notifications from an external system. Using workflows, there is no good way of modeling this, as there is no control over the state of the queue. The developer has to manually store the state of the queue in a third-party data-store, such as a database, and trigger the next function when the specified

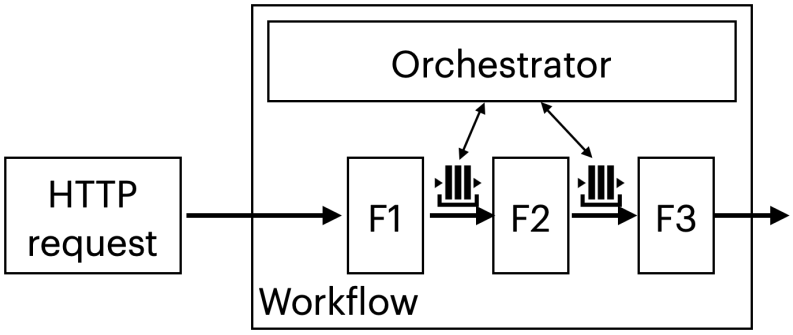



**Fig. 2.4.:** Workflow-based framework overview. Function F1 is triggered by an HTTP request and chained, using a queuing service, to function F2. F2 is chained to F3.

counter value is reached. Moreover, how is the state of the queue synchronized, and where is it stored?

Usually, to store state, workflows use a third-party data-store, such as a database or an object store. However, this separation between message queues and state storage is prone to data inconsistencies, as shown in Figure 1.2.

**AWS Step Functions** Step Functions [Awsc] provides a graphical interface for designing and visualizing workflows, as well as an execution environment for running those workflows. With Step Functions, developers can model complex business processes, such as order processing, data processing pipelines, and long-running transactions, as a series of *steps*. Each step can be a single AWS Lambda function, an activity worker, or a task that calls another AWS service, such as Amazon Simple Notification Service (SNS) or Amazon DynamoDB. Step Functions automatically tracks the progress of workflows and ensures that each step is executed correctly.

Step Functions stores state in an external database, such as DynamoDB or Amazon Simple Storage Service (S3), and messages between functions use a queue, such as Amazon Simple Queue Service (SQS) [Eva].

Step Functions offer limited capabilities to synchronize functions [GL+18; Jon+19]. The framework does not provide a signal for functions to coordinate, which is useful to guarantee the order of events, or to ensure joint progress to the next stage of computation. Furthermore, *separation* between state storage and function message queuing, is prone to data inconsistencies.

**Azure Durable Functions** With Durable Functions [Azub], developers can model complex workflows as a series of functions that can run in parallel, call other Azure services, such as Azure Storage and CosmosDB, and be orchestrated into a single workflow. When an instance of a Durable Function is triggered, the orchestrator creates a new execution context and stores it in Azure Storage. This execution context includes the current state of the function instance, as well as metadata about the function's history and any pending actions. As the function executes, the orchestrator updates the execution context and uses it to make decisions about what actions to take next. Once the function has completed, the execution context is marked as complete in Azure Storage and any output values are returned to the caller.

Azure's Durable Functions programming model enhances functions with critical sections (i.e., region of the application where only one orchestrator is allowed to call a specific function). This allows the orchestrator to read or modify multiple function state atomically (modifications are visible instantly), which is useful for an application that require strong consistency guarantees, such as a money transfer between bank accounts.

However, Durable Functions use a *separate* message queue service to communicate between each other and a *separate* data-store to persist state, which is prone to inconsistencies. Furthermore, critical sections allow strong guarantees between concurrent functions, but at the expanse of parallelism.

**IBM Composer** With IBM Composer [Ibm], developers can build workflows using a visual interface, and they can connect to a variety of IBM Cloud services and APIs. Function state is managed by storing data in the context object of each function, which is persisted in a *separate* data-store. The context object is passed from one function to another as the workflow progresses, allowing each function to access and update the state as needed.

IBM Cloud Functions are composed together using an external queue service and state is stored in a database.

*Separation* between state storage and inter-function communication, is prone to data inconsistencies.

**Cloudburst** Cloudburst [Sre+20] is an academic serverless platform that is built on top of Anna [Wu+18], an auto-scaling key-value store, tailored for serverless computing. With Cloudburst, developers can deploy stateful serverless functions

with efficient and low-latency shared state. Furthermore, direct communication between functions is possible.

In Cloudburst, each function instance is assigned a unique identifier. These identifiers are translated to physical addresses (i.e. IP and port) to support direct messaging. Shared objects are stored in Anna and support causal consistency. Furthermore, with Cloudburst, direct communication between functions is possible. When a message is sent in a serverless function, Cloudburst establishes a TCP connection to directly send the message. If a TCP connection cannot be established, the message is written to a key in Anna that serves as the receiving thread's inbox. The receiving function explicitly calls a *recv* function to retrieve messages.

Cloudburst separates direct messaging from shared state, which is prone to data inconsistencies.

### Actor-based frameworks

Actor-based serverless frameworks are a type of stateful serverless computing platform that are built on top of an actor programming model.

The actor model is a message-passing model that was originally introduced by Hewitt et al. [HBS73] and later revised by Agha [Agh85]. Frameworks such as Orleans, Cloudflare Durable Objects, Lightbend Akka Serverless or Azure Durable Entities allow application state to be used in actors.These frameworks, however, do not offer a unified consistent view of messages and shared objects.

The actor model is an asynchronous (i.e. without blocking) message-passing model. An actor is an entity that runs with others concurrently. It responds to a message it receives by making a local decision, creating other actors, and sending messages to other actors. An actor may modify its own private state, but can only affect another indirectly by sending a message.

An actor consists of three elements: an address, a mailbox and a behavior. An actor has a unique, immutable *address* that is used to send it messages. Its *inbox* is usually a First-In-First-Out (FIFO) queue of messages. Finally, its *behavior* (i.e. its code) specifies its response to a message.

Actors can be used to represent individual serverless functions or groups of related serverless functions that share a common state, as shown in Figure 2.5. Functions that are grouped in a single actor shared a local state and are run sequentially. This provides fine-grain control, that is integrated into the programming model, which

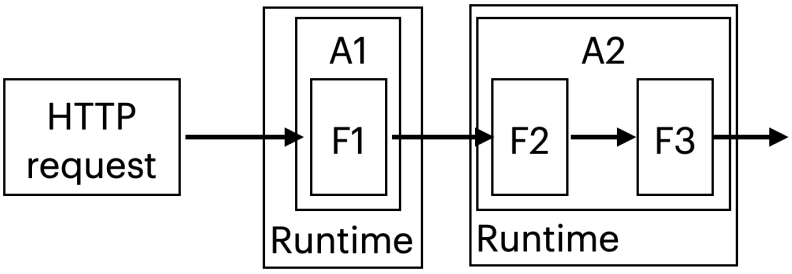



**Fig. 2.5.:** Actor-based framework overview. Function F1 is instantiated in actor A1 and is triggered by an HTTP request. Functions F2 and F3 are instantiated in actor A2 and share the actor's local state. Actors communicate using direct messages.

provides shared variables that enable developers to write complex applications. For instance, waiting for 10 documents to be available is done by using a counter in an actor. The function is triggered when a document is available and the counter is incremented. When the counter reaches 10, the next function is triggered.

One of the key advantages of the actor model is its ability to handle concurrency and parallelism. Actors operate independently of each other, multiple actors can perform their actions simultaneously without interfering with each other.

The actor model has a long history [DKVCDM16], and is widely used in many programming languages, such as Erlang, Scala, SALSA, E, AmbientTalk and in frameworks, such as Lightbend's Akka, Microsoft Orleans or Apache OpenWhisk.

**Akka Serverless** Akka Serverless [Akkb] is a serverless computing platform that is built on top of the Lightbend Akka actor framework [Akka]. It provides a scalable and fault-tolerant environment for running serverless applications, and it supports Java and Scala programming languages. Akka Serverless also provides a rich set of libraries and tools for integrating with other cloud services and data sources, making it easy to build and run serverless applications that are fully integrated with existing systems.

In Akka Serverless, a function is invoked inside an actor. A function's state is coupled to the state of the actor on which the function is run. Akka Serverless uses Event-Sourcing to restore an actor's state. Event-sourcing consists in saving messages that are sent to an actor, in order to replay those messages, in the order in which they were sent, to restore the state of the actor. Additionally, snapshots may be used to limit the number of saved messages.

Actors can share a global state *without* using a third-party database. An integrated key-value store (KVS) is available for actors to shared mutable objects. Also, actors

communicate using asynchronous messages, which eliminates the need for an external queuing service. Global shared state and messaging is native to Akka serverless.

Akka serverless does not provide atomic modification of globally shared objects, which limits the development of certain types of applications, such as banking operations. Furthermore, Akka serverless guarantees at-most-once message delivery and FIFO message ordering per sender–receiver pair. In other words, a message sent between two actors is FIFO *but* is not guaranteed to be delivered, as it may be lost in transit.

**Azure Durable Entities** Azure Durable Entities [Azua] is a feature of Azure Durable Functions, a workflow-based serverless computing platform provided by Microsoft Azure. Durable Entities provide a programming model that allows developers to define entities, which are objects that have a state and behavior. Durable Entities support a wide range of programming languages, including C#, Java, JavaScript, and Python, and they are implemented as functions that are executed in the Azure Functions environment. Durable Entities are designed to be highly scalable, reliable, and efficient, and they provide automatic and transparent state management, with the state of each entity being stored in Azure storage.

Internally, entities encapsulate local durable state that are stored using a replay-based model, which consists in persisting and restoring intermediate state by recording and replaying events in a history [Bur+21]. Additionally, like actors, entities store events in a queue (i.e. mailbox) and execute them one at a time.

Azure Durable Entities guarantees that messages sent to the same entity are delivered in the order in which they are sent.

**Cloudflare Durable Objects** Cloudflare Durable Objects [Clo] are designed to be highly scalable, reliable, and efficient, and they provide automatic and transparent state management, with the state of each object being stored in Cloudflare's global network of data centers.

Similarly to actors, a Durable Object is instantiated in a runtime, called a Cloudflare Worker. Each Durable Object has access to its own isolated storage and is automatically replicated and fail-over in case of failures. A Durable Object is global unique and uses a transactional key-value store to share objects atomically between durable objects.

Cloudflare Durable Objects does not guarantee transitive message delivery of messages. Furthermore, Cloudflare KV is eventually consistent [How], which means that shared objects will eventually be available to other Durable Objects.

Actor-based serverless computing frameworks provide a way to build scalable, concurrent, and distributed systems that are easy to reason about and manage, making them well-suited for a wide range of use cases. However, the serverless framework that we study lack consistency guarantees and use distinct storage for actor state and message queuing, which can result in data inconsistencies.

### 2.3.3 Example of a stateful application that has consistency issues

Stateful serverless computing frameworks, whether workflow-based or actor-based, store function state in a third-party data store. These frameworks rely on the guarantees provided by the underlying data-store and in the framework's asynchronous messaging mechanisms. This ad-hoc way of handling data can lead to potential inconsistencies that can cause application crashes and service outage, as illustrated in Figure 1.2.

As a more specific example, consider the file download and zip application (Figure 2.6), built by Warden [War]. This design, known as *claim-check pattern* [HW13], is commonly used in serverless services, to avoid message size limitations. Unfortunately, this simple cloud application does work as intended. A user may request one hundred files to be downloaded and only retrieve ninety-seven files in the resulting zip file. Given as an argument by the user, a list of files are packaged into a single compressed zip file. The resulting file is visible on an online web interface and sent by email to the user. This use-case chains together AWS Step Functions using asynchronous messages. The first Step Function spawns an AWS Lambda function (stateless) per file to download. The downloaded files are stored in an AWS S3 Bucket (following the recommendation by AWS for large files because of the 32 KB message size limit [Awsb]). A reference to the S3 bucket is sent to the second Step Function, which reads the bucket and compresses the files it contains into a single zip file.

Unfortunately, this application does not work as intended. A user might request one hundred files to be downloaded and retrieve only ninety-seven files in the resulting zip file. This anomalous behavior is the result of eventual object replication in AWS S3 buckets and eventual message delivery between AWS Step Functions. The second Step Function might read from a S3 bucket that is not yet up-to-date. The

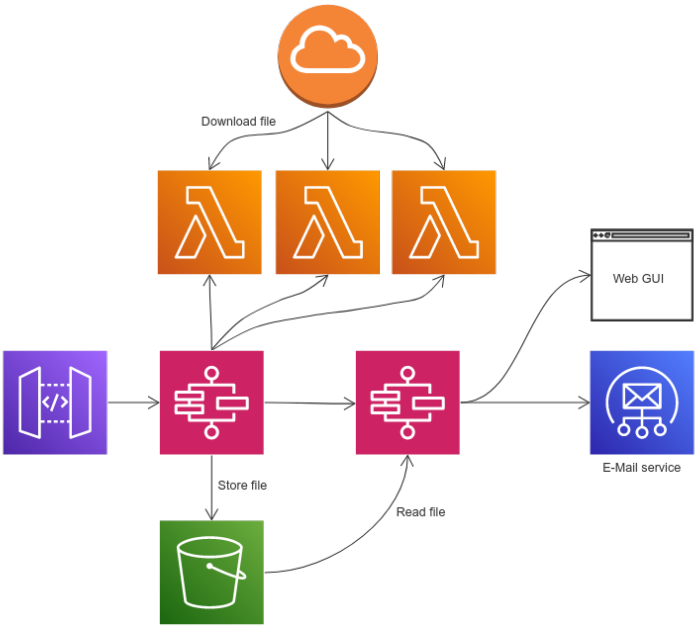



**Fig. 2.6.:** The file download and zip serverless service deployed on AWS using Step Functions, Lambda and an S3 bucket. This service handles the download, compression and ZIP of a list of files given as an input by a user. Large objects whose reference is sent by message.

solution reported by Warden [War], is to *manually* delay the compression task in the second Step Function, by computing a MD5 hash of the files, which essentially results in performing manual data consistency. If the file hashes match, then all files are available and the compressing task can complete, otherwise at least one file is missing, and the Step Function cannot proceed.

This common issue is due to the lack of guarantees between multiple data contexts, which forces users to manually ensure that messages and shared objects remain consistent. Unified consistency guarantees should be provided by the serverless framework.

## 2.4 Summary

Today, serverless computing is a popular abstraction for cloud computing. By using functions-as-a-service, users no longer need to worry about deployment strategies or infrastructure management. This is one major step in simplifying cloud computing, but two steps back, as serverless frameworks handle state in an ad-hoc and non-integrated way, which can lead to inconsistencies.

Stateful serverless frameworks should include guarantees to prevent data consistency issues. However, frameworks used today do not offer these guarantees, which forces users to *manually* check for inconsistencies, which is cumbersome and error-prone.

This thesis focuses on *stateful* serverless frameworks, as they are prone to inconsistencies due to separate data consistency contexts of their data store and their messaging layer. Moreover, we focus on *actor-based* frameworks, as they offer better development possibilities compared to a workflow-based framework. Table 2.1 summarizes the current state of the art concerning serverless frameworks.

| | | | Guarantees | |
|---|---|---|---|---|
| Framework | Stateful | Type | Message | Memory |
| Google Cloud Functions | ✗ | Unspecified | N/A | N/A |
| Azure Functions | ✗ | Unspecified | N/A | N/A |
| AWS Lambda | ✗ | Unspecified | N/A | N/A |
| Apache OpenWhisk | ✗ | Actor | N/A | N/A |
| AWS Step Functions | ✓ | Workflow | Eventual | Store dependent |
| Azure Durable Functions | ✓ | Workflow | Eventual | Store dependent |
| IBM Compose | ✓ | Workflow | Eventual | Store dependent |
| Cloudburst | ✓ | Workflow | Eventual | Causal |
| Akka Serverless | ✓ | Actor | Eventual | Store dependent |
| Azure Durable Entities | ✓ | Actor | Eventual | Store dependent |
| Cloudflare Durable Objects | ✓ | Actor | Eventual | Store dependent |
| TTCC | ✓ | Actor | Causal | Causal |

**Tab. 2.1.:** Table of serverless frameworks.

# 3 Consistency in Message-Passing Systems

In a message-passing system, such as the actor model, *message ordering* is the set of rules that constrain the order in which messages are visible. Actors do not offer guarantees for message delivery. We identify that a stronger message ordering guarantee is necessary to ease reasoning and avoid certain transitive message ordering issues. Furthermore, to help developers reason about their application, our target message ordering model should be asynchronous, must not violate causality (Figure 1.2, message $m2$ will be delivered after the update to $y$ is replicated to the bottom node), and ensure isolation between functions to cleanly integrate into an actor-based serverless framework.

In this chapter, we introduce a formal definition of causal message ordering for actors. Then, we explain the concept of actor isolation and why this property is crucial for the actor model.

## 3.1 Abstract Model

Borrowing from Burckhardt [Bur14] and Viotti and Vukolić [VV16], we model a system execution using a multi-graph $A = (\mathcal{E}, vis)$ built on a set $\mathcal{E}$ of *events*.[1] Events comprise send, receive, read and write operations.

More specifically:

**Program-order** $\xrightarrow{\text{PO}}$ is a binary relation over $\mathcal{E}$ that expresses the natural execution order of operations by a process.

**Visibility** $vis$ is a binary relation over $\mathcal{E}$ that describes the propagation of information through the system. It satisfies the following rules:

1. $\xrightarrow{vis}$ is acyclic.
2. It is transitive: $\forall e, f, g \in \mathcal{E} : e \xrightarrow{vis} f \wedge f \xrightarrow{vis} g \implies e \xrightarrow{vis} g$
3. Program order implies visibility: $\xrightarrow{PO} \subseteq \xrightarrow{vis}$

[1] Burckhardt also defines a total arbitration order, but it is not necessary for our purpose.

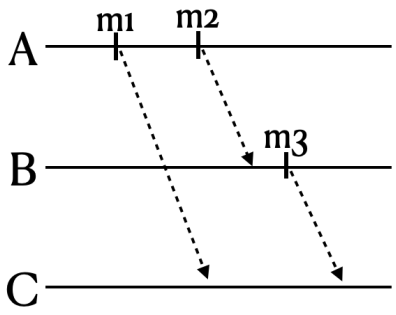



**Fig. 3.1.:** Example of causal message delivery. Actor C receives message $m1$ before message $m2$.

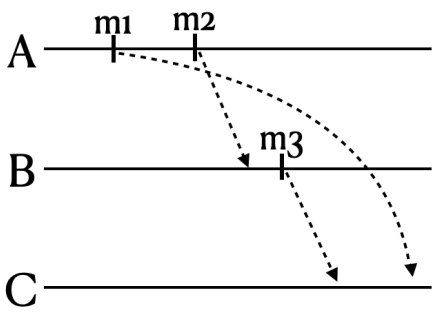



**Fig. 3.2.:** Example of non-causal message delivery. Actor C receives message $m2$ before message $m1$.

For instance, $a$ is visible to $b$ (i.e., $a \xrightarrow{\text{vis}} b$) means that the effects of $a$ are visible to the process invoking $b$. Two operations are said *concurrent* if they are not ordered by $vis$.

## 3.2 Causal Message Ordering

Causal message ordering is based on Lamport's happens-before relation [Lam78], which captures the potential causal relationships between events of multiple processes (which can be actors). This guarantees that all causally-related operations are visible, in a common order, and without gaps (i.e: all causally related dependencies of an event are satisfied).

Causal message ordering is a useful model because it is asynchronous and provides useful guarantees that ease the programmers' reasoning. In causal message ordering, the relationship between causally related messages is preserved. This means that if message $m1$ causally precedes message $m2$, the receiving actor will receive $m1$ before $m2$, as illustrated on Figure 3.1. Figure 3.2 shows the non-causal delivery of message $m3$, as $m1$, which is a causal dependency, is not yet received. If two messages are not causally related, their order can be observed differently. Causal message ordering allows for higher availability than strong message ordering, as it does not require synchronization, even under network partition.

We note messages $m, n$ (messages are assumed unique); message-related events are send and receive, noted $send(m)$ and $recv(m)$ respectively. A message is *causally delivered* if and only if it satisfies the common rules (1)–(3), as well as the following:

④ A received message must be sent: $rcv(m) \in \mathcal{E} \implies send(m) \in \mathcal{E}$

⑤ A send precedes the corresponding receive: $send(m) \xrightarrow{vis} rcv(m)$

⑥ A message does not overtake another message:

$$send(m) \xrightarrow{vis} send(n) \implies \neg(rcv(n) \xrightarrow{PO} rcv(m))$$

**Receiving $m$ implies that $m$ was sent** Rule ⑤ states that $m$ is visible when it is received, which is after it was sent.

**Message visibility order** Rule ⑥ defines the order in which messages $m$ and $n$ are made visible (delivered). If an actor sends $m$, and later an actor sends $n$, a destination actor must observe $m$ before $n$. We use negation ($\neg$) because a destination might receive only one of the messages.

## 3.3 Isolation

We define isolation as a constraint that prevents an actor from directly altering memory of another actor. This is a useful property as it prevents data-based deadlocks (i.e., a message-based deadlock is still possible).

An actor alternates between two states: ready to accept a message, or busy processing a message. A *turn* is the processing of a single message by an actor until completion [DKVCDM16]. An actor executes a single turn at a time, processing a single message per turn; it runs the turn to completion without interruptions or blocking, before waiting for the next message.

The actor model is based on the Isolated Turn Principle [DKVCDM16], which states that once a turn has started, it will always run to completion without sharing state, and without blocking. Thus, the actor is isolated and the processing of a turn is free from deadlocks. The programmer can reason about the application as a sequence of isolated, functional turns.

This intuition is captured by the Isolated Turn Principle as a combination of three guarantees:

- **Continuous message processing**: An actor's turn terminates without interruption.

- **Consecutive message processing**: An actor processes messages from its own inbox, and processes them one by one. Within one actor, turns do not interleave.
- **Isolation**: An actor can only access its own memory. Actor systems usually achieve this by disallowing shared mutable state between actors. Hence, turns are free from low-level data races.

The Isolated Turn Principle is important because it helps to ensure correctness despite concurrent execution. By enforcing a strict order of message processing and isolation between actors, the actor model ensures that there are no race conditions, or other synchronization problems, which can otherwise arise when multiple entities attempt to access or modify shared state simultaneously.

## 3.4 Summary

Actors provide a natural abstraction that provides a simple and intuitive way to represent message-driven systems as a collection of independent actors. Each actor can represent a single unit of functionality (i.e. a serverless function) that is responsible for processing a specific message, and actors can communicate with each other asynchronously through message passing. Furthermore, multiple functions that are instantiated in the same actor, share the actor's local state, which is a useful to store state between serverless function invocations.

Causal message ordering is a useful model as it guarantees that if one message causes another message, messages to the receiving actor will be delivered with respect to that causal relationship. Causal message ordering is important as it dictates the order in which causally related messages are visible, and due to its asynchronous nature, is compatible with serverless computing.

The *Isolated Turn Principle* governs the way actors interact with each other. An actor processes a single message at a time, without interference from other actors. When an actor receives a message, it processes that message to completion before processing any other messages. This ensures that each actor is isolated from other actors during message processing, and that the behavior of the system is predictable and consistent, as an actor cannot directly change the state of another actor. Actor isolation is important because it helps to ensure the correctness of concurrent systems. By enforcing a strict order of message processing, the actor model ensures that there are no race conditions or other synchronization problems that can arise when multiple entities attempt to access or modify shared state simultaneously.

# 4 Consistency for Shared Memory

Shared memory comes in many forms: physical memory shared between threads and processes, a distributed file system shared between nodes or a database. Consistency refers to the behavior of shared memory across multiple nodes or processes that are geographically dispersed. The goal is to ensure that all nodes in the system have a consistent view of the shared memory. Achieving consistency in distributed systems is challenging because of network latency and communication delays that can lead to differences in the order in which objects replicate across nodes, even if they start from the same initial state. This can result in conflicts and inconsistencies in the shared memory.

There are many consistency models, ranging from strict serializability or linearizability, where updates become visible instantaneously to all processes, to the weakest consistency models, where updates are delivered to remote nodes in no predictable order [SS18]. Each model offers different trade-offs between performance, fault-tolerance, and consistency guarantees [Bre00; FLP85].

## 4.1 Causal Consistency

We borrow our shared-memory execution model from Cerone et al. [CBG15]. They consider a database consisting of *objects* $Obj = \{x, y, \dots\}$. Events consist of $wr(x, v)$, writing version $v$ to object $x$, and $rd(x, v)$, reading $v$ from $x$; a write associates a new, unique version to the object being updated.

An execution is *causally consistent for shared memory* if and only if it satisfies the common rules (1)–(3), as well as the following:

(7) A version read must be written: $rd(x, v) \in \mathcal{E} \implies wr(x, v) \in \mathcal{E}$

(8) A write precedes the corresponding read: $wr(x, v) \xrightarrow{vis} rd(x, v)$

(9) An update does not overtake another update:

$$wr(x, v_1) \xrightarrow{vis} wr(x, v_2) \xrightarrow{vis} wr(y, w) \implies \neg(rd(y, w) \xrightarrow{PO} rd(x, v_1))$$

**Reading an object implies that the object was written** Rule (8) states that an update to object $x$ with version $v$, is visible before reading $x$.

**Object reads the latest version** Rule (9) states that once an update, tagged with version $v_2$, is visible, then no subsequent operation can see a version prior to $v_2$. In other words, only the latest version of an object is visible.

Note that Burckhardt [Bur14] defines $\mathcal{F}_\mathcal{T}$ as a generic return type for operations in $H$. We do not need to introduce $\mathcal{F}_\mathcal{T}$ in our model, since we only require one specific read operation that only depends on a version.

## 4.2 Isolation

To maintain isolation and asynchrony, database systems employ multiversion concurrency control (MVCC), which uses snapshots and transactions [BG83]. MVCC works by creating multiple versions of each data item, and assigning a unique timestamp to each version, which allows an actor to read and write data without waiting for a lock or blocking other processes. When a transaction reads or writes data, it only sees the version of the data that is valid at the time the transaction started, which ensures that different transactions do not interfere with each other. MVCC is widely used in modern database systems to improve concurrency and performance, and it is particularly useful in environments with high transaction rates and high concurrency.

### 4.2.1 Transactions

A transaction is an isolated unit of work. To achieve isolation, it reads from a snapshot and makes all its writes visible at once (if it commits) or discards them all (if it aborts).

Transactions guarantee atomicity, which refers to the property that ensures that all the operations in the transaction are treated as a single, indivisible unit of work. In other words, if any part of a transaction fails, the entire transaction is rolled back to its previous state, as if the transaction had never been executed. This is known as the "all-or-nothing" principle [SS18], and it ensures that the data remains consistent, even in the presence of concurrent access and updates by multiple users or applications and in the presence of failures or errors.

Formally, we define atomicity for transaction $T$ as: $\forall g, e \in T \land f \in T \implies e \xrightarrow{\text{vis}} g \Leftrightarrow f \xrightarrow{\text{vis}} g$. Either all of $T$'s effects are visible, if the transaction is committed, or none are, if the transaction aborts or hasn't terminated yet. This is known as the "all-or-nothing" principle [SS18], and it ensures that the data remains consistent, even in the presence of concurrent access and updates by multiple processes and in the presence of failures or errors.

### 4.2.2 Snapshots

A snapshot is the state of the system at a given point in time. Each concurrent transaction reason from its own snapshot. Any read of a transaction comes from the transaction's own writes, if any, or otherwise from its snapshot, which includes all the transactions that precede it. Snapshots ensure that processes remain mutually isolated, and improve concurrency of reads and updates, as each transaction can run in parallel without affecting others. For objects $x$ and $y$, the predecessor set of a transaction $T$ is $pred_T(x) = \{y \mid y \xrightarrow{\text{vis}} x \land y \notin T\}$. The snapshot property can be defined as follows for transaction $T$: $x \in T \land y \in T \implies pred_T(x) = pred_T(y)$.

Writing in a transaction is staged to a buffer local to that transaction. These writes become visible atomically when the transaction commits.

## 4.3 Conflict-free Programming

Traditional approaches to distributed system design are not generalizable to serverless computing because they depend on strong synchronization between multiple parties, such as that provided by uniform consensus. Unfortunately, such strong synchronization primitives are impossible to realize in a scalable manner on serverless architectures. An alternative to using such primitives is to rely on conflict-free programming [Sha+11].

Consider a distributed data structure, replicated across $n$ nodes to improve robustness to node failures. Each node has a copy of the current value of the data structure. When an operation is invoked on a node, a new value is calculated and all nodes must be updated with this new value. This operation must be performed consistently in the event of concurrent operations, node failures, and network disruptions ranging from variable latency to message drop or partitions.

Relying on strong synchronization primitives, one would use a solution based on a uniform consensus algorithm such as Multi-Paxos [CGR07] or Raft [OO14]. Many industrial systems use this solution, for example, Google's Chubby lock service uses Multi-Paxos. In this solution, the consensus algorithm is run on all $n$ nodes for each operation, which, in addition to being expensive and not scalable, does not provide availability in the event of a major node failure or network disruption.

The alternative, proposed in the context of conflict-free programming, is based on the observation that, in most cases, consensus is not strictly necessary to maintain replicated data structures. Replicated data structures can be realized using a conflict-free replicated data type (CRDT) [Sha+11]. A CRDT satisfies the mathematical property of Strong Eventual Consistency (SEC), which ensures that replicas are consistent as soon as they observe and execute the same set of operations.

This allows programming distributed serverless applications that do not have to explicitly worry about synchronization between actors that share a global state. Instead, actors can *concurrently* read and update a shared variable, without coordination.

## 4.4 Summary

Causal message ordering and causal consistency for shared memory are both compatible with availability. On top of causality, atomic transactions and convergence guarantees can be supported, using CRDTs, without impacting availability [Zaw+15; Akk+16]. This improved memory model is called Transactional Causal Plus Consistency (TCC+) and is compatible with actor-based serverless frameworks because of the isolation properties of MVCC.

# Part II

Contributions

# 5 Unified Model

MVCC is compatible with the actor model because both, message-passing and shared memory, are based on the principle of maintaining independent, isolated, and concurrent operations on shared resources. In the actor model, each actor is a self-contained unit of computation that can perform independent operations on its private state, and it communicates with other actors by exchanging messages. Similarly, in MVCC, each transaction operates on a consistent snapshot of the database, which ensures that it sees a consistent and isolated view of the data, regardless of the concurrent updates made by other transactions. MVCC allows multiple transactions to access the same data simultaneously, without interfering with each other, which is compatible with the independent and concurrent operation of actors in the actor model.

Cases such as Figure 1.2 are a common architectural scheme. This design is commonly used in serverless services, to avoid message size limitations [Awsb]. To avoid these issues, consistency is needed that unifies the message view and the shared memory view.

In this chapter, we unify the definitions of consistency in message-passing (Chapter 3) with shared memory (Chapter 4). We call this model *Transactional-Turn Causal Consistency* (TTCC). This unified model cleanly integrates turns with transactions.

We first describe a formal design that unifies causal consistency for message passing and shared memory. Our unified model demonstrates how to apply the properties of *separate* causal contexts into a *mutually* causally consistent model. Then, in a second section, we unify the isolation property of the actor model's message passing with isolated snapshots used in shared memory systems.

## 5.1 Design Objectives

Our requirements are to provide a mutually consistent shared object and message passing memory model, while ensuring the strongest consistency model that is compatible with availability and actor.

TTCC provides distributed access to shared data. A client serverless function can read, update and send messages on an arbitrary node in the cloud, with seamless, atomic and causal, data replication and message delivery guarantees.

In the light of what we saw in the existing work, we will here justify some protocol choices used in our approach, which aims to solve the shortcomings observed in some existing systems.

We now turn to a system design for satisfying the above requirements efficiently. Our design is a unification of the definition of causal message delivery [Bur14; VV16] and causal shared memory [CBG15]. Our design must remain compatible with stateful serverless frameworks; specifically, data observed in a serverless function is always isolated from other functions. It should also remain available.

The trade-off is that increasing consistency guarantees, from eventually to causal consistency, requires extra metadata for messages and shared objects, which has a direct performance impact on response time and throughput (described later).

TTCC ensures convergence by using CRDTs, which merge concurrent conflicting operations deterministically [Sha+11].

## 5.2 Causal consistency

Causal consistency ensures that updates to shared objects and messages are observed in the order determined by the causality of operations. Causal consistency is separately defined for message passing in the actor model (Section 3.2) and for shared memory (Section 4.1). Messages and shared objects are *independently* causally consistent (using their own causal context) but are not causally consistent with each other. This separation in causal context can cause inconsistencies (Figure 1.2 and 2.6). TTCC avoids such inconsistencies by considering the causal context for messages and shared objects as a single, unified causal context.

Let us consider a joint actor/shared memory model. The actor is a sequence of turns, where the reception of a message triggers a turn. Actors may share memory. In

this model, *events* are sending and receiving messages, noted $send(m)$ and $rcv(m)$ respectively, and reads and updates operations to the shared memory, noted $rd(x, v)$ and $upd(x, v)$ respectively.

In the unified model, actors communicate through any mixture of message-passing and shared-memory access. An execution is *causally consistent for shared memory and messages* if and only if it satisfies the common, message-passing, and memory rules above (1)–(9), as well as the following *interaction rules*:

(10) An update does not overtake a message:

$$send(m) \xrightarrow{vis} wr(x, v) \implies \neg(rd(m, v) \xrightarrow{PO} rcv(m))$$

(11) A message does not overtake an update:

$$wr(x, v_1) \xrightarrow{vis} wr(x, v_2) \xrightarrow{vis} send(m) \implies \neg(rcv(m) \xrightarrow{PO} rd(m, v_1))$$

These rules define visibility when messages interact with shared-memory operations.

**Message visibility order** Rule (10) states that if an actor writes version $v$ to $x$ knowing $send(m)$, then the receiving actor must receive $m$ before observing version $v$ for key $x$.

**Shared objects visibility order** Rule (11) states that if an actor sends $m$ while knowing $wr(x, v_2)$, then the destination actor must no longer observe the earlier $v_1$ after receiving $m$. Indeed, upon $m$ reception, the receiving actor sees the $send(m)$ causal dependencies, i.e., $wr(x, v_1) \xrightarrow{vis} wr(x, v_2)$. Hence, the read must return $v_2$, the freshest visible version of $x$.

The system is *causally consistent* if: $e \xrightarrow{\text{vis}} f$, and some actor observes both $e$ and $f$, then it observes $e$ before observing $f$; and if there exists updates $x_1$ and $x_2$ to some object $x$, such that $x_1 \xrightarrow{\text{vis}} x_2 \xrightarrow{\text{vis}} e$ for some event $e$, then an actor that both observes $e$ and reads $x$ must observe $x_2$.

## 5.3 Isolation

For isolation, a turn processes a single received message and observes a causally consistent snapshot. A turn is itself a sequence of transactions, in which message sends and updates occur atomically (i.e., all if the transaction commits, or none, if it aborts).

To preserve atomicity with causal consistency and intuitive error handling, our model only allows the sending of the first message per destination actor per transaction.

### 5.3.1 Transactional turn

An actor is either ready to accept a message, or busy processing a message. A *turn* is the processing of a single message by an actor until completion (Section 3.3). An actor executes a single turn at a time, processing a single message per turn, and it runs the turn to completion without interruptions, before waiting for the next message. In other words, a turn cannot be interrupted by the reception of a message. Take for example Figure 5.1, where actor A processes message $m1$. During the processing of message $m1$, actor A may modify its local state, update a shared object and/or send a message ($m2$), which would trigger a new turn for actor B.

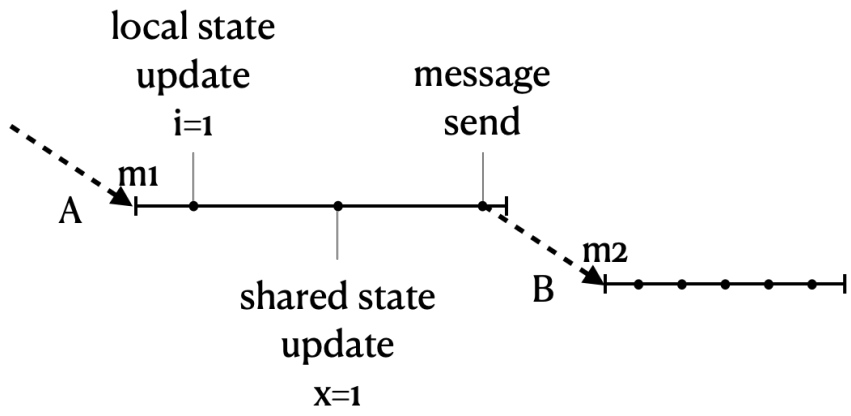



**Fig. 5.1.:** Example of operations that can occur during an actor's turn.

Each turn observes a causally-consistent snapshot, and its message sends, and its memory updates occur atomically per turn (i.e., all if the turn commits, or none, if it aborts). A memory update only affects the turn's causally-consistent snapshot, while messages are buffered until the transaction commits. On commit, updated objects are replicated to other nodes and messages are sent to their destination actor.

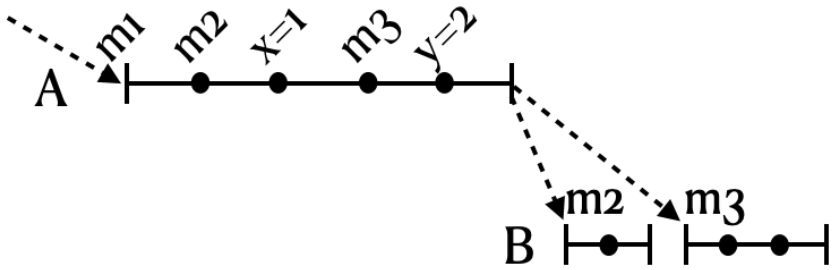



**Fig. 5.2.:** Reception of messages m2 and m3 breaks the atomic property of a transaction.

This mechanism ensures that an actor remains isolated as the turn's snapshot is isolated.

A turn's snapshot is exposed through the use of a transaction. Multiple transactions may occur during an actor's turn. Each transaction commit that contains an update operation to a shared object and/or sending a message, generates a new snapshot version that is instantaneously visible to following transactions in the same turn.

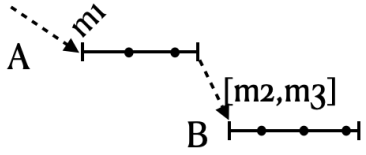
In summary, snapshots and transactions allow for isolation and atomicity, which makes the use of shared memory compatible with the actor model.

### 5.3.2 Single message per transaction

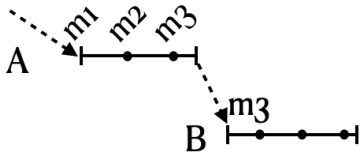
For actors, a turn corresponds to the processing of a single message. In other words, the reception of a message triggers an actor's turn. We first explore the scenario where we allow the sending of multiple messages (Figure 5.2) during an actor's turn.

On reception of message $m1$, actor A sends messages $m2$ and $m3$ to actor B. Actor B receives messages $m2$ followed by message $m3$, as messages are received in the order in which they are sent. Each message is processed in a *separate* turn, which is problematic as this breaks the expected atomic property of a transaction as messages $m2$ and $m3$ must be visible at the same time. When message $m2$ is processed, object $y$ is visible, but message $m3$ is not. This does not respect our unified model, in which message $m3$ is sent *before* update to object $y$ (Rule **??**).

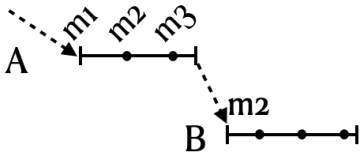
To prevent the violation of the atomic property of our unified model, we explore three possible scenarios, as illustrated in Figure 5.3. The first, in which we allow




**(a)** Messages m2 and m3 are received in a list.




**(b)** Message m2 is silently ignored while message m3 is delivered.




**(c)** Message m2 is delivered, while message m3 explicitly provokes an error.

**Fig. 5.3.:** Multiple scenarios are possible when multiple message are sent to the same destination actor. This figure explores the four possible scenarios.

sending multiple message to a single destination actor; and the second and third, where we restrict sending multiple message to a single destination actor.

**Scenario a** To circumvent the violation of the atomic property of a transaction, Figure 5.3a explores the scenario where messages $m2$ and $m3$ are both received at the same time. This implies that messages are grouped and processed in a single actor turn. This solution is viable but cumbersome for the developer, as this implies that a corresponding callback function is available to receive an ordered tuple of messages (in this case containing messages $m2$ and $m3$). Furthermore, this increases the size of a single message which may not be supported by certain serverless frameworks that limit the size of messages (e.g., AWS EventBridge is limited to 256 KB per message).

**Scenario b** An alternative solution is to restrict the transmission of multiple message to the same destination actor. Figure 5.3b explores the case where sending message $m2$ is aborted if a message ($m3$) is sent in the same transaction. More generally, only the last message to a destination actor is sent. Other messages are silently ignored, which is an undesirable action, or the transaction throws an error and aborts, which is not very convenient, especially with long transactions.

**Scenario c** Alternatively, Figure 5.3c explores the case where only the first message ($m2$) is actually sent. Following messages ($m3$) provoke an error (i.e. throw an exception) that can be caught and handled within the transaction's body.

This solution, while restrictive, has the advantage of being explicit (no silent errors), atomic and leaves the business logic to the developer.

To preserve atomicity (i.e., visibility at the same time) and intuitive error handling, our model only allows the sending of the first message per destination actor per transaction. Multiple transactions may occur in an actor's turn. In which case, they are run sequentially.

# 6 Protocol Design

In this section, we explore the design space for TTCC by modifying the metadata used to encode causal dependencies and the replication mechanisms to deliver causal messages. We explore three protocols that all use a version vector with one entry per node to track causal dependencies for shared objects, but differ with causal message metadata and delivery.

- Protocol 1 uses a single version vector to maintain a unified causality consistent state between messages and shared memory.
- Protocol 2 implements message-passing on top of shared memory. The shared memory consistency protocol is standard (based on a version vector, delaying out-of-order updates); sending a message appends it to a FIFO queue in memory.
- Protocol 3 uses separate metadata to mutually track causality for messages and shared memory. This simulates the case where both message and shared object protocols are causal but use their own independent metadata. Protocol 3 uses a version vector to track causality with shared objects, and an additional matrix for messages.

Our protocol executes in two phases: in an actor (when a transaction is executed and when a message is received) and in a *replicator* actor that is unique per node. Replicators of different nodes communicate with each other and are responsible for maintaining transactions, snapshots and replication. A transaction operation (read, update, send message) runs inside an actor, and accesses an isolated snapshot version that is managed by the local replicator. The replicator provides the latest *local*, causally consistent snapshot to new transactions. A transaction originating from the local node is immediately visible to local actors when it commits, as local actors share the latest local snapshot. However, a transaction arriving from a remote node is visible to local actors only after the preceding transactions have committed locally.

The replicator maintains causally consistent snapshots, without coordination, by maintaining a *globally stable snapshot* (GSS) [Akk+16], which represents a snapshot that is known to be available on all nodes. GSS is computed on every node and tracks

neighboring node's version vector. When a node receives a message, we compute GSS by finding the minimal value for each node. For example, node 1 is known to be at version vector $vv_1 = [2, 5, 1]$, and node 2 is at version vector $vv_2 = [2, 3, 3]$. Then, GSS computes to $[2, 3, 1]$. A transaction from a remote node is visible locally only when the GSS advances past the transaction's snapshot version.

**Causal message delivery** To implement causal message delivery, TTCC delays messages until all its causal dependencies are satisfied. Conversely, sending a message is non-blocking. Causal dependencies are propagated by piggy-packing metadata to messages. For instance, if an actor sends $m$ then $n$, the metadata of $n$ indicates that $n$ causally depends on $m$.

**Causal shared-memory** To maintain causal consistency for shared memory, TTCC maintains multiple versions of objects and exposes them through isolated snapshots. Write operations are non-blocking and replication is done asynchronously. When reading an object, TTCC materializes only the requested value for the given object, as opposed to all objects in the snapshot, to reduce compute and memory consumption.

**Memory-message interactions** TTCC unifies causal consistency for shared memory and causal message delivery, by considering the interactions between the two memory models. Messages are delayed until causally dependent messages are delivered (Rule (6)) *and* shared-memory is up-to-date (Rule (11)). A snapshot is causally visible, when causally dependent snapshots are available (Rule (9)). Visibility of a snapshot is not delayed by causally dependent messages as the reception of a message triggers an actor's turn, which exposes a causally consistent snapshot.

## 6.1 Protocol 1: Single Version Vector

Protocol 1 uses a single version vector to track causal dependencies for shared objects and messages. This protocol best reflects a *fully* unified memory-message model, where messages and shared memory both use the same data structure to encode their causal dependencies.

Metadata is embedded into a message, which is then used by the destination actor to ensure causal delivery. A message may be received by the destination actor and

delayed if the shared memory is not causal consistent with respect to the message's version vector.

We first describe the notation and definitions for the terms used to describe Protocol 1. Then, we present the execution of the protocol on an actor and on a replicator.

### 6.1.1 Notation and definitions

Table 6.1 introduces the notation followed in this section to describe the execution of our protocols on an actor and on a replicator. We assume a singleton Replicator $R$ on each node. A snapshot $S$ is a tuple composed of a version vector $vv_S$ and a dataset $data_S$. The $GSS$ is a snapshot that is known to be available on all nodes at a given point in time. $LLSS$ stores a set of local snapshots that are committed. When the protocol updates $GSS$, snapshots from $LLSS$ are merged into $GSS$ using CRDT logic. An ongoing transaction $T$ is stored in $ongoing$ at index $T$. $R$ stores its neighbor $n$'s version vector in $kvv$ at index $n$. When $kvv$ updates, the protocol recompute $GSS$. $lastVV$ stores the latest Version Vector seen by an actor.

| | |
|---|---|
| $R$ | Local replicator actor |
| $T$ | Transaction |
| $q_T$ | Queue containing messages for transaction $T$ |
| $S$ | Snapshot |
| $vv_S$ | Version vector of $S$ |
| $data_S$ | Dataset of $S$ |
| $GSS$ | Globally Stable Snapshot |
| $LLSS$ | Set of Locally Latest Stable Snapshots |
| $ongoing[T]$ | Ongoing transaction is stored at index $T$ |
| $kvv[n]$ | Known Version Vector for neighbor is stored at index $n$ |
| $m$ | Message sent between a pair of actors |
| $from_m$ | Sender actor of $m$ |
| $vv_m$ | Version Vector of $m$ |
| $lastVV$ | Last seen Version Vector |
| $B$ | Buffer for delayed messages |
| $+=$ | CRDT merge operation |

**Tab. 6.1.:** Notation used in the description of Protocol 1.

### 6.1.2 Execution on an actor

Algorithm 1 shows the pseudocode for Protocol 1 for executing transaction $T$ and the reception of message $m$ on an actor.

**Algorithm 1** Execution of Protocol 1 on actor a

```
1:  function START_TRANSACTION
2:      id_T ← send StartTrx to R
3:      return T
4:  function READ(id_T, key)
5:      v ← send Read(id_T, key) to R
6:      return v
7:  function UPDATE(id_T, key, v)
8:      send Update(id_T, key, v) to R
9:  function COMMIT(T)
10:     send Commit(T) to R
11: function ABORT(T)
12:     send Abort(T) to R
13:     clear q_T
14: function SEND_MSG(m, to)
15:     if to ∈ q_T then
16:         throw an exception
17:     else
18:         append m to q_T[to]
19: function CHECK_DEPENDENCIES(m)
20:     deps ← vv_m − from_m
21:     for all d ∈ deps do
22:         return lastVV[d] == d
23: function IS_DELIVERABLE(m)
24:     if vv_m ≤ lastVV &
25: vv_m[from_m] < lastVV[from_m] &
26: CHECK_DEPENDENCIES(m) then
27:         lastVV+ = vv_m
28:         return true
29:     else
30:         return false
31: function DELIVER_CAUSAL_MESSAGES
32:     for all m ∈ B do
33:         if IS_DELIVERABLE(m) then
34:             deliver m
35:             remove m from B
36: function ON_MESSAGE(m)
37:     if IS_DELIVERABLE(m) then
38:         deliver m
39:         DELIVER_CAUSAL_MESSAGES
40:     else
41:         B ← m
```

A transaction begins by sending a synchronous $StartTransaction$ message to its local replicator $R$, which contains its transaction id; we use a locally-generated UUID [LSM05] as it is unique and does not require coordination. $R$ responds with an initialized transaction snapshot, which contains the latest locally available snapshot, which is stored in $LLSS$. If $LLSS$ is empty, we use $vv_{GSS}$. Finally, if $GSS$ is empty, we use an empty version vector.

A read operation in a transaction sends a $Read$ message that contains $id_T$ and a $key$ to the requested value, to $R$. $R$ replies with $ReadSuccess$ message containing the requested value or a $NotFound$ message if the requested $key$ is missing.

A write operation sends an $Update$ message containing $id_T$, a $key$ and an associated $value$ to $R$. $R$ responds with an $UpdateSuccess$ message or an $UpdateFailure$ message.

A message sent in a transaction is stored in a buffer $q_T$ until the transaction is commits or aborts. To enforce our unified memory model, we verify that only one message is sent per a destination (Alg. 1, line 15). On commit, the actor sends a $Commit$ message containing the transaction id and $q_T$ to $R$. On abort, $q_T$ is emptied, and no messages are sent.

When it receives a message (Alg. 1, line 36), the actor checks if $m$ is causally deliverable. A message is causally deliverable if the following conditions are met:

(1) $vv_m \leq lastVV$; the receiving actor maintains the last seen shared memory version vector in $lastVV$. If the message's version vector is $> lastVV$, then shared memory is not up-to-date. The message cannot be delivered.

(2) $vv_m[from_m] < lastVV[from_m]$; the message's sequence number for the sender's entry contained in the version vector must be $<$ than the sequence number contained in $lastVV[from_m]$ for the sender's entry. This guarantees consecutive delivery of messages.

(3) $\forall d \leftarrow vv_m - vv_m[from_m], d == lastVV[d]$; finally, we check that causal dependencies for the message are satisfied. $d$ represents the dependencies for $m$ which is computed by selecting all entries in $vv_m$ except $from_m$. For all entries in $d$, the value of the version vector's entry must be equal to $lastVV[d]$.

If $m$ is not deliverable, it is appended to buffer $B$. After the delivery of $m$, the protocol checks $B$ for any other deliverable messages.

### 6.1.3 Execution on Replicator

Replicator $R$ is responsible for maintaining ongoing transactions, multiple snapshot versions (MVCC) and triggering replication to other replicators. Algorithm 2 shows the pseudocode of the protocol for executing transaction $T$ on $R$.

$R$ locally centralizes transaction operations and manages multiple snapshot versions. When $R$ receives a $StartTransaction$ for $T$ and $T \notin ongoing$, the protocol initializes the transaction context by appending the latest snapshot in $LLSS$ to $ongoing[T]$. $R$ replies with a message containing the latest $vv_{LLSS}$, which represents the latest locally available snapshot.

**Algorithm 2** Execution of Protocol 1 on replicator actor $R$.

1: **function** ON_START_TRX($T$)
2: **if** $ongoing[T]$ does not exist **then**
3: $ongoing[T] \leftarrow$ latest $LLSS$
4: **return** latest $vv_{LLSS}$
5: **function** ON_READ($T$, $key$)
6: $value = data_{GSS}$ for $key$
7: $value+ = data_{LLSS}$ for $key$
8: $value+ = data_{ongoing[T]}$ for $key$
9: **return** $value$
10: **function** ON_UPDATE($T$, $k$, $v$)
11: put $v$ in $ongoing[T]$ at $k$
12: **function** ON_COMMIT($T, vv_T$)
13: $commitVv \leftarrow$ latest $vv_{LLSS}$
14: **if** update or message $\in ongoing[T]$ **then**
15: increment $commitVv[self]$
16: $kvv[self] \leftarrow commitVv$
17: $LLSS[commitVv] \leftarrow ongoing[T]$
18: remove $T$ from $ongoing$
19: BROADCAST($LLSS[commitVv]$)
20: **function** BROADCAST($S$)
21: **for** all $n \in allNodes$ **do**
22: send $SnapshotUpdate(S)$ to $n$
23: **function** ON_SNAPSHOT_UPDATE($from$, $S$)
24: **if** $vv_S$ is concurrent **then**
25: $vv_S+ = vv_{LLSS}$
26: $data_S+ = data_{LLSS}$
27: update $LLSS$ with $vv_S$ and $data_S$
28: **else**
29: update $LLSS$ with $vv_S$ and $data_S$
30: update $kvv[from]$ with $vv_S$
31: UPDATE_GSS
32: **function** UPDATE_GSS
33: **for** $i = 1, 2, \ldots, size(kvv)$ **do**
34: $vv_{GSS} \leftarrow min(kvv[i])$
35: $data_{GSS} =$ data from $GSS$
36: $data_{GSS}+ = data_{LLSS}$ from $vv_{LLSS}$ until $vv_{GSS}$
37: $GSS \leftarrow (vv_{GSS}, data_{GSS})$
38: remove merged data from $LLSS$

When $R$ receives a $Read$ message (Alg. 2, line 5), the protocol first materializes the requested data as snapshots only contain a partial view of the overall data set. For transaction $T$, snapshots are contained in $GSS$, $LLSS$ and in $ongoing[T]$. The protocol requires that $T \in ongoing$. The process of materialization consists in applying all operations from previous snapshots for a given key to form the final value. The steps to materialize value $v$ for key $k$ are:

(1) First, we set the initial value of $v$ to $data_{GSS}$ for $k$. This value is known to be available on all nodes. If $k \notin data_{GSS}$, $v$ is not set.

(2) Then, all values for $k$ that are $\leq vv_T \in LLSS$ are merged using the underlying CRDT merge operation. The resulting value is merged into $v$. If, $k \notin LLSS$, there is no resulting value, and nothing is merged. $LLSS$ corresponds to snapshots that have been locally committed but are not yet merged into $GSS$.

(3) Finally, $data_{ongoing[T]}$ is merged into $v$. This applies update operations to $k$ from the current transaction.

An important note to consider is that value $v$ for key $k$ is materialized, as described above, for every read operation. This is inefficient and may be optimized by using a cache (we discuss this later).

On reception of an $Update$ message, key $k$ and value $v$ are stored in $data_{ongoing[T]}$. $ongoing[T]$ corresponds to the current ongoing transaction's snapshot.

When $R$ receives a $Commit$ message (Alg. 2, line 12), the latest local commit version vector ($vv_{LLSS}$) is used as a basis to the current transaction's commit time stamp. If $ongoing[T]$ contains update operations or $q_T$ is not empty, the protocol increments $cvv_T$. Then, the protocol updates $kvv[self]$ with $cvv_T$ to maintain an updated version vector for the current node. Finally, to terminate the commit and make the new snapshot visible to other actors, $data_T$ moves from $ongoing$ into $LLSS$ at $cvv_T$. Finally, the resulting snapshot is broadcast to all nodes.

On reception of an snapshot broadcast update message (Alg. 2, line 23), $R$ checks if $vv_S$ is concurrent with a snapshot contained in $LLSS$. This may be the case, as local transactions can commit without coordination with other nodes. If $vv_S$ is concurrent, we merge $vv_S$ and $data_S$ with $vv_{LLSS[vv_S]}$ and $data_{LLSS[vv_S]}$ respectively. Then, we update $LLSS$ with the resulting snapshot. If $vv_S$ is not concurrent, we update $LLSS$ with $S$. Finally, $kvv[from]$ is set to $vv_S$ before updating $GSS$.

We update $GSS$ by first computing the new version vector $vv_{GSS}$ by finding the $min$ value for each $kvv$ [Akk+16]. We merge $data_{GSS}$ with all $data_{LLSS} \leq vv_{GSS}$.

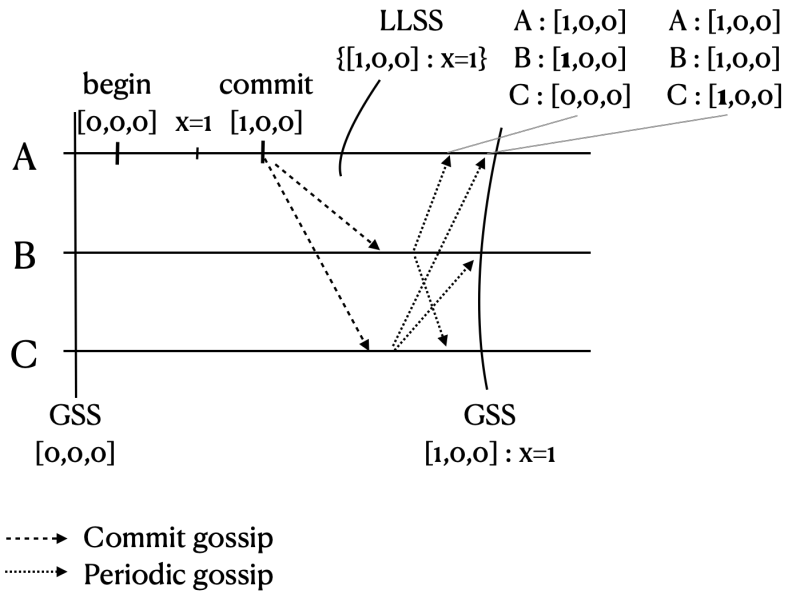



**Fig. 6.1.:** Global Stable Snapshot update mechanism.

Finally, we remove snapshots from $LLSS$ that are merged into $GSS$. The $GSS$ update mechanism is visualized in Figure 6.1.

## 6.2 Protocol 2: Version Vector + Shared Message Queue

Protocol 2 uses the available causal shared memory between actors to store a unique FIFO queue that represents an extension to an actor's message mailbox. Each actor has a single queue.

An actor that receives a message subscribes to its local replicator, who notifies the actor when at least one message is available for delivery. When an actor receives such a notification, the actor dequeues and delivers all the messages that are present in the queue. A message is guaranteed to be causally delivered as it relies on the underlying causally consistent shared memory. In this protocol, like in Protocol 1, metadata usage is minimal as we only use a single version vector for causal shared memory. However, all messages to a destination actor from a node are serialized into the queue in shared memory. Furthermore, messages rely on the replication protocol of the shared memory, which is triggered periodically and broadcasts the replication messages to all nodes, even if the destination actor is not present on the destination node.

### 6.2.1 Notation and definitions

Table 6.2 introduces the notation followed in this section. We assume a singleton replicator $R$ on each node. A snapshot $S$ is a tuple composed of a version vector $vv_S$ and a dataset $data_S$. The $GSS$ is a snapshot that is a view of the data store that is known to be available on all nodes at a given point in time. $LLSS$ stores local snapshots that are committed but are not merged into the $GSS$ yet. An ongoing transaction $T$ is stored in $ongoing$ at index $T$. $R$ stores its neighbor $n$'s version vector in $kvv$ at index $n$. $R$ stores the address of subscribed actors in $subs$ at index $k$.

| | |
|---|---|
| $S$ | Snapshot |
| $vv_S$ | Version vector of $S$ |
| $data_S$ | Dataset of $S$ |
| $GSS$ | Globally stable snapshot |
| $LLSS$ | Locally latest stable snapshot |
| $ongoing[T]$ | Ongoing operations for transaction $T$ |
| $kvv[n]$ | Version vector for neighbor $n$ |
| $m$ | Message sent between a pair of actors |
| $from_m$ | Sender actor of $m$ |
| $B$ | Buffer for delayed messages |
| $R$ | Local replicator actor |
| $R_n$ | Replicator actor at node $n$ |
| $T$ | Transaction |
| $subs$ | list of all subscribed actors |
| $subs[k]$ | list of subscribed actors for key $k$ |
| $+=$ | CRDT merge operation |

**Tab. 6.2.:** Notation used in the description of Protocol 2.

### 6.2.2 Execution on an actor

Algorithm 3 shows the pseudocode for Protocol 2 for executing transaction $T$ on actor $a$.

Algorithm 3 is very similar to Algorithm 1 for Protocol 1. Transaction start, read, update, commit and abort operations remain the same (See Alg. 1). The key difference stands in how a message is sent and how an actor receives a message.

**Algorithm 3** Execution of Protocol 2 on actor a

```
1:  function ACTOR_INIT
2:      send Subscribe(self) to R
3:  function START_TRANSACTION
4:      id_T ← send StartTrx to R
5:      return T
6:  function READ(id_T, key)
7:      v ← send Read(id_T, key) to R
8:      return v
9:  function UPDATE(id_T, key, v)
10:     send Update(id_T, key, v) to R
11: function COMMIT(T)
12:     send Commit(T) to R
13: function ABORT(T)
14:     send Abort(T) to R
15:     clear q_T
16: function SEND_MSG(id_T, m, to)
17:     q ← send Read(id_T, to) to R
18:     append m to q
19:     send Update(id_T, to, q) to R
20: function ON_Q_CHANGE(q)
21:     for all m ∈ q do
22:         deliver m to self
23:     empty q
```

Protocol 2 relies on a notification mechanism that alerts an actor of a change on a specified shared object. When actor $a$ initializes (Alg. 3, line 1), it sends a subscription message to changes on a shared message queue object (Alg. 3, line 2) to its local replicator. The subscription key uniquely identifies the subscribing actor. Usually, actor frameworks identify actors using a unique actor address. Such an identifier is suitable to use a subscription key.

Actor $a$ now expects to receive a notification when at least one message is available in shared memory. The notification message contains a FIFO queue ($q$) which includes all causally available messages for actor $a$. On reception of a notification message (Alg. 3, line 20), messages are delivered to $a$. Messages contained in $q$ are guaranteed to be causally consistent with shared objects, as $q$ is also a shared object.

When actor $a$ sends a message $m$ to a destination actor (Alg. 3, line 16), Protocol 2 performs a read and update operation. The protocol first retrieves the queue ($q$) specific to the destination actor. Then, $m$ is appended to $q$. Finally, the protocol sends an $Update$ message containing $q$ to $R$. On commit, sent messages are replicated to other nodes, which will trigger an alert to subscribed actors and messages will be delivered.

### 6.2.3 Execution on Replication actor

Algorithm 4 shows the pseudocode of the protocol for executing transaction $T$ on $R$.

**Algorithm 4** Execution of Protocol 2 Replicator Actor $R$

```
1: function ON_SUBSCRIBE(k, from)
2:     append from to subs[k]
3: function ON_START_TRX(T)
4:     if ongoing[T] does not exist then
5:         ongoing[T] ← latest LLSS
6:         return latest vv_LLSS
7: function ON_READ(T, key)
8:     value = data_GSS for key
9:     value+ = data_LLSS for key
10:    value+ = data_ongoing[T] for key
11:    return value
12: function ON_UPDATE(T, k, v)
13:    update ongoing[T] for k with v
14: function ON_COMMIT(T, vv_T)
15:    commitVv ← latest vv_LLSS
16:    if update operation or message send ∈ ongoing[T] then
17:        increment commitVv[self]
18:    kvv[self] ← commitVv
19:    LLSS[commitVv] ← ongoing[T]
20:    remove T from ongoing
21:    notify subscribers for all modified keys in T
22:    BROADCAST(LLSS[commitVv])
23: function BROADCAST(S)
24:    for all n ∈ allNodes do
25:        send SnapshotUpdate(S) to n
26: function ON_SNAPSHOT_UPDATE(from, S)
27:    if vv_S is concurrent then
28:        vv_S+ = vv_LLSS
29:        data_S+ = data_LLSS
30:        update LLSS with vv_S and data_S
31:    else
32:        update LLSS with vv_S and data_S
33:    update kvv[from] with vv_S
34:    UPDATE_GSS
35: function UPDATE_GSS
36:    for i = 1, 2, ..., size(kvv) do
37:        vv_GSS ← min(kvv[i])
38:    data_GSS = data from GSS
39:    data_GSS+ = data_LLSS from vv_LLSS until vv_GSS
40:    GSS ← (vv_GSS, data_GSS)
41:    remove merged data from LLSS
```

Algorithm 4 is very similar to Algorithm 2 for Protocol 1. Most message callback functions remain the same. The key difference is in how an actor requests to subscribe to any change to a given key.

In addition to the same behaviors as in Protocol 1, $R$ also handles actor's subscription to a given key $k$ (Alg. 4, line 1). On reception of a subscription message from $from$, $R$ appends $from$ to $subs[k]$.

Protocol 2 notifies subscribed actors when an update occurs to a shared object. On commit, after the transaction snapshot moves from $ongoing$ to $LLSS$, $R$ sends a notification message is sent to all subscribed actors in $subs$ (Alg. 4, line 21). A notification message is sent to an actor $a$ if, $a$ is subscribed to key $k$. Protocol 2 leverages this mechanism to notify actors that messages are available for delivery.

## 6.3 Protocol 3: Version Vector + Matrix

Protocol 3 uses a version vector to track causal dependencies for shared objects and an additional matrix that uses one integer per source-destination actor pair for messages. This scenario is similar to an ad-hoc scenario where shared memory and messages are mutually causally consistent, but their causal context is not merged into a single causal context. This would be the case if a user wants to manually ensure causality in two separate systems. The advantage of this method is a fully separate mechanism for shared objects and message passing. However, additional metadata is required, which increases with the number of communicating actor pairs.

### 6.3.1 Notation and definitions

Table 6.3 introduces the notation followed in this section. We assume a singleton Replicator $R$ on each node. A snapshot $S$ is a tuple composed of a version vector $vv_S$ and a dataset $data_S$. The $GSS$ is a snapshot that is a view of the data store that is known to be available on all nodes at a given point in time. $LLSS$ stores local snapshots that are committed but are not merged into the $GSS$ yet. An ongoing transaction $T$ is stored in $ongoing$ at index $T$. $R$ stores its neighbor $n$'s version vector in $kvv$ at index $n$.

| | |
|---|---|
| $S$ | Snapshot |
| $vv_S$ | Version vector of $S$ |
| $data_S$ | Dataset of $S$ |
| $GSS$ | Globally stable snapshot |
| $LLSS$ | Locally latest stable snapshot |
| $ongoing[T]$ | Ongoing operations for transaction $T$ |
| $kvv[n]$ | Version vector for neighbor $n$ |
| $m$ | Message sent between a pair of actors |
| $from_m$ | Sender actor of $m$ |
| $m_{seq_{x,y}}$ | Sequence number for $x, y$ actor pair for $m$ |
| $lastSeq_{x,y}$ | Last seen sequence number for $x, y$ actor pair |
| $B$ | Buffer for delayed messages |
| $R$ | Local replicator actor |
| $R_n$ | Replicator actor at node $n$ |
| $T$ | Transaction |
| $q_T$ | FIFO queue containing messages |
| $+=$ | CRDT merge operation |

**Tab. 6.3.:** Notation used in the protocol description.

### 6.3.2 Execution on a causal actor

Algorithm 5 shows the pseudocode for Protocol 3 for executing transaction $T$ on actor $a$.

Algorithm 5 handles message delivery with respect to the shared memory causal context and message causal context. Although very similar to algorithms 1 and 3 with the handling of shared memory, message metadata uses a separate data structure and is thus treated differently.

On reception of a message (Alg. 5, line 39), both causal contexts are checked for causal delivery. A message $m$ is causally deliverable if all the following conditions are met:

(1) $vv_m \leq lastVV$; the receiving actor maintains a last seen shared memory version vector in $lastVV$. If the message's version vector is $> lastVV$, then shared memory is not up-to-date. The message cannot be delivered and is delayed for future delivery.

**Algorithm 5** Execution of Protocol 3 on actor a

```
1: function START_TRANSACTION
2:     id_T ← send StartTrx to R
3:     return T
4: function READ(id_T, key)
5:     v ← send Read(id_T, key) to R
6:     return v
7: function UPDATE(id_T, key, v)
8:     send Update(id_T, key, v) to R
9: function COMMIT(T)
10:    send Commit(T) to R
11: function ABORT(T)
12:    send Abort(T) to R
13:    decrement seq_{from_m,to_m}
14:    clear q_T
15: function SEND_MSG(m, to)
16:    if to ∈ q_T then
17:        throw an exception
18:    else
19:        increment seq_{from_m,to_m}
20:        add seq and vv_LLSS to m
21:        append m to q_T
22: function CHECK_DEPENDENCIES(m)
23:    deps ← lastSeq − from_m
24:    for all d ∈ deps do
25:        return d_to == lastSeqself, d_from
26: function IS_DELIVERABLE(m)
27:    if vv_m ≤ lastVV &
28: lastSeq_{from_m,self} == m_{seq_{from_m,self}} −
    1 &
29: CHECK_DEPENDENCIES(m) then
30:        lastVV+ = vv_m
31:        return true
32:    else
33:        return false
34: function DELIVER_CAUSAL_MESSAGES
35:    for all m ∈ B do
36:        if IS_DELIVERABLE(m) then
37:            deliver m
38:            remove m from B
39: function ON_MESSAGE(m)
40:    if IS_DELIVERABLE(m) then
41:        deliver m
42:        DELIVER_CAUSAL_MESSAGES
43:    else
44:        B ← m
```

(1) $lastSeq_{from_m,self} == m_{seq_{from_m,self}} - 1$; the receiving actor maintains the last sequence numbers for each actor pair that the actor has received. A message $m$'s sequence number is compared to the last seen sequence number, for a sender-receiver actor pair. If $m$'s sequence number is consecutive, it is eligible to be delivered.

(1) $d_to == lastSeqself, d_from$; finally, the rest of the sequence matrix is compared to the maintained last sequence matrix. If the $m$'s matrix elements, excluding the sender, are equal to the last sequence for the given actor pair, $m$ is deliverable.

If any of the previous rules fail, delivery of message $m$ is buffered in $B$ for later retry. $B$ is iterated over for potential message delivery every time a new message arrives, causing a potential causal context update.

When actor $a$ sends a message $m$ to a destination actor, the protocol first increments $seq_{from_m,to_m}$ (Alg. 5, line 19). This ensures that $m$ is delivered in causal order with respect to other messages from the sender to the receiving actor. $seq_{from_m,to_m}$ and $vv_{LLSS}$ is added to $m$ before being appended to $q_T$. Messages are sent on commit.

### 6.3.3 Execution on Replication actor

Algorithm 6 shows the pseudocode of the protocol for executing transaction $T$ on $R$.

Algorithm 6 is similar to algorithm 2 for Protocol 1. Concurrency logic is added on an actor, when a message is sent, before reaching $R$. $R$ is agnostic of message and metadata type, thus no particular logic is necessary.

## 6.4 Summary

With Protocols 1, 2 and 3, we explore the design space of our unified memory model by varying the metadata used to maintain causality. In all three protocols, we use a version vector to track causal consistency for shared memory. Protocol 2 uses a causal shared memory to send messages. Protocol 3 uses an additional matrix to track causality with messages.

In all three protocols, the size of the version vector is proportional to the number of replicator actors ($m$). In Protocol 3, the size of the matrix is equal to the number

**Algorithm 6** Execution of Protocol 3 on replicator actor $R$.

1: **function** ON_START_TRX($T$)
2: **if** $ongoing[T]$ does not exist **then**
3: $ongoing[T] \leftarrow$ latest $LLSS$
4: **return** latest $vv_{LLSS}$
5: **function** ON_READ($T$, $key$)
6: $value = data_{GSS}$ for $key$
7: $value+ = data_{LLSS}$ for $key$
8: $value+ = data_{ongoing[T]}$ for $key$
9: **return** $value$
10: **function** ON_UPDATE($T$, $k$, $v$)
11: update $ongoing[T]$ for $k$ with $v$
12: **function** ON_COMMIT($T, vv_T$)
13: $commitVv \leftarrow$ latest $vv_{LLSS}$
14: **if** update operation or message send $\in ongoing[T]$ **then**
15: increment $commitVv[self]$
16: $kvv[self] \leftarrow commitVv$
17: $LLSS[commitVv] \leftarrow ongoing[T]$
18: remove $T$ from $ongoing$
19: BROADCAST($LLSS[commitVv]$)
20: **function** BROADCAST($S$)
21: **for** all $n \in allNodes$ **do**
22: send $SnapshotUpdate(S)$ to $n$
23: **function** ON_SNAPSHOT_UPDATE($from$, $S$)
24: **if** $vv_S$ is concurrent **then**
25: $vv_S+ = vv_{LLSS}$
26: $data_S+ = data_{LLSS}$
27: update $LLSS$ with $vv_S$ and $data_S$
28: **else**
29: update $LLSS$ with $vv_S$ and $data_S$
30: update $kvv[from]$ with $vv_S$
31: UPDATE_GSS
32: **function** UPDATE_GSS
33: **for** $i = 1, 2, \ldots, size(kvv)$ **do**
34: $vv_{GSS} \leftarrow min(kvv[i])$
35: $data_{GSS} =$ data from $GSS$
36: $data_{GSS}+ = data_{LLSS}$ from $vv_{LLSS}$ until $vv_{GSS}$
37: $GSS \leftarrow (vv_{GSS}, data_{GSS})$
38: remove merged data from $LLSS$

of total actors in the system ($n$). Table 6.4 summarizes the size complexity of the metadata used in the three protocols to track causality.

| Protocol | Metadata | Additional Metadata Space Complexity | |
|---|---|---|---|
| | | Shared Memory | Message |
| 1 | Version Vector | $O(m)$ | $O(m)$ |
| 2 | Version Vector | $O(m)$ | N/A |
| 3 | Version Vector + Matrix | $O(m)$ | $O(n^2)$ |

**Tab. 6.4.:** Summary of metadata used in Protocols 1, 2 and 3. $m$ is the number of replicators and $n$ is the number of actors of the whole system.

# 7 System API and Implementation

TTCC is designed to provide a simple API for using an integrated shared memory in an actor framework. This section presents our reference implementation and programming interface for Protocol 1 (Section 6.1) for the Akka actor framework. The code is open-source and available on GitHub [1].

We first introduce the Akka actor framework, which we modify to implement TTCC. Then, we discuss on the modularity of our design. In the following section, we introduce our transactional API that exposes a causally consistent snapshot to an actor. Finally, we explain our implementation for causally consistent shared memory and message delivery for Akka.

## 7.1 Akka Actor Framework

We implement TTCC on top of the Akka actor framework [Akka]. Akka is an modern and open-source implementation of the actor model based on Agha's work [Agh85]. De Koster [DKVCDM16] classifies Akka as a *classic actor model*, which makes it a suitable candidate for TTCC as the *Isolated Turn Principle* applies (actor isolation).

Additionally, Akka serves as the base actor framework for multiple serverless frameworks, such as Akka Serverless [Akkb] and Apache OpenWhisk [Apa]. Lastly, Akka has a limited but working *DistributedData* extension [Akkc] that enables actors to share data using eventual consistency guarantees.

An actor accesses data in the shared store through a replicator actor that provides a key-value API and that handles data replication. The replicator is a singleton instance per node. In other words, an Akka cluster may only have one instance of a replicator actor per cluster node (Figure 7.1). Replicator actors know other replicators and can communicate using message-passing. The replicator actor spreads object updates

[1] https://github.com/benoitmartin88/akka/tree/unified_cc_single_vv

to its neighbors via direct replication and gossip-based dissemination. In this key-value API, a key is a unique identifier of a CRDT data value. A CRDT supports concurrent updates from any node without coordination and provide high read and write availability, with low latency.

Akka's *DistributedData* extension enables the sharing of data between actors, but without any guarantees. Data will eventually be replicated to all nodes in an unknown order. This may cause inconsistencies and errors.

Our solution consists in applying our unified memory model next to the existing Akka *DistributedData* extension. This will prevent inconsistencies when using messages and shared objects between actors.

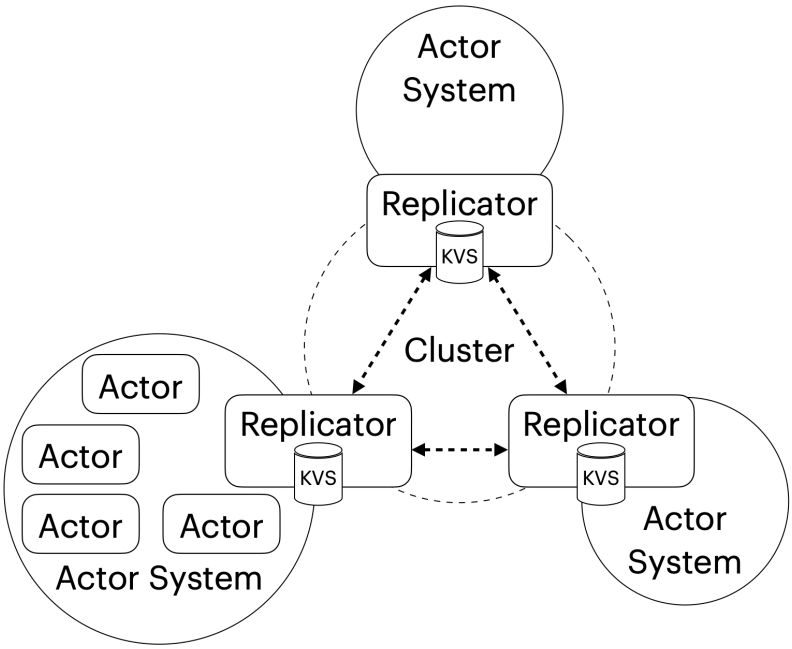



**Fig. 7.1.:** Akka DistributedData extension general architecture.

## 7.2 Modular design

The actor model is inherently modular. Actors are isolated and communicate only using asynchronous message-passing, making it easy to replace a functional component if needed. Our implementation is not a replacement of the existing Akka mechanisms for shared memory or inter-actor messages, but rather an addition that can be used conjointly with the existing *DistributedData* extension. This approach has the advantage of bringing the ability to adapt an existing application without breaking it. Our implementation is retro-compatible with an existing Akka code-base.

## 7.3 API

Our implementation exposes a causally consistent snapshot through the use of a transactional API (Listing 7.1). Our transactional API exposes read, update and message operations using a *context* object that hides all the protocol's complexities, which has the advantage of simplifying the user interface without limiting functionality. This context encapsulates a causally consistent snapshot that is set when the transaction starts and on which transaction operations are applied.

By default, Akka supports the following replicated data-types:

- Counters: GCounter, PNCounter
- Sets: GSet, ORSet
- Maps: ORMap, ORMultiMap, LWWMap, PNCounterMap
- Registers: LWWRegister, Flag

Additional custom data types are also supported, but details are omitted.

```
1 val flagKey = FlagKey("my flag")
2 val counterKey = PNCounterKey("my counter")
3
4 val t = new Transaction((context) -> {
5   var flag = context.get(flagKey) // get flag value
6   flag = flag.switchOn            // toggle flag
7   context.update(flagKey, flag)  // update flag value
8 })
9 t.commit()
```

**Listing 7.1:** Example of ATCC transactional's API. A CRDT Flag type is toggled inside the scope of a transaction which is later commited.

## 7.4 Implementation

Our reference implementation of TTCC is in Scala and sits on top of the existing Akka *DistributedData* extension. We first describe our implementation architecture for causal shared memory before explaining causal delivery of inter-actor messages.

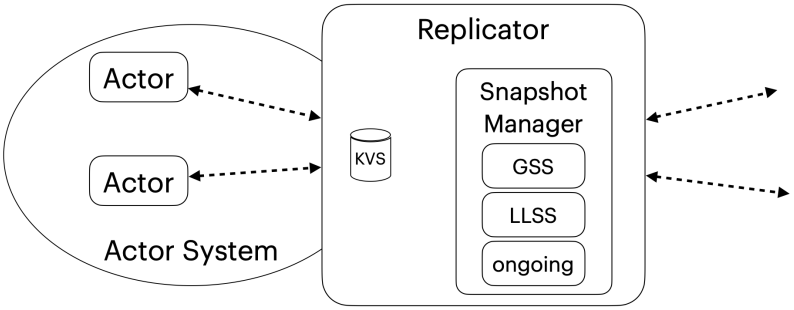



**Fig. 7.2.:** Detail view of Akka's Replicator actor. An extension named SnapshotManager handles read and updates to GSS, LLSS and ongoing data structures.

### 7.4.1 Causal shared memory

We extend Akka's *DistributedData* extension to implement algorithms 1 and 2 for the TTCC protocol without breaking any of the native Akka implementation. Our contribution adds a *SnapshotManager* class that is responsible for managing current transactions and snapshot versions. The *SnapshotManager* class implements TTCC logic from Algorithm 2 and is instantiated in the Replicator actor, which makes it accessible to the replicator when messages are received (Figure 7.2). We use the following data structures to best implement TTCC:

**Snapshot** A snapshot is a tuple composed of a version vector and a Map that represents stored keys and CRDT values.

**Global Stable Snapshot** $GSS$ is a snapshot and is updated when we update the data structure that holds known neighbor's version vector. A new version vector is received when a neighboring *SnapshotManager* sends a replication message.

**Locally Last Stable Snapshot** $LLSS$ represents local committed transactions that are not yet merged into $GSS$. $LLSS$ is a TreeMap that stores version vectors and a Map of committed data (keys and values). A TreeMap makes data materialization fast ($O(log(n))$) as we need to find all snapshots that are smaller or equal to a given version vector.

**Current transactions** We store local current ongoing transactions in a HashMap data structure that represents the transaction's id and snapshot.

We update the Akka's replicator actor to include additional messages and logic to handle transactions. Listing 7.2 show three extra message handlers: *TrxPrepare*, *TrxCommit* and *TrxAbort* that correspond to a commit prepare, commit and abort respectively. Get and update message handlers are modified to include an optional

transaction parameter, which is used to check if the operation is part of a transaction. This ensures retro-compatibility with existing Akka logic (eventually consistent store) without having to create new message handlers.

```
val normalReceive: Receive = {
[...]
case TrxPrepare(tid, req) => receiveTrxPrepare(tid, req)
case TrxCommit(trxn, req) => receiveTrxCommit(trxn, req)
case TrxAbort(tid, req)  => receiveTrxAbort(tid, req)
case Get(key, readC, req, trxn) => receiveGet(key, readC, req, trxn)
case u @ Update(key, writeC, req, tid) => receiveUpdate(tid, key,
    u.modify, writeC, req)
[...]
```

**Listing 7.2:** Modifications to Akka's Replicator actor. Additional message callback methods are added to handle transaction prepare, commit, abort, get and update operations.

```
def receiveTrxCommit(trxn: Trx.Context, req: Option[Any]): Unit = {
    if (!snapshotManager.currentTransactions.contains(trxn.tid)) {
      replyTo ! TrxCommitError(
        "no transaction with id " + trxn.tid + ": prepare not called or
            wrong transaction id",
        req)
    } else {
      val increment = trxn.messages.exists(x => x._2.nonEmpty)
      val commitVV = snapshotManager.commit(trxn.tid, increment)

      snapshotManager.updateKnownVersionVectors(selfUniqueAddress,
          commitVV)

      triggerSnapshotGossip(
        Some(commitVV),
        Some(snapshotManager.currentTransactions(trxn.tid)._1._2),
        Some(trxn.messages.toMap))

      snapshotManager.clear(trxn.tid)

      replyTo ! TrxCommitSuccess(req)
    }
}
```

**Listing 7.3:** Replicator actor's transaction commit callback method. The transaction commit message is first checked for validity before a commit version vector is computed.

Listing 7.3 shows the message handler responsible for receiving a transaction commit. On reception of a *TrxCommit* message, we first check that the message contains a transaction identifier (generated when transaction prepare is called) that is present in the *SnapshotManager*'s *currentTransactions* data structure. A *TrxCommitError* message is returned to the actor if the message's transaction identifier is unknown, which would signify that the transaction identifier is invalid (not from a call to prepare) or that the transaction is aborted. If the transaction is found, we check if at least one message is sent in the transaction (Listing 7.3, line 7). If this is the case, we make sure that the commit version vector reflects this by incrementing the version vector. Then, the *SnapshotManager*'s commit method is called (Listing 7.3, line 8). This is when the transaction's snapshot, referenced by the transaction identifier, is moved into $LLSS$. A commit version vector is returned by the *SnapshotManager* after commit. We update the entry int the map containing known version vectors for our node, which triggers a potential $GSS$ update (Listing 7.3, line 10). Then, after the *SnapshotManager* reflects the latest change, we trigger replication (Listing 7.3, line 12) and a *SnapshotGossip* message is sent to every other known replicator actors. Finally, we clear traces of the transaction before sending a *TrxCommitSuccess* message back to the actor (Listing 7.3, lines 17 and 19).

### 7.4.2 Causal messages

We send causal messages inside a transaction using a dedicated *causalTell* method for messages that require causal delivery. A message using Akka's classic *tell* or *!* syntax is not guaranteed to be causally delivered. A message sent using our custom *causalTell* method is associated with the transaction's causal context (i.e., its version vector) and is causally delivered to the recipient actor. This ensures that our implementation does not interfere with existing inter-actor messaging capabilities.

To ensure atomicity, the transaction's updates and messages remain in a private buffer until the transaction commits; at this point they all become visible at once. If the transaction aborts, we delete the buffer. On commit, we send the buffered messages and updates to the replicator actor, which then forwards the messages to the node on which the destination actor is present.

On reception of a remote committed transaction, the replicator actor handles the update for shared objects before forwarding causal messages to the appropriate actor.

```
1  class PongActor(var system: ActorSystem) extends CausalActor {
2    override def receive: Receive = {
3      super.receive.orElse({
4        case msg: Ping => msg.replyTo ! Pong()
5        case msg: CausalPing =>
6          new Transaction(system, self, (ctx) => {
7              ctx.causalTell(Pong(), msg.replyTo)
8          }).commit()
9      })
10 }}
```

**Listing 7.4:** Example causal actor that responds to a Ping message with a Pong message. Both causal messages and non-causal messages are supported.

Actors that receive causal messages inherit the *CausalActor* class (Listing 7.4, line 1). This base class is responsible for delaying delivery of messages until the context is causally consistent (See 6.1.2). Listing 7.4 shows the definition a *PongActor* that responds to a *Ping* message (line 4) with a *Pong* message and *CausalPing* message (line 5) with a causally deliverable *Pong* message. Both causal and non-causal messages are supported.

A message, sent using the *causalTell* method, is wrapped inside a *CausalMessageWrapper* message type. This type is hidden from the user and only used by Akka to intercept causal messages. When sent between distant nodes, *CausalMessageWrapper* messages are serialized using Google Protocol Buffer [2].

```
1    def receive: Receive = {
2      case msg: CausalChange =>
3        lastSeenVersion = msg.versionVector
4        checkAndDeliverCausalMessages()
5        onCausalChange(lastSeenVersion)
6      case msg: CausalMessageWrapper =>
7        if (checkIfCausallyDeliverable(msg)) {
8          msg.messages.foreach(m => self.forward(m))
9          checkAndDeliverCausalMessages()
10       } else {
11         // buffer message and wait for causal context to be correct
12         buffer.enqueue(msg)
13 }}
```

**Listing 7.5:** Message reception in the CausalActor base class.

[2] https://protobuf.dev

Listing 7.5 shows how the *CausalActor* base class intercepts and delays a *CausalMessageWrapper* message. On reception of a *CausalMessageWrapper* $m$ (Listing 7.5, line 6), we check if the underlying message is causally deliverable (Algorithm 1, line 23). If $m$ is deliverable, we forward the wrapped message to the *self*, as we are the base class. After a successful delivery, we check if previously buffered message can be delivered. If $m$ is not causally deliverable, $m$ is queued for later delivery.

# Part III

Experimental Evaluation

# 8 Performance Evaluation

Our experimental evaluation address the following questions: What is the overhead of causal consistency? How does TTCC scale on a single node and multiple nodes?

In this chapter, we first explain our methodology and experimental setup in Section 8.1. Then, in Section 8.3 we introduce our additions to YCSB. Finally, in Section 8.4 and 8.5 we perform our evaluations.

## 8.1 Methodology and Experimental Setup

We implement our three protocols and a non-causal version in a transactional key-value store (KVS) that supports messages. We conduct performance benchmarks by using a modified Yahoo! Cloud Serving Benchmark (YCSB) [Coo+10] that includes transactions and messages (YCSB+MT). Performance experiments are run on multiple nodes, each equipped with two Intel Xeon E5-2690v3 clocked at 2.60 GHz with 192 GB of memory.

We run two experiments to verify the overhead of our unified causally consistency model and the scaling capability of our reference implementation. Experience 1 tests the scaling of our three protocols on a single compute node. The result of this experience defines the best configuration to reach the maximum throughput and latency on a single node. These configuration parameters are then used in experience 2 to scale our protocols on up to three compute nodes (Figure 8.1).

We introduce three new synthetic workloads that are directly inspired from the original YCSB workloads. Each workload tests a different read/write/message ratio.

To exclude any overhead due to marshalling and HTTP servers, we perform our measurements inside our KVS after the HTTP request is deserialized and before the HTTP response is returned to the client.

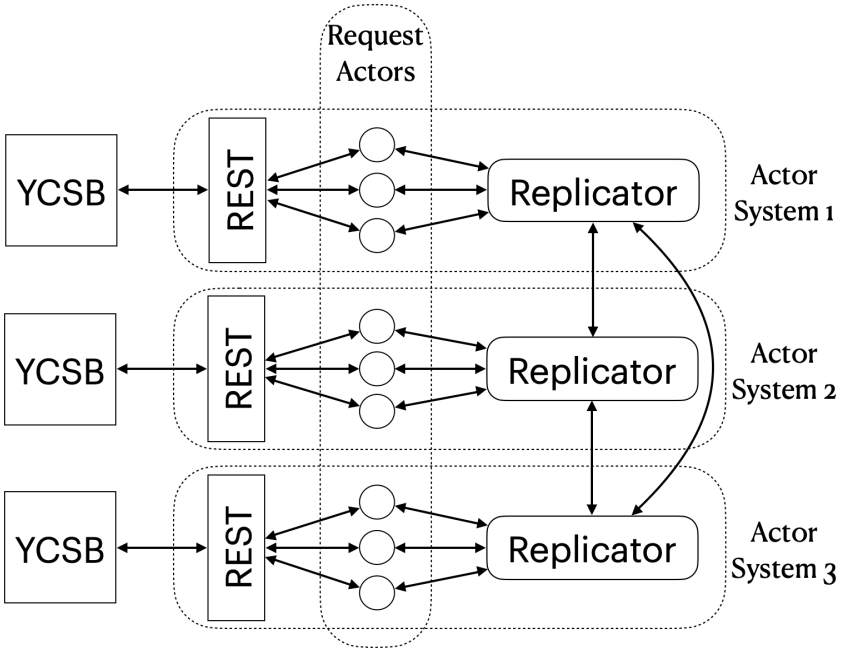



**Fig. 8.1.:** YCSB+MT experimental setup. Three actor systems are run, each with an associated YCSB+MT instance.

## 8.2 Key-Value Store

To evaluate our multiple protocols, we implement a replicated Key-Value Store (KVS). We use Akka's HTTP extension [1] to implement to expose our KVS using a REST interface that is accessible on each *host*. To best reply to users' requests, we manage a pool of *RequestActor*. Each HTTP request is associated to a *RequestActor* that is responsible for the completion of the HTTP request. After the request is done, the actor is recycled into the pool. The following paths are accessible by a REST client.

**Prepare** Accessible using an HTTP GET method on the following URL: *http://host/prepare*. A unique transaction identifier (*tid*) is returned. We use this identifier to identify transaction operations.

**Update** We expose a transaction update operation using an HTTP PUT method at the following URL: *http://host/tid/key*. The value is sent using the HTTP PUT *data* field.

**Read** A transaction read operation is accessible using an HTTP GET method on the following URL: *http://host/tid/key*.

[1] https://doc.akka.io/docs/akka-http/10.0

**Message** A message $m$ for transaction *tid* is sent to actor $dest$ using the following HTTP GET URL: *http://host/message/tid/*$dest$*/*$m$.

**Commit** A transaction commit is accessible using an HTTP GET method on the following URL: *http://host/commit/tid*.

## 8.3 YCSB+MT

The YCSB project regroups common sets of workloads for evaluating the performance of different key-value stores. However, it does not support transactions or messages. We extend YCSB version 0.17.0 to include a transactional interface and messages[2].

### 8.3.1 Transactions and messaging support

To test our reference KVS implementation, we require the support for messaging and transactions.

**Messages** To support messages, we add a new database operation type. Listing 8.1 and 8.2 show the modifications to the *DB.java* and *RestClient.java* files to include an additional *message* method. By default, this method returns a *Status.NOT_IMPLEMENTED* type, as implementation to support messages is not mandatory. To perform our experiments, we implement the *message* method in the *rest* binding to allow YCSB+MT to connect to our key-value store using a REST interface.

```
1 public Status message(String tid, String dest, String msg) {
2     return Status.NOT_IMPLEMENTED;
3 }
```

**Listing 8.1:** New message method in DB.java.

Our implementation sends a message to an already spawned destination actor ($dest$) that is identified by a unique identifier. By default, the message we send contains a string value that is composed of 100 random characters. This is configurable by setting the *fieldlength* and *fieldlengthdistribution* properties.

[2]Our fork is available at: https://github.com/benoitmartin88/YCSB

```
1   public Status message(String tid, String dest, String msg) {
2     Map<String, ByteIterator> result = new HashMap<>();
3     final String path = urlPrefix + "message/" + tid.concat("/") + dest.
      concat("/") + msg;
4     int responseCode;
5     try {
6       responseCode = httpGet(path, result);
7     } catch (Exception e) {
8       responseCode = handleExceptions(e, path, HttpMethod.GET);
9     }
10    return getStatus(responseCode);
11  }
```

**Listing 8.2:** New message method override in RestClient.java.

**Transactions** To support transactions, we create a new *TransactionalWorkload* class that extends the abstract *Workload* class (Listing 8.3). We override the *doTransaction* abstract method to implement transaction logic. The *doTransactionPrepare* method ((Listing 8.3, line 2)) sends a transaction prepare message and returns a unique transaction identifier that we use for all subsequent operations related to this transaction. We run a configurable number of operations per transaction that is configured in the workload's configuration file (*operationsPerTransaction* parameter in Listing 8.3, line 3). Then, we generate an operation (i.e., read, update and message) and call the appropriate handler method. Finally, we commit the transaction using the transaction identifier (Listing 8.3, line 16).

```
1  public boolean doTransaction(DB db, Object threadstate) {
2  String tid = doTransactionPrepare(db);
3  for(int i=0; i<operationsPerTransaction; ++i) {
4    switch (operationchooser.nextString()) {
5    case "READ":
6      doTransactionRead(db, tid);
7      break;
8    case "UPDATE":
9      doTransactionUpdate(db, tid);
10     break;
11   case "MESSAGE":
12     doTransactionMessage(db, tid);
13     break;
14   }
15 }
16 db.commit(tid);
17 return true;
18 }
```

**Listing 8.3:** doTransaction method from TransactionalWorkload.java.

**Properties file** Our additions to YCSB are configurable using YCSB's existing configuration mechanism. We add the following configuration properties:

- **operationspertransaction** We define the number of operations (read, update, message) that are run in a single transaction. By default, this value is set to 10 operations per transaction.
- **messageproportion** We set the proportion of messages that we send with respect to *operationspertransaction*. For instance, if *operationspertransaction* is set to the default value of 10, and *messageproportion* is set to 0.5 (50%), we send 5 messages.
- **messagedestinationcount** This parameter scales the size of the actor pool to whom we send messages to. By default, this value is set to 8 destination actors.

### 8.3.2 Workload

We provide three new workloads that are inspired from the original YCSB *workloada*, *workloadb* and *workloadc*. Each workload perform ten operations per transaction using the ratios described in Table 8.1.

| Workload | Read (%) | Write (%) | Message (%) |
|---|---|---|---|
| A | 33 | 33 | 33 |
| B | 90 | 5 | 5 |
| C | 5 | 90 | 5 |

**Tab. 8.1.:** YCSB+MT workload ratios for workloads a, b and c.

## 8.4 Experiment 1

### 8.4.1 Overview

In this experiment, we evaluate system performance when scaling TTCC on a *single* node. We increase the number of YCSB+MT client threads until performance saturation. Considering our hardware (two Intel Xeon e5-2690v3), we increase the number of client threads as follows: 1, 4, 8, 12 and 16.

In Akka, a message sent from one local actor to another is not serialized and deserialized. This optimization applies to all operations that we run in this experiment as we deploy the KVS on one node.

Moreover, metadata size varies depending on the protocol. Protocol 1, 2 and 3 uses a single unified version vector, with one entry per node, to track causal consistency for shared memory. In this experiment, the version vector will always contain only a single entry, which represents the mapping of a string (i.e., node identifier) to a 64-bit (8 bytes) integer. The node identifier depends on an actor system name that the user specifies. In this experiment, the version vector size is 83 bytes.

In Protocol 3, we use an additional matrix to track causality for messages. The size of the matrix depends on the number of concurrent clients. In this experiment, we scale up to 16 client threads. Each matrix entry maps two actor identifiers to a 64-bit integer.

### 8.4.2 Results

Our results show that our baseline performs better in all workloads for read, write and message operations.

**Sending a message using shared memory is inefficient** Overall, our results show that Protocol 2 (msg in shm) (sending a message using shared memory) always induces a large overhead compared to the other protocols, for read, write and message operation types. We explain this by the delay due to shared memory replication that is based on a broadcast mechanism. Effectively, each message is broadcast to every node and is delayed until the causal context (i.e. causal dependencies) is up-to-date.

**Data materialization is costly** Protocols 1, 2 and 3 materialize the requested data on every read operation, which causes a high response time overhead. Workload B (90% *read* operations), compared to Workload C (90% *write* operations), shows that read operations induces an overhead of 4.76% for Protocol 1, 6.67× for Protocol 2 and 3. Caching materialized data would greatly benefit read performance.

**Asynchronous write operations** Protocols 1, 2 and 3 write to an isolated snapshot, which results in efficient response time. For all three workloads, write operation perform similarly for Protocols 1 and 3. However, Protocol 2 always under-performs compared to the other protocols and our baseline. We explain this by the additional metadata that is sent to the local replicator.

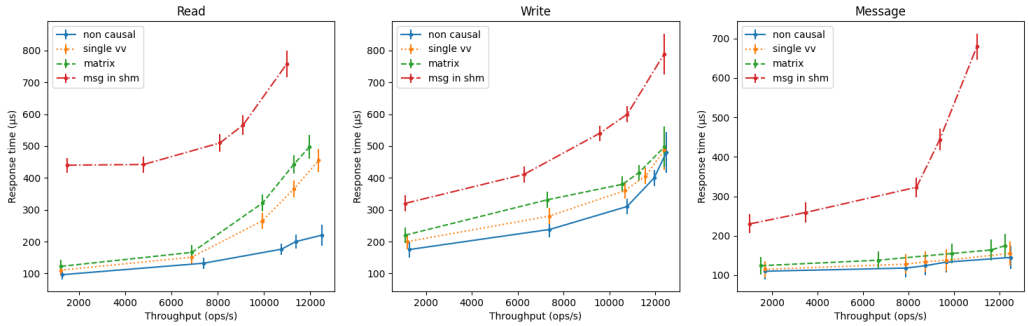



**Fig. 8.2.:** Transactional workload A (33R/33W/33M)

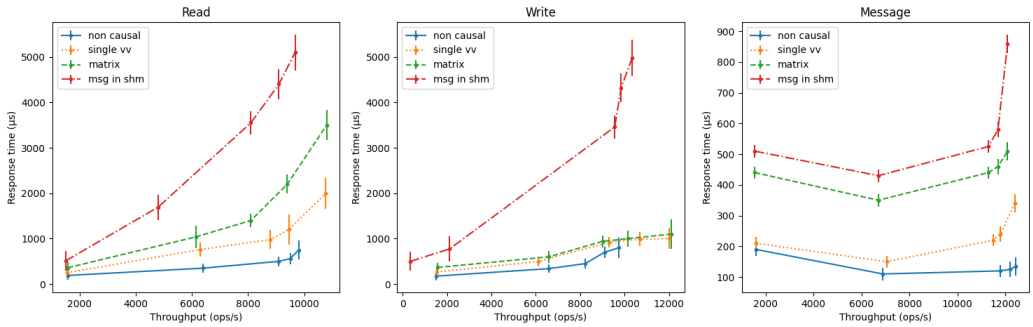



**Fig. 8.3.:** Transactional workload B (90R/5W/5M)

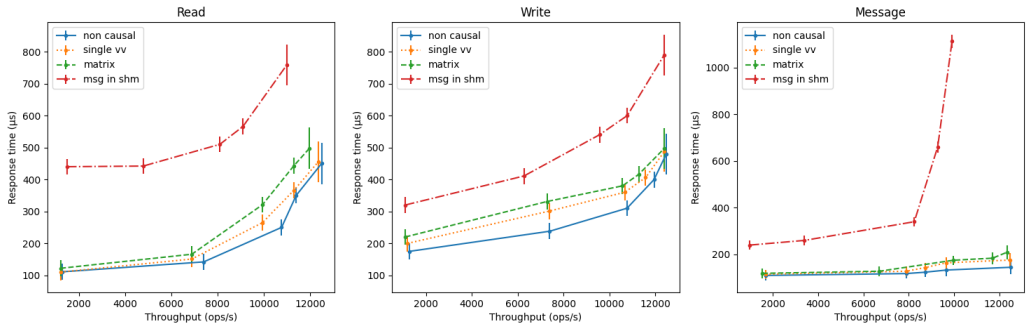



**Fig. 8.4.:** Transactional workload C (5R/90W/5M)

## 8.5 Experiment 2

### 8.5.1 Overview

In this experiment, we evaluate system performance when scaling TTCC on a *multiple* node. We use our results from Experiment 1 to set the YCSB+MT client threads to achieve the highest throughput on each instance. We select 16 threads per YCSB+MT instance and increase the number of YCSB+MT instances and KVS replicas from one to three.

Note that a message sent between JVMs is serialized using Google Protocol Buffers.

We exclude Protocol 2 from this experiment as it does not scale well on one node. We measure the overhead of Protocol 1 and 3 by comparing them with a non-causal baseline.

### 8.5.2 Results

Our results show that our baseline performs better in all workloads for read, write and message operations. Furthermore, Protocol 2, which uses an extra matrix, performs the worst (up to 5.6×, 5.64× and 6.56× for read, write and message operations respectively).

**Data materialization is costly** As with Experiment 1, Protocols 1 and 3 materialize the requested data on every read operation, which causes a high response time overhead. Workload B (90% *read* operations), compared to Workload A and C, where there are fewer read operations, shows a response time overhead of up to 4× for Protocol 1 and up to 7× for Protocol 3. Caching materialized data would greatly benefit read performance.

**Asynchronous write operations** As with Experiment 1, Protocols 1 and 3 write to an isolated snapshot, which results in efficient response time. For all three workloads, write operation perform similarly for Protocols 1 and our baseline. We explain this by the nature of writing to an isolated snapshot, which enables concurrent writes without synchronization.

However, Protocol 3 always under-performs compared to the other protocols and our baseline. We explain this by the additional metadata that is sent to the local replicator.

**Write operations impacts message delivery** Write operations impacts message delivery, as Figure 8.6 shows. Workload C (90% writes) shows a significant increase in message response time compared to workload A and C, where there are less write operations.

Each transaction commit generates an additional causal dependency that a message must wait for, which increases message delay.

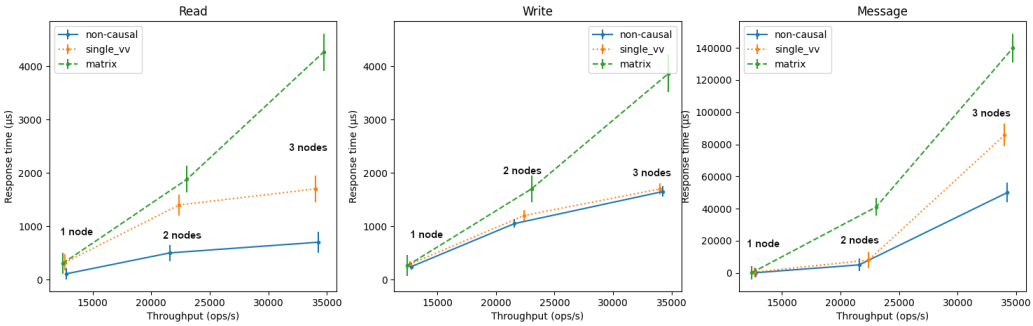



**Fig. 8.5.:** Transactional workload A (33R/33W/33M)

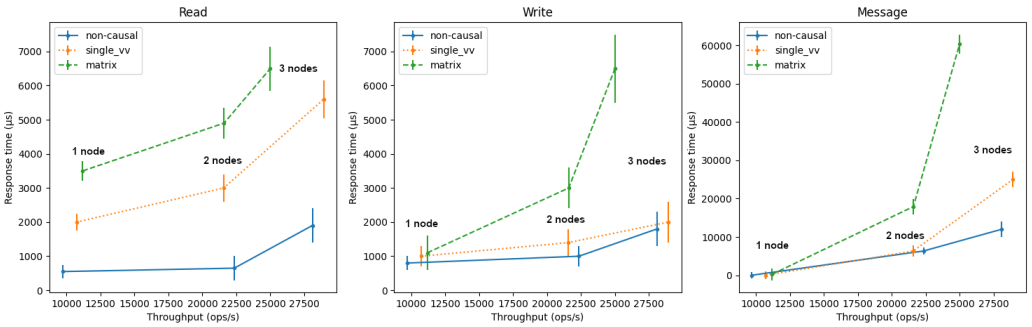



**Fig. 8.6.:** Transactional workload B (90R/5W/5M)

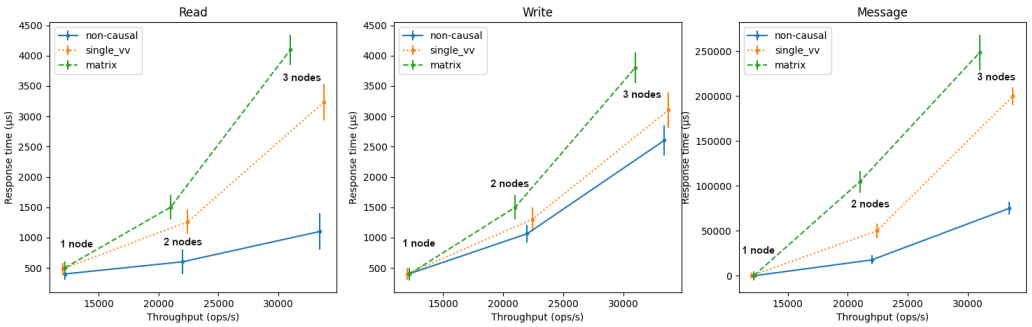



**Fig. 8.7.:** Transactional workload C (5R/90W/5M)

## 8.6 Summary

This evaluation chapter of the thesis presented an experimental evaluation for TTCC. We introduced our experimental setup and our experimental key-value store on which our two experiments run. Furthermore, we describe YCSB+MT, an extension to the original YCSB workload generator on which we add support for messages and transaction.

Our experimental evaluation shows that using a unique version vector (Protocol 1) is beneficial compared to using a causally consistent shared memory or an additional matrix to track causality.

In a single node setup, Protocol 1 always performs better than Protocol 2 and 3. Furthermore, Protocol 1 performs very similarly to a non-causal baseline due to the lack of replication and local operations.

The overhead of maintaining causality using a unique version vector ranges from 1.21× to 4.62× for read operations compared to a non-causal baseline and depending on the workload. For write operations, the overhead ranges from 1× to 1.40×; and for message operations, the overhead ranges from 1× to 4.58× depending on the workload.

Table 8.2 summarizes the response time overhead of Protocol 1 (our reference protocol that uses a single version vector) compared to a non-causal baseline.

Note that, in our approach, the requested data is materialized on every read operation. An improvement idea cache materialized views to improve response time.

| | | 1 node | | | 2 nodes | | | 3 nodes | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | R | W | M | R | W | M | R | W | M |
| Workload | A | 2.98× | 1.19× | 4.58× | 2.80× | 1.14× | 1.59× | 2.43× | 1.03× | 1.72× |
| | B | 3.64× | 1.25× | 2.52× | 4.62× | 1.40× | 1.00× | 2.95× | 1.11× | 2.08× |
| | C | 1.21× | 1.00× | 1.21× | 2.10× | 1.22× | 2.67× | 2.94× | 1.19× | 2.53× |

**Tab. 8.2.:** Table summarizing the overhead of maintaining causal consistency for Protocol 1 when scaling on 1, 2 and 3 nodes.

# Part IV

Discussion and Conclusion

# 9 Discussion

Many *stateful* serverless frameworks are based on the actor model. However, existing frameworks separate shared state storage from message-passing, which may lead to inconsistency and application crashes. TTCC extends the actor model to ensures that the message and memory view are mutually causally consistent. Serverless frameworks can benefit from TTCC to support causally consistent serverless functions and prevent inconsistencies that are caused by the separation of state storage and message-passing.

Several aspects in TTCC remain open for improvements and investigation.

**Read materialization cache** Currently, our reference implementation materializes the requested data on every read operation, which causes a high response time overhead. An possible improvement is to cache objects that are read. This would cause a computational overhead, but could result in improved response time.

**Implement TTCC on another framework** Our reference implementation extends the Akka actor framework, which has limited shared memory performance. It would be interesting to implement TTCC on a different framework, such as Cloudburst (serverless) or Orleans (actor).

**Additional experiments** Given more time, additional experimental evaluations would be interesting. TTCC could be compared to existing serverless cloud frameworks. Furthermore, a complex real-life use-case would be beneficial.

**Protocol 3 matrix size optimization** In Protocol 3, we use a matrix to track causal delivery of a message between two actors. The size of this matrix can be optimized as the diagonal of this matrix is unused. Furthermore, the matrix may be pruned for actor pairs that are not on the message's causal path.

**Dynamic function placement** Collocated functions benefit from locally shared actor state. It would be interesting to optimize function placement depending on their interaction graph.

# 10 Conclusion

Stateful serverless computing frameworks chain functions together using a message-based infrastructure and store their durable state in a separate database. This separation between storage and compute creates serious challenges that may lead to inconsistency and application crashes.

This thesis first explored the paradigms and challenges that are inherent to serverless computing. Then, we present the requirements to cleanly integrate message-passing with shared memory.

In the second part of the thesis, we presented TTCC, a transactional, causally consistent, unified model for message passing and shared memory. TTCC is compatible with actor-based frameworks and provides an intuitive memory model that ensures that multiple pieces of information remain *mutually* consistent, whether sent using messages or shared in a distributed memory. TTCC is asynchronous, preserves isolation, and ensures that the message and memory view are mutually causally consistent.

Finally, we presented our experimental results. They show that Protocol 1, which uses a single unified version vector, performs the best compared to the protocols 2 and 3. Protocol 1 performs with a 4.62× response time overhead compared to a non-causal baseline protocol. However, we discussed that optimizations are possible. Write operations add a 1.40× response time overhead, and causal delivery of messages add 4.58× to the response time.

# Bibliography


[Agh85] Gul Abdulnabi Agha. “ACTORS: A Model of Concurrent Computation in Distributed Systems”. In: (June 1, 1985). Accepted: 2004-10-20T20:10:20Z (cit. on pp. 17, 59).

[Akka] *Akka Actor Framework*. URL: https://akka.io/ (visited on Feb. 18, 2023) (cit. on pp. 18, 59).

[Akkb] *Akka Serverless*. URL: https://www.lightbend.com/akka-serverless (visited on Feb. 17, 2023) (cit. on pp. 18, 59).

[Akk+16] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, et al. “Cure: Strong Semantics Meets High Availability and Low Latency”. In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS). Nara, Japan: IEEE, June 2016, pp. 405–414 (cit. on pp. 2, 30, 41, 47).

[Ama] *Amazon Web Services (AWS)*. Amazon Web Services, Inc. URL: https://aws.amazon.com/ (visited on Feb. 17, 2023) (cit. on p. 9).

[Eva] *An Evaluation of Amazon’s Grid Computing Services: EC2, S3, and SQS*. URL: https://dash.harvard.edu/handle/1/24829568 (visited on Mar. 1, 2023) (cit. on p. 15).

[Apa] *Apache OpenWhisk is a serverless, open source cloud platform*. URL: https://openwhisk.apache.org/ (visited on Feb. 17, 2023) (cit. on p. 59).

[Awsa] *AWS Lambda - Amazon Web Services*. Amazon Web Services, Inc. URL: https://aws.amazon.com/lambda/ (visited on Feb. 17, 2023) (cit. on p. 9).

[Awsb] *AWS S3 large payloads*. URL: https://docs.aws.amazon.com/step-functions/latest/dg/avoid-exec-failures.html (visited on Feb. 17, 2023) (cit. on pp. 20, 33).

[Awsc] *AWS Step Functions*. Amazon Web Services, Inc. URL: https://aws.amazon.com/step-functions/ (visited on Feb. 17, 2023) (cit. on p. 15).

[Azua] *Azure Durable Entities*. Feb. 1, 2023. URL: https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities (visited on Feb. 17, 2023) (cit. on p. 19).

[Azub] *Azure Durable Functions*. Feb. 1, 2023. URL: https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview (visited on Feb. 17, 2023) (cit. on p. 16).

[Azuc] *Azure Functions – Serverless Functions in Computing | Microsoft Azure*. URL: https://azure.microsoft.com/en-us/products/functions (visited on Feb. 17, 2023) (cit. on p. 9).

[BP+22] Daniel Barcelona-Pons, Pierre Sutra, Marc Sánchez-Artigas, Gerard París, and Pedro García-López. "Stateful Serverless Computing with Crucial". In: *ACM Transactions on Software Engineering and Methodology* 31.3 (Mar. 7, 2022), 39:1–39:38 (cit. on p. 12).

[BG83] Philip A. Bernstein and Nathan Goodman. "Multiversion concurrency control - theory and algorithms". In: *ACM Transactions on Database Systems* 8.4 (Dec. 1, 1983), pp. 465–483 (cit. on p. 28).

[Bre00] Eric A. Brewer. "Towards robust distributed systems (abstract)". In: *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. PODC '00. Portland, Oregon, USA: Association for Computing Machinery, 2000, p. 7 (cit. on p. 27).

[Bur14] Sebastian Burckhardt. "Principles of Eventual Consistency". In: *Foundations and Trends® in Programming Languages* 1.1 (2014), pp. 1–150 (cit. on pp. 23, 28, 34).

[Bur+21] Sebastian Burckhardt, Chris Gillum, David Justo, et al. "Durable Functions: Semantics for Stateful Serverless". In: *Proceedings of the ACM on Programming Languages* 5 (OOPSLA Oct. 20, 2021), pp. 1–27 (cit. on pp. 13, 19).

[CBG15] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. "A Framework for Transactional Consistency Models with Atomic Visibility". In: (2015). In collab. with Marc Herbstritt, 14 pages (cit. on pp. 27, 34).

[CGR07] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. "Paxos made live: an engineering perspective". In: *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. PODC07: ACM Symposium on Principles of Distributed Computing 2007. Portland Oregon USA: ACM, Aug. 12, 2007, pp. 398–407 (cit. on p. 30).

[Clo] *Cloudflare Durable Objects · Cloudflare Workers docs*. Feb. 9, 2023. URL: https://developers.cloudflare.com/workers/learning/using-durable-objects/ (visited on Feb. 17, 2023) (cit. on p. 19).

[Coo+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM symposium on Cloud computing*. SoCC '10. New York, NY, USA: Association for Computing Machinery, June 10, 2010, pp. 143–154 (cit. on p. 69).

[DKVCDM16] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. "43 years of actors: a taxonomy of actor models and their key properties". In: *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. AGERE 2016. New York, NY, USA: Association for Computing Machinery, Oct. 30, 2016, pp. 31–40 (cit. on pp. 18, 25, 59).

[Akkc] *Distributed Data • Akka Documentation* (cit. on p. 59).

[FLP85] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. “Impossibility of distributed consensus with one faulty process”. In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382 (cit. on p. 27).

[GL+18] Pedro Garcia Lopez, Marc Sanchez-Artigas, Gerard Paris, et al. “Comparison of FaaS Orchestration Systems”. In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). Zurich: IEEE, Dec. 2018, pp. 148–153 (cit. on p. 15).

[Goo] *Google Cloud Functions*. Google Cloud. URL: `https://cloud.google.com/functions` (visited on Feb. 17, 2023) (cit. on p. 9).

[HBS21] Hassan B. Hassan, Saman A. Barakat, and Qusay I. Sarhan. “Survey on serverless computing”. In: *Journal of Cloud Computing* 10.1 (July 12, 2021), p. 39 (cit. on p. 10).

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. “A Universal Modular ACTOR Formalism for Artificial Intelligence”. In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI’73. event-place: Stanford, USA. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245 (cit. on p. 17).

[HW13] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns: designing, building, and deploying messaging solutions*. 17. print. The Addison-Wesley signature series. Boston Munich: Addison-Wesley, 2013. 683 pp. (cit. on p. 20).

[How] *How KV works · Cloudflare Workers docs*. Nov. 9, 2022. URL: `https://developers.cloudflare.com/workers/learning/how-kv-works/` (visited on Mar. 2, 2023) (cit. on p. 20).

[Ibm] *IBM Composer*. GitHub. URL: `https://github.com/ibm-functions/composer` (visited on Feb. 17, 2023) (cit. on p. 16).

[Jon+19] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, et al. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Tech. rep. UCB/EECS-2019-3. EECS Department, University of California, Berkeley, Feb. 2019 (cit. on p. 15).

[Lam78] Leslie Lamport. “Time, clocks, and the ordering of events in a distributed system”. In: *Communications of the ACM* 21.7 (July 1, 1978), pp. 558–565 (cit. on p. 24).

[LSM05] Paul J. Leach, Rich Salz, and Michael H. Mealling. *A Universally Unique IDentifier (UUID) URN Namespace*. Request for Comments RFC 4122. Internet Engineering Task Force, July 2005 (cit. on p. 44).

[Llo+11] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. “Don’t settle for eventual: scalable causal consistency for wide-area storage with COPS”. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP ’11. New York, NY, USA: Association for Computing Machinery, Oct. 23, 2011, pp. 401–416 (cit. on p. 2).

[MPS20] Benoît Martin, Laurent Prosperi, and Marc Shapiro. “An environment for composable distributed computing”. In: EuroDW 2020 - 14th EuroSys Doctoral Workshop. Apr. 27, 2020 (cit. on p. 4).

[MS22] Benoît Martin and Marc Shapiro. “Shared memory for the actor model”. In: *Conférence francophone d’informatique en Parallélisme, Architecture et Système (COMPAS)*. Amiens, France, Aug. 7, 2022, p. 9 (cit. on p. 4).

[Net] *Netflix & AWS Lambda Case Study*. Amazon Web Services, Inc. URL: `https://aws.amazon.com/solutions/case-studies/netflix-and-aws-lambda/` (visited on Mar. 1, 2023) (cit. on p. 12).

[OO14] Diego Ongaro and John Ousterhout. “In search of an understandable consensus algorithm”. In: *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*. USENIX ATC’14. USA: USENIX Association, June 19, 2014, pp. 305–320 (cit. on p. 30).

[Sha+11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. “Conflict-free Replicated Data Types”. In: *SSS 2011 - 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Ed. by Xavier Défago, Franck Petit, and Vincent Villain. Vol. 6976. Lecture Notes in Computer Science. Grenoble, France: Springer, Oct. 2011, pp. 386–400 (cit. on pp. 29, 30, 34).

[SS18] Marc Shapiro and Pierre Sutra. “Database Consistency Models”. In: *Encyclopedia of Big Data Technologies*. Ed. by Sherif Sakr and Albert Zomaya. Cham: Springer International Publishing, 2018, pp. 1–11 (cit. on pp. 27–29).

[Sre+20] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, et al. “Cloudburst: stateful functions-as-a-service”. In: *Proceedings of the VLDB Endowment* 13.12 (Aug. 2020), pp. 2438–2452 (cit. on p. 16).

[Sta] *Stateful Programming Models in Serverless Functions*. InfoQ. URL: `https://www.infoq.com/presentations/serverless-workflows-actors/` (visited on Mar. 2, 2023) (cit. on p. 14).

[VV16] Paolo Viotti and Marko Vukolić. *Consistency in Non-Transactional Distributed Storage Systems*. Apr. 12, 2016. arXiv: `1512.00168[cs]` (cit. on pp. 23, 34).

[War] Jesse Warden. *Large Step Function Data – Dealing With Eventual Consistency in S3 – Software, Fitness, and Gaming*. URL: `https://jessewarden.com/2020/09/large-step-function-data-dealing-with-eventual-consistency-in-s3.html` (visited on Feb. 17, 2023) (cit. on pp. 20, 21).

[Wu+18] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. “Anna: A KVS for Any Scale”. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018 IEEE 34th International Conference on Data Engineering (ICDE). ISSN: 2375-026X. Apr. 2018, pp. 401–412 (cit. on p. 16).

[Zaw+15] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, et al. *Write Fast, Read in the Past: Causal Consistency for Client-side Applications*. report. May 20, 2015 (cit. on pp. 2, 30).

# List of Figures

# List of Tables

# List of Listings