

Распределённые транзакции умерли,  
да здравствуют распределённые транзакции!



Поддержка транзакций в Orleans 2.0

Sergey Bykov  
Microsoft, @sergeybykov

@msftorleans  
<https://github.com/dotnet/orleans/>

# Обо мне



## В Microsoft с 2001 г.

- Host Integration Server
- BizTalk Server
- Windows Embedded
- Bing
- Research
- Halo
- Xbox

# План

- Основы
- Проблемы с распределёнными транзакциями
- Известные подходы
- Сложности для разработчиков на Orleans
- Поддержка транзакций в Orleans
- Как это всё работает
- В разработке
- Заключение

# Основы: канонический пример

Перевод \$100 со счёта А на счёт В

**A**tomicity      всё или ничего

**C**onsistency    ограничения: отрицательный баланс недопустим

**I**solation        tx2 не увидит лишние \$100 в В до завершения tx1

**D**urability       сохранность данных

# Реляционная БД всё обеспечит ... локально

Данные хранятся локально

Сеть не мешает

Отказы в основном коррелируют

Фокус на компромиссе между производительностью и изоляцией:

- Serializable

- Repeatable read

- Read committed

- Read uncommitted

- Snapshot isolation

- ...

# Распред. транзакции – совсем другая история

Конфигурация слишком сложна

Совместимость не 100%

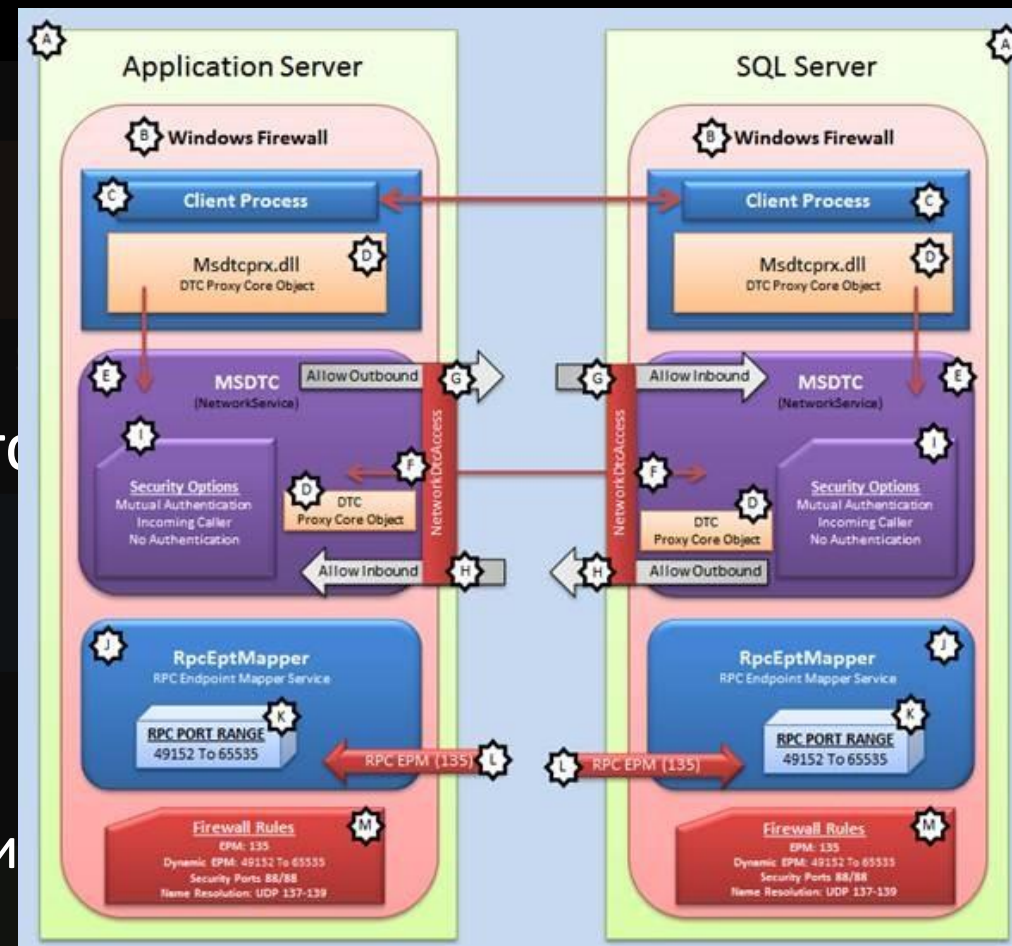
Современные NoSQL ДБ не интегрируются

Низкая производительность

Задержка – вызовы координатора

Проп. способность – блокировки и 1 координатор

Добиться надёжности нелегко



# Распределённые транзакции умерли

## Life beyond Distributed Transactions: an Apostate's Opinion

Position Paper

Pat Helland

Amazon.Com  
705 Fifth Ave South  
Seattle, WA 98104  
USA

[PHelland@Amazon.com](mailto:PHelland@Amazon.com)

The positions expressed in this paper are personal opinions and do not in any way reflect the positions of my employer Amazon.com.

### ABSTRACT

Many decades of work have been invested in the area of distributed transactions including protocols such as 2PC, Paxos, and various approaches to quorum. These protocols provide the application programmer a façade of global serializability. Personally, I have invested a non-trivial portion of my career as a strong advocate for the implementation and use of platforms providing guarantees of global serializability.

My experience over the last decade has led me to liken these platforms to the Maginot Line<sup>1</sup>. In general, application developers simply do not implement large scalable applications assuming distributed transactions. When they attempt to use distributed transactions, the projects founder because the performance costs and fragility make them impractical. Natural selection kicks in...

<sup>1</sup> The Maginot Line was a huge fortress that ran the length of the Franco-German border and was constructed at great expense between World War I and World War II. It successfully kept the German army from directly crossing the border between France and Germany. It was quickly bypassed by the Germans in 1940 who invaded through Belgium.

Instead, applications are built using different techniques which do not provide the same transactional guarantees but still meet the needs of their businesses.

This paper explores and names some of the practical approaches used in the implementations of large-scale mission-critical applications in a world which rejects distributed transactions. We discuss the management of fine-grained pieces of application data which may be repartitioned over time as the application grows. We also discuss the design patterns used in sending messages between these repartitionable pieces of data.

The reason for starting this discussion is to raise awareness of new patterns for two reasons. First, it is my belief that this awareness can ease the challenges of people hand-crafting very large scalable applications. Second, by observing the patterns, hopefully the industry can work towards the creation of platforms that make it easier to build these very large applications.

### 1. INTRODUCTION

Let's examine some goals for this paper, some assumptions that I am making for this discussion, and then some opinions derived from the assumptions. While I am keenly interested in high availability, this paper will ignore that issue and focus on scalability alone. In particular, we focus on the implications that fall out of assuming we cannot have large-scale distributed transactions.

#### Goals

This paper has three broad goals:

#### • Discuss Scalable Applications

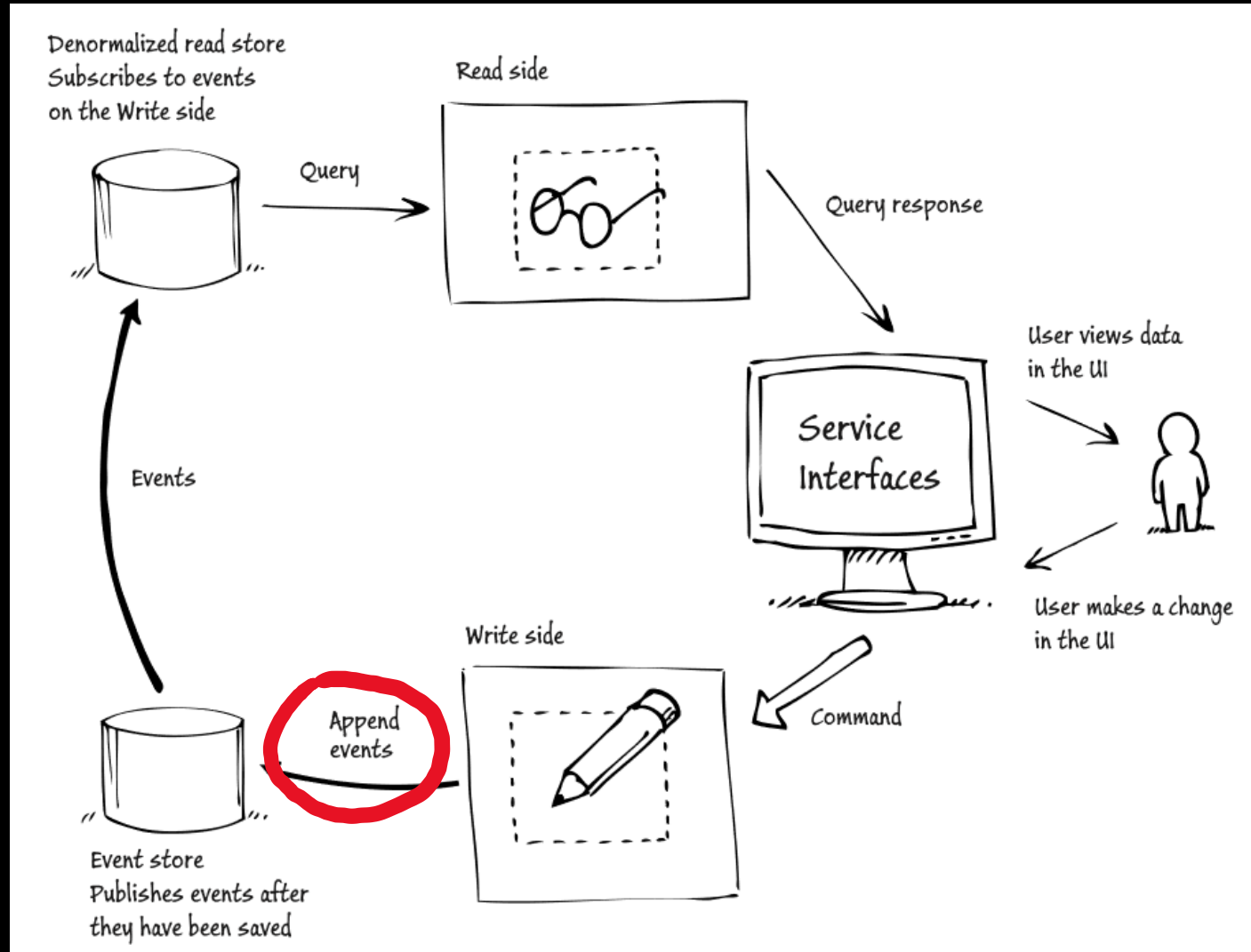
Many of the requirements for the design of scalable systems are understood implicitly by many application

Today, we see new design pressures foisted onto programmers that simply want to solve business problems. Their realities are taking them into a world of almost-infinite scaling and forcing them into design problems largely unrelated to the real business at hand.

Unfortunately, programmers striving to solve business goals like eCommerce, supply-chain-management, financial, and health-care applications increasingly need to think about scaling without distributed transactions. They do this because attempts to use distributed transactions are too fragile and perform poorly.

We are at a juncture where the patterns for building these applications can be seen but no one is yet applying these patterns consistently. This paper argues that these nascent patterns can be applied more consistently in the hand-crafted development of applications designed for almost-infinite scaling. Furthermore, in a few years we are likely to see the development of new middleware or platforms which provide automated management of these applications and eliminate the scaling challenges for applications developed within a stylized programming paradigm. This is strongly parallel to the emergence of TP-monitors in the 1970s.

# CQRS / Event Sourcing



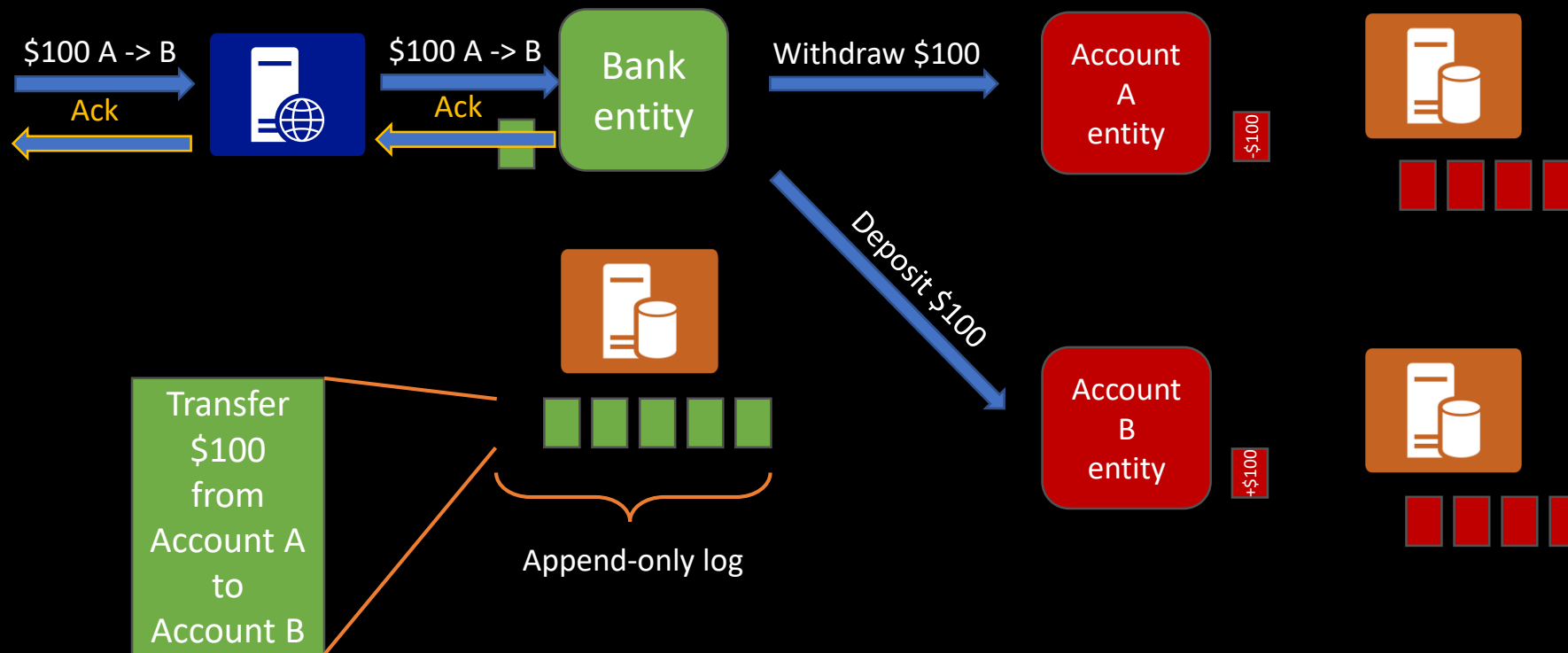
## Exploring CQRS and Event Sourcing

A journey into high scalability, availability, and maintainability with Windows Azure

Dominic Betts  
Julián Domínguez  
Grigori Melnik  
Fernando Simonazzi  
Mani Subramanian



# Перевод \$100 с Event Sourcing



# CQRS / Event Sourcing

No **A**tomicity

Eventual **C**onsistency

No **I**solation

**D**urable

Тем не менее хорошо подходит для многих сценариев.

Что делать, когда CQRS+ES недостаточно?

Strong Consistency Наносит Ответный Удар

или

New Kids on The Block

# Spanner

## Spanner: Google's Globally-Distributed Database

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford

Google, Inc.

### Abstract

Spanner is Google's scalable, multi-version, globally-distributed, and synchronously-replicated database. It is the first system to distribute data at global scale and support externally-consistent distributed transactions. This paper describes how Spanner is structured, its feature set, the rationale underlying various design decisions, and a novel time API that exposes clock uncertainty. This API and its implementation are critical to supporting external consistency and a variety of powerful features: non-blocking reads in the past, lock-free read-only transactions, and atomic schema changes, across all of Spanner.

### 1 Introduction

Spanner is a scalable, globally-distributed database designed, built, and deployed at Google. At the highest level of abstraction, it is a database that shards data across many sets of Paxos [21] state machines in datacenters spread all over the world. Replication is used for global availability and geographic locality; clients automatically failover between replicas. Spanner automatically reshards data across machines as the amount of data or the number of servers changes, and it automatically migrates data across machines (even across datacenters) to balance load and in response to failures. Spanner is designed to scale up to millions of machines across hundreds of datacenters and trillions of database rows.

Applications can use Spanner for high availability, even in the face of wide-area natural disasters, by replicating their data within or even across continents. Our initial customer was F1 [35], a rewrite of Google's advertising backend. F1 uses five replicas spread across the United States. Most other applications will probably replicate their data across 3 to 5 datacenters in one geographic region, but with relatively independent failure modes. That is, most applications will choose lower la-

tency over higher availability, as long as they can survive 1 or 2 datacenter failures.

Spanner's main focus is managing cross-datacenter replicated data, but we have also spent a great deal of time in designing and implementing important database features on top of our distributed-systems infrastructure. Even though many projects happily use Bigtable [9], we have also consistently received complaints from users that Bigtable can be difficult to use for some kinds of applications: those that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication. (Similar claims have been made by other authors [37].) Many applications at Google have chosen to use Megastore [5] because of its semi-relational data model and support for synchronous replication, despite its relatively poor write throughput. As a consequence, Spanner has evolved from a Bigtable-like versioned key-value store into a temporal multi-version database. Data is stored in schematized semi-relational tables; data is versioned, and each version is automatically timestamped with its commit time; old versions of data are subject to configurable garbage-collection policies; and applications can read data at old timestamps. Spanner supports general-purpose transactions, and provides a SQL-based query language.

As a globally-distributed database, Spanner provides several interesting features. First, the replication configurations for data can be dynamically controlled at a fine grain by applications. Applications can specify constraints to control which datacenters contain which data, how far data is from its users (to control read latency), how far replicas are from each other (to control write latency), and how many replicas are maintained (to control durability, availability, and read performance). Data can also be dynamically and transparently moved between datacenters by the system to balance resource usage across datacenters. Second, Spanner has two features that are difficult to implement in a distributed database: it

Spanner's main focus is managing cross-datacenter replicated data, but we have also spent a great deal of time in designing and implementing important database features on top of our distributed-systems infrastructure. Even though many projects happily use Bigtable [9], we have also consistently received complaints from users that Bigtable can be difficult to use for some kinds of applications: those that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication. (Similar claims have been made by other authors [37].) Many applications at Google have chosen to use Megastore [5] because of its semi-relational data model and support for synchronous replication, despite its relatively poor write throughput. As a consequence, Spanner has evolved from a Bigtable-like versioned key-value store into a temporal multi-version database. Data is stored in schematized semi-relational tables; data is versioned, and each version is automatically timestamped with its commit time; old versions of data are subject to configurable garbage-collection policies; and applications can read data at old timestamps. Spanner supports general-purpose transactions, and provides a SQL-based query language.

# Cloud Spanner

	CLOUD SPANNER	TRADITIONAL RELATIONAL	TRADITIONAL NON-RELATIONAL
Schema	✓ Yes	✓ Yes	✗ No
SQL	✓ Yes	✓ Yes	✗ No
Consistency	✓ Strong	✓ Strong	✗ Eventual
Availability	✓ High	✗ Failover	✓ High
Scalability	✓ Horizontal	✗ Vertical	✓ Horizontal
Replication	✓ Automatic	🔄 Configurable	🔄 Configurable

Spanner reasonably claims to be an “effectively CA” system despite operating over a wide area, as it is always consistent and achieves greater than 5 9s availability. As with Chubby, this combination is possible in practice if you control the whole network, which is rare over the wide area. Even then, it requires significant redundancy of network paths, architectural planning to manage correlated failures, and very careful operations, especially for upgrades. Even then outages will occur, in which case Spanner chooses consistency over availability.

# Cosmos DB - globally distributed, multi-model DB service

Replicate data globally  
andri

Save


Discard

Manual Failover

Failover Priorities


Click on a location to add or remove regions from your Azure Cosmos DB account.

\* Each region is billable based on the throughput and storage for the account. [Learn more](#)



Failover Priorities

Drag-and-drop read regions items to reorder the failover priorities.

Tip: Drag  on the left of the hovered row to reorder the list.


























WRITE REGION

Central US

READ REGIONS	PRIORITIES
Australia East	1
Brazil South	2
Canada Central	3
Canada East	4
Central India	5
East Asia	6
East US	7

OK

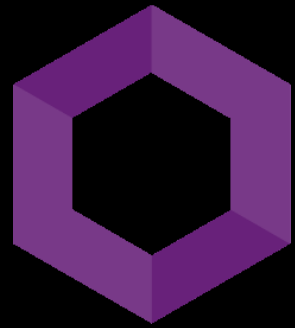
# Cosmos DB: Consistency Options

Strong	Bounded-stateless	Session	Consistent prefix	Eventual
				
Data consistency				
				
App availability				
				
Latency				
				
Throughput				
				

# Снова единая база данных!







# Microsoft Orleans

# Orleans: Фреймворк для Облака\*

## Модель программ-я

Для всех инженеров

Простая, но мощная

3x –10x меньше кода

## Масштабируемость

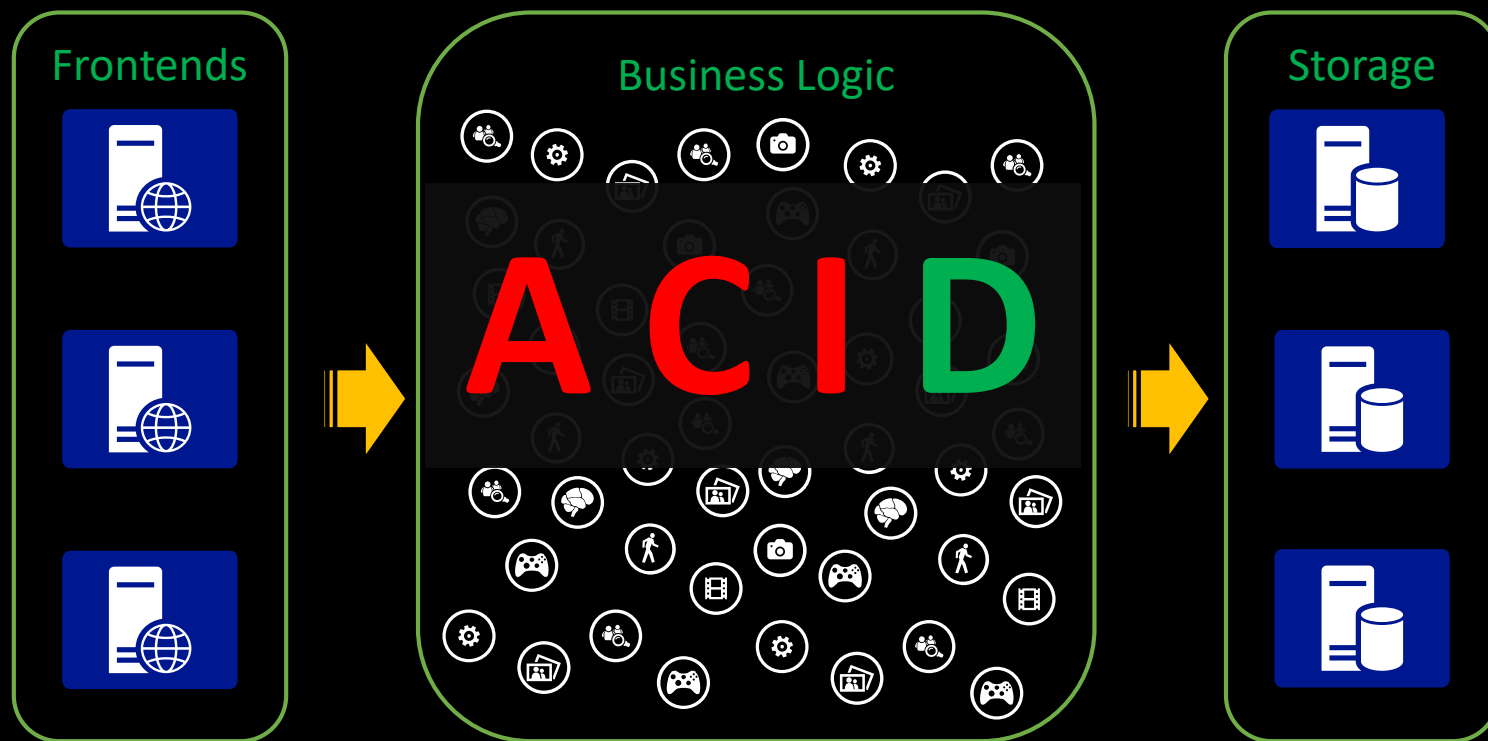
Нет узких мест

Нет единой точки отказа

Проверенные временем  
приёмы и алгоритмы

\*Помимо облачных сервисов, Orleans хорошо подходит для построения самых разнообразных распределённых приложений

# Модель программирования Orleans



Грейн – объект со стабильным ID

- Живёт вечно, в виртуальном смысле
- Инкапсулирует своё состояние
- Нет прямого доступа извне
- Только посылка сообщений

Грейн управляет своим состоянием

- Поддержка разных хранилищ данных

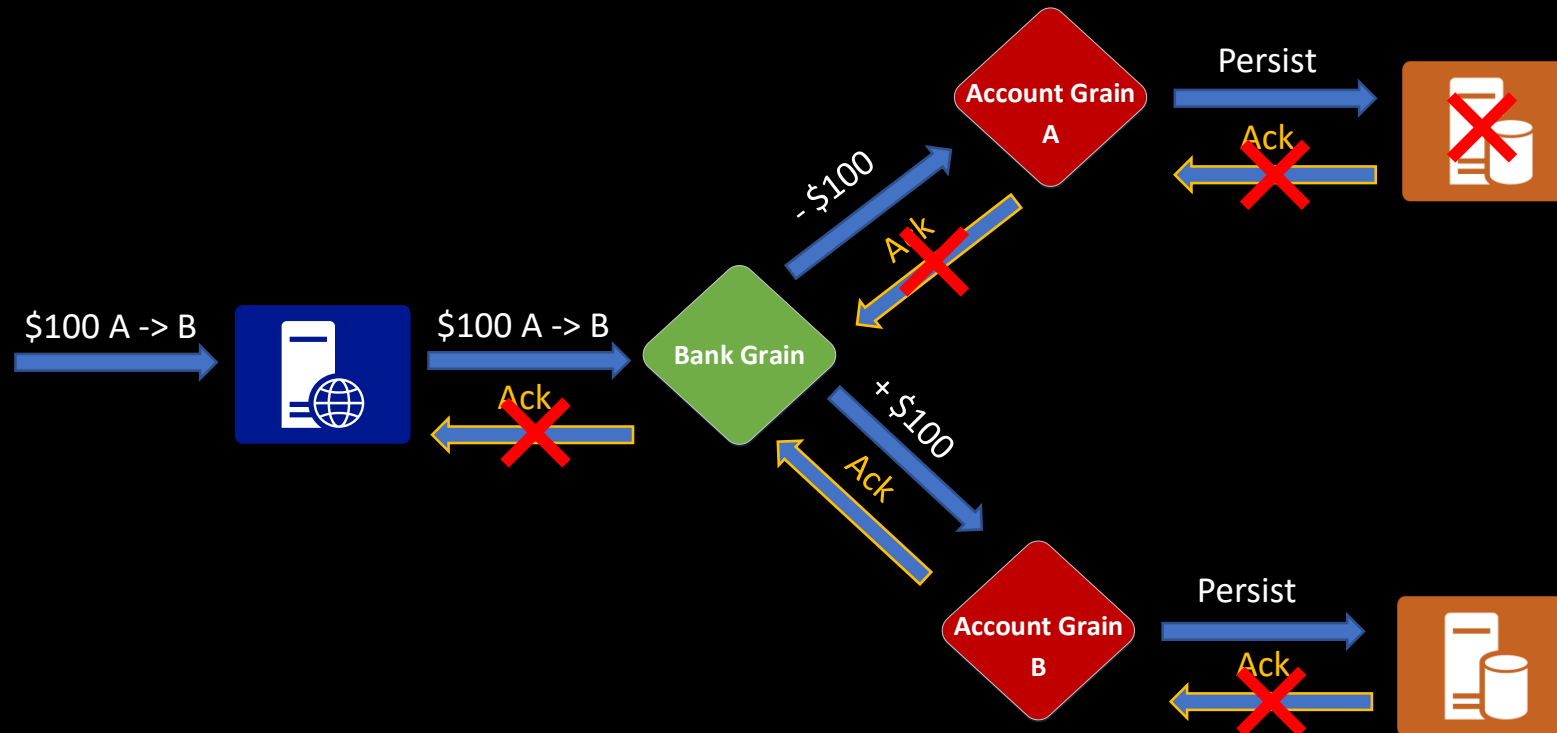
Отлично подходит для Event Sourcing

- Изолированный журнал изменений

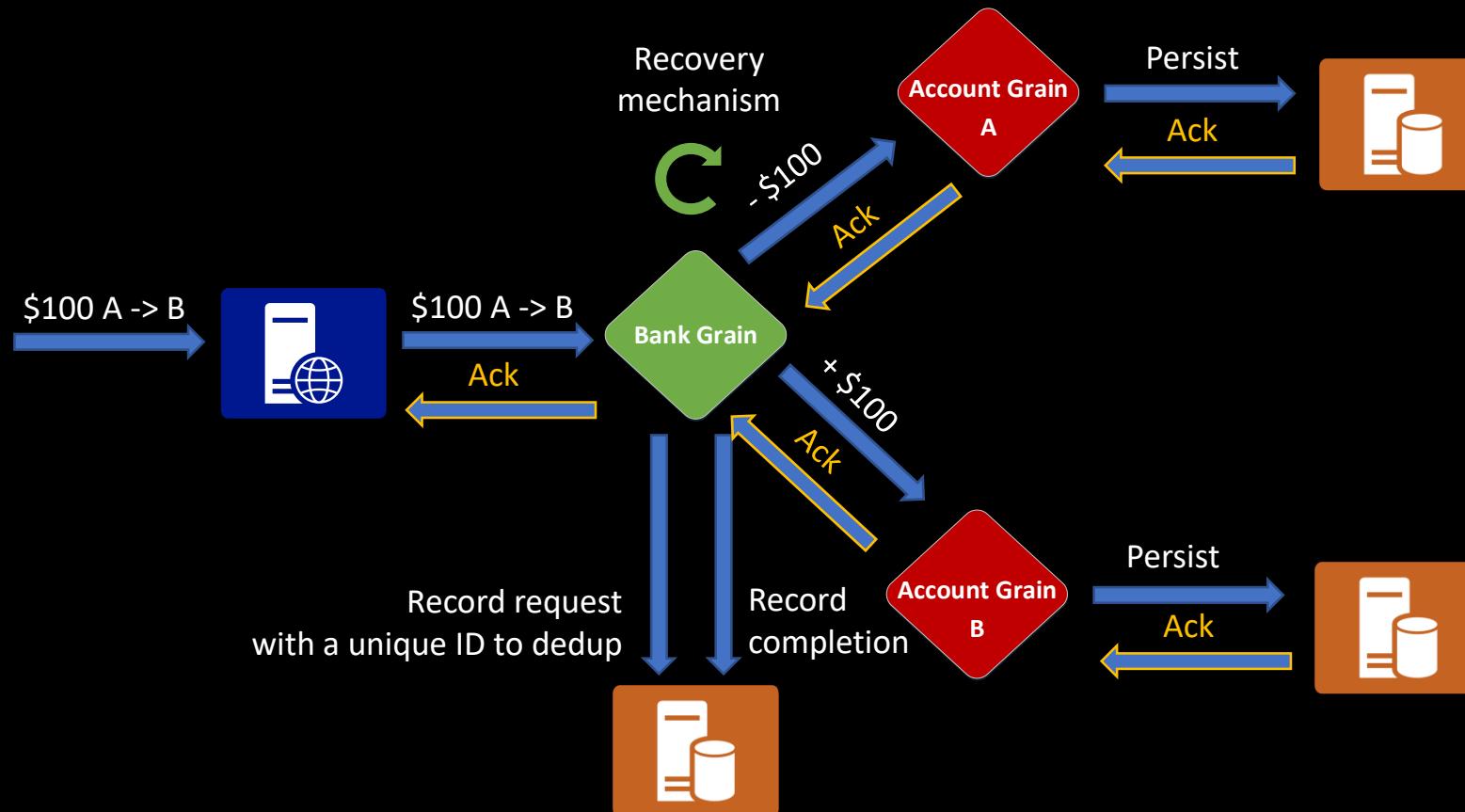
Отличная масштабируемость

Нет механизмов координации

# Перевод \$100 в Orleans



# Что нам придется делать



# Наблюдение от Martin Kleppmann

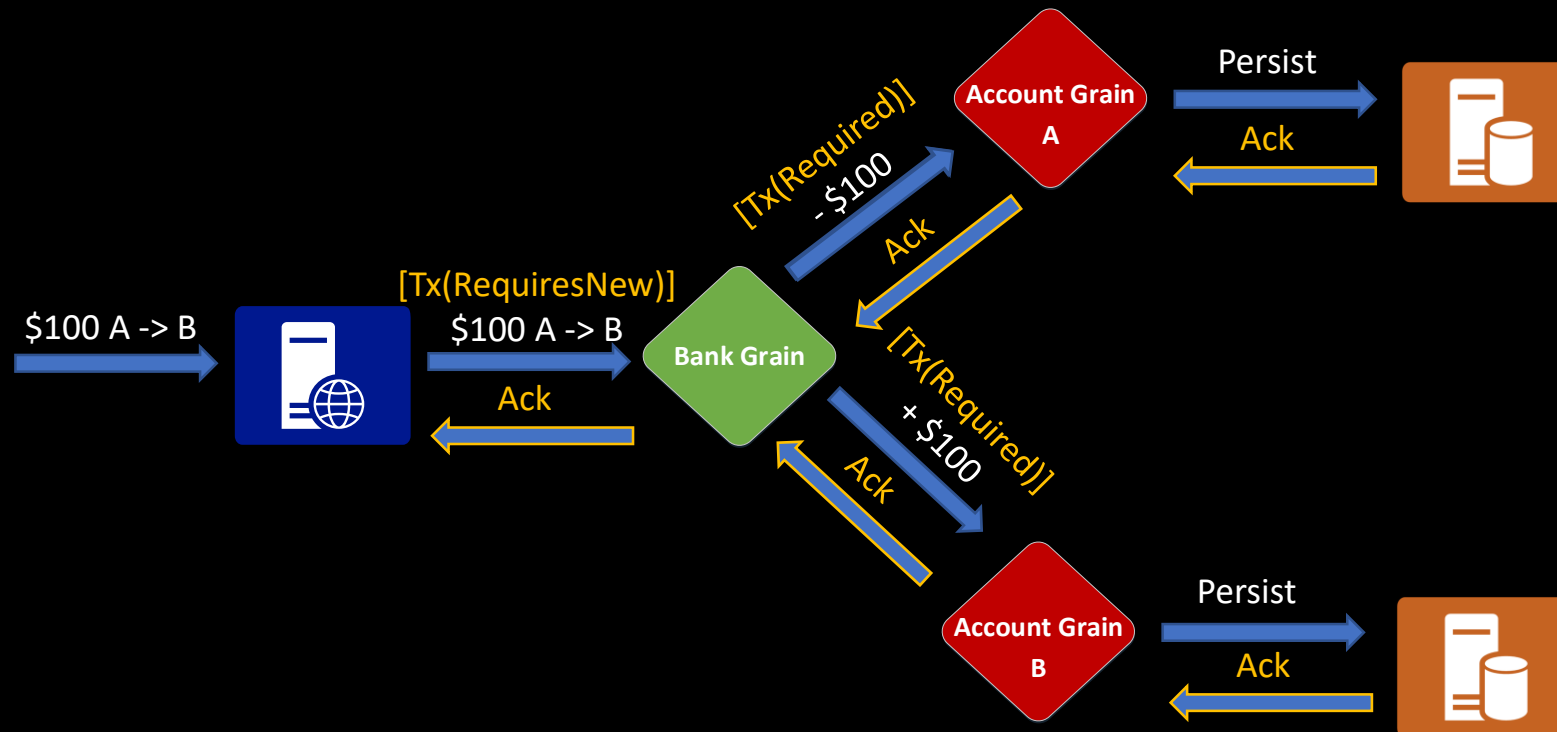


Sept 25-26, 2015

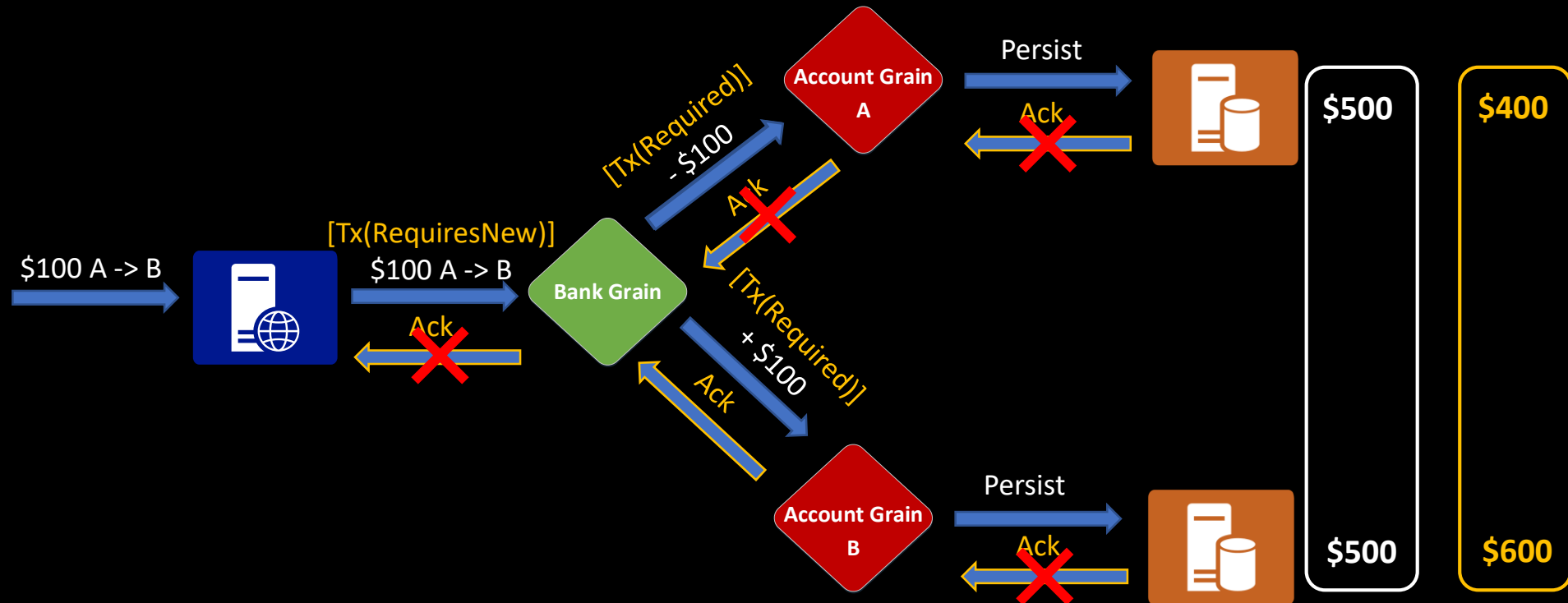
[thestrangeloop.com](http://thestrangeloop.com)

Every sufficiently large deployment of  
**microservices**  
contains an ad-hoc, informally-  
specified, bug-ridden, slow  
implementation of half of  
**transactions**

# Операция перевода в идеале



# Хотелось бы иметь гарантии ACID





Транзакции Orleans предоставляют ACID

# Интерфейс грейна банка

```
public interface IBankGrain : IGrainWithIntegerKey
{
    [Transaction(TransactionOption.RequiresNew)]
    Task Transfer(Guid fromAccount, Guid toAccount, uint amount);
}
```

# Интерфейс грейна банковского счёта

```
public interface IAccountGrain : IGrainWithGuidKey
{
    [Transaction(TransactionOption.Required)]
    Task Withdraw(uint amount);

    [Transaction(TransactionOption.Required)]
    Task Deposit(uint amount);

    [Transaction(TransactionOption.Required)]
    Task<uint> GetBalance();
}
```

# Операция перевода

```
public class BankGrain : Grain, IBankGrain
{
    Task Transfer(Guid fromAccount, Guid toAccount, uint amount)
    {
        var from = GrainFactory.GetGrain<IAccountGrain>(fromAccount);
        var to = GrainFactory.GetGrain<IAccountGrain>(toAccount);

        return Task.WhenAll(
            from.Withdraw(amount),
            to.Deposit(amount));
    }
}
```

# Account Grain: Balance State Facet

```
public class Balance
{
    public uint Value { get; set; } = 1000;
}

public class AccountGrain : Grain, IAccountGrain
{
    private readonly ITransactionalState<Balance> balance;

    public AccountGrain(
        [TransactionalState("balance")] ITransactionalState<Balance> balance)
    {
        this.balance = balance ?? throw new ArgumentNullException(nameof(balance));
    }
}
```

# Account Grain: операции

```
Task IAccountGrain.Deposit(uint amount)
{
    this.balance.State.Value += amount;
    this.balance.Save();
    return Task.CompletedTask;
}
```

```
async Task<uint> IAccountGrain.GetBalance()
{
    return this.balance.State.Value;
}
```

```
Task IAccountGrain.Withdraw(uint amount)
{
    this.balance.State.Value -= amount;
    this.balance.Save();
    return Task.CompletedTask;
}
```

# Это в принципе всё, что требуется!

Осталось только добавить конфигурацию

```
var builder = new SiloHostBuilder()
    .UseLocalhostClustering()
...
    .AddMemoryGrainStorageAsDefault()
    .UseInClusterTransactionManager()
    .UseInMemoryTransactionLog()
    .UseTransactionalState();

var host = builder.Build();
await host.StartAsync();
```

Как это работает?



# Архитектура

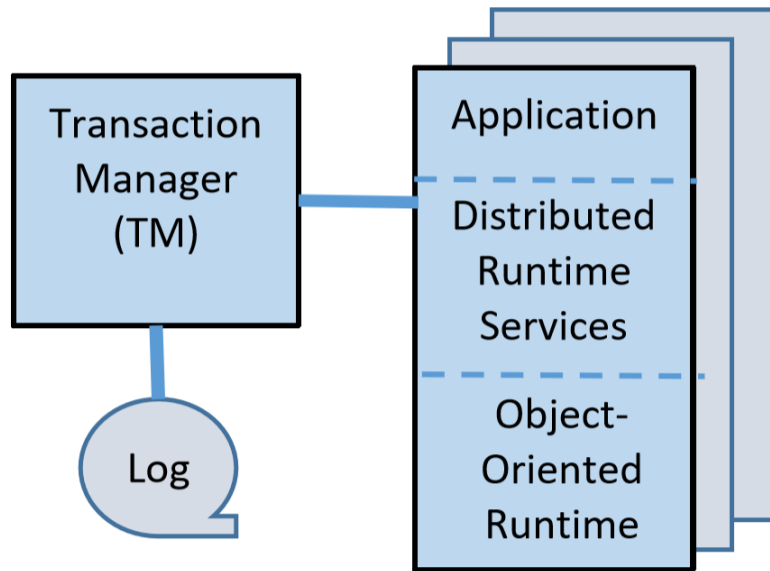


Figure 1 System Architecture



## Transactions for Distributed Actors in the Cloud

Tamer Eldeeb  
Columbia University  
tamer.eldeeb@columbia.edu

Philip A. Bernstein  
Microsoft Research  
philbe@microsoft.com

**Abstract**—Many cloud-service applications have a middle tier organized as micro-services or actors. Such applications have small objects that are spread over many servers and communicate via message passing. Transactions in such an application are necessarily distributed. However, distributed transactions usually perform poorly in this environment, primarily because locks must be held until after the forced-writes of two-phase commit, which are slow in cloud storage systems. We present a new transaction protocol that avoids this blocking by releasing all of a transaction's locks during phase one of two-phase commit, and by tracking commit dependencies to implement cascading abort. While a transaction  $T$  runs phase one, later conflicting transactions batch their updates. After  $T$  is prepared, the delayed batch can prepare, enabling a distributed form of group commit. We describe how to implement our protocol in an object-oriented runtime such as JVM or .NET. The performance measurements of our implementation in the Orleans actor framework show throughput up to 20x that of two-phase locking and two-phase commit.

**Keywords**—database system, transaction, two-phase locking

### I. INTRODUCTION

Many cloud services have a 3-tier architecture with a stateless front-end, a stateful middle-tier that implements business logic, and a storage layer. The stateful middle-tier is needed due to its heavy CPU and memory requirements, which make it uneconomical to embed as stored procedures in the storage layer. Today, the middle-tier is frequently organized as a set of micro-services. This is driven in part by the popularity of container technology like Docker [26], which is offered by most major cloud providers [5][29][32][39]. An application built as micro-services is composed of small, independent services that are versioned, deployed, upgraded and scaled separately. Services communicate via well-defined APIs (usually REST) and do not have access to shared data.

A similar model is actors, which are also popular in building middle-tier cloud applications, especially interactive ones such as games, social networks, Internet of Things, and telemetry [6][10]. Such applications are made of many actors, which are objects that do not share memory and interact via asynchronous messages. Actors are an extreme case of very small micro-services. In what follows, we use the more familiar, generic term “object” unless talking about a specific actor system.

It is often necessary to perform an operation that spans multiple objects with strong consistency and fault tolerance guarantees. Since objects are isolated from each other, multi-actor operations require a transaction mechanism [14]. Since cloud services are distributed across many servers for scalability and availability, most transactions that access multiple objects

will access multiple servers. Such transactions must be distributed. The well-known challenges of scaling distributed transactions, e.g. in [31][45], has led many cloud systems to offer limited transaction support (e.g., within a shard or with weak isolation) or no support at all. This puts the burden on developers to use ad-hoc methods to obtain cross-object consistency, which is hard to do well.

To understand the scalability challenge, consider the most popular distributed transaction protocol, two-phase locking (2PL) for isolation with two-phase commit (2PC) for atomicity. To ensure isolation, a transaction holds locks until it finishes executing. Each object in its readset can release read-locks when it receives a prepare-request in phase-one of 2PC. However, each object in its writeset must hold its write locks until it receives a commit request in phase-two of 2PC. This technique, called **strict 2PL**, ensures that a transaction that aborts before phase-two can undo its writes easily, without cascading aborts.

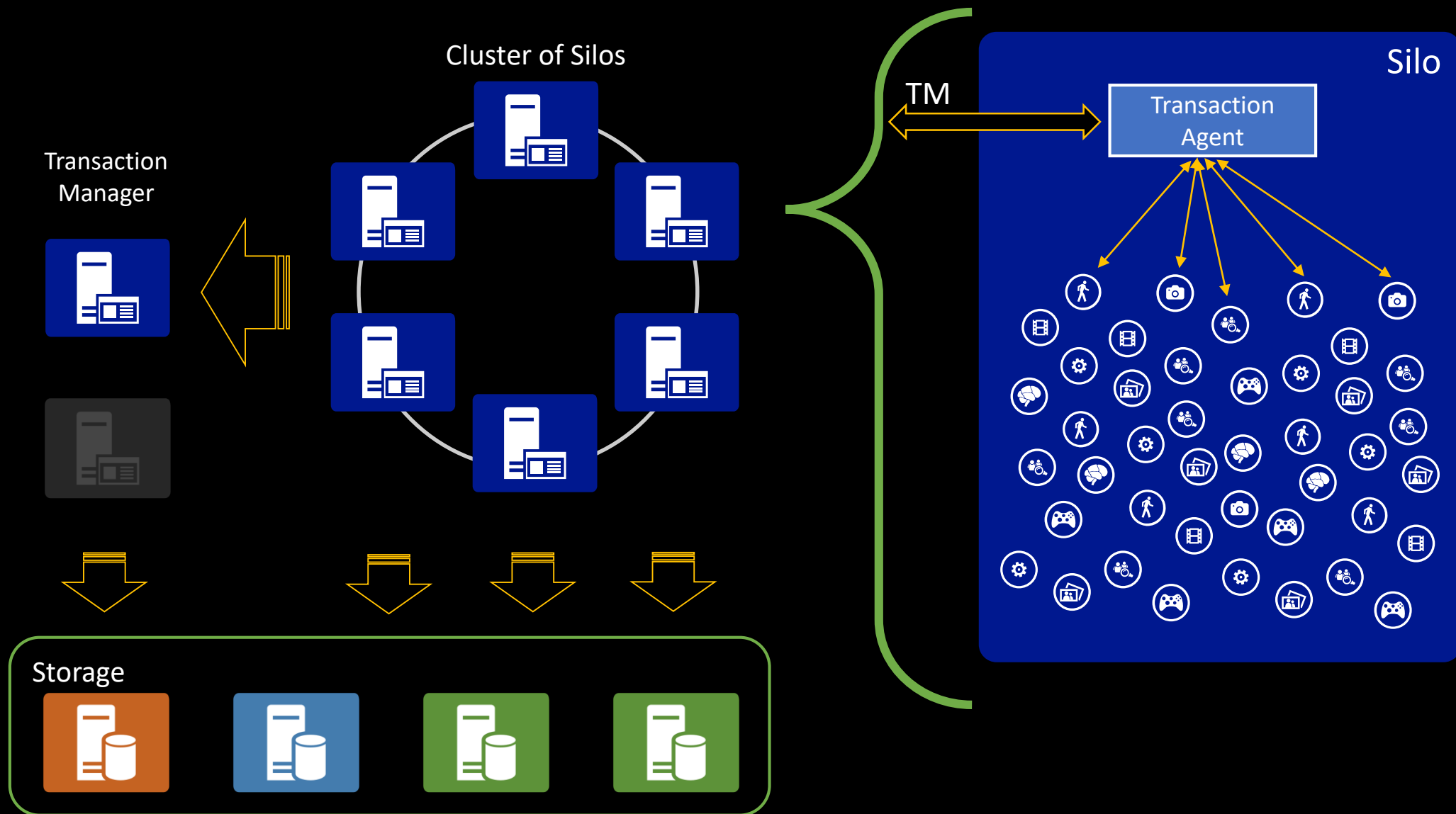
Two-phase commit does two synchronous writes to storage before phase-two, one to prepare and one to commit. Thus, a data item participating in a transaction needs to be locked and inaccessible for two round-trips to storage. This greatly limits throughput on write-hot data, which is a bottleneck in most transaction applications. For example, a typical cloud storage system has high latency due to networking, disk, and replication overhead. In our runs, a write to cloud storage takes on average ~20 milliseconds (ms) within a single datacenter. Thus, a transaction holds locks for ~40 ms. This limits throughput to 25 transactions/second (tps) on write-hot data, even though distributed transaction execution typically takes a few milliseconds. Low-latency SSD-based cloud storage is faster [4][28], but it still incurs double-digit millisecond 2PC latencies, plus higher cost.

This problem of lock-holding time also applies to system that use optimistic concurrency control (OCC). Like strict 2PL, an OCC validator needs to set write locks on a transaction  $T$ 's writeset before validating  $T$  and hold those locks until  $T$  is committed, to ensure  $T$  can be aborted without cascading aborts.

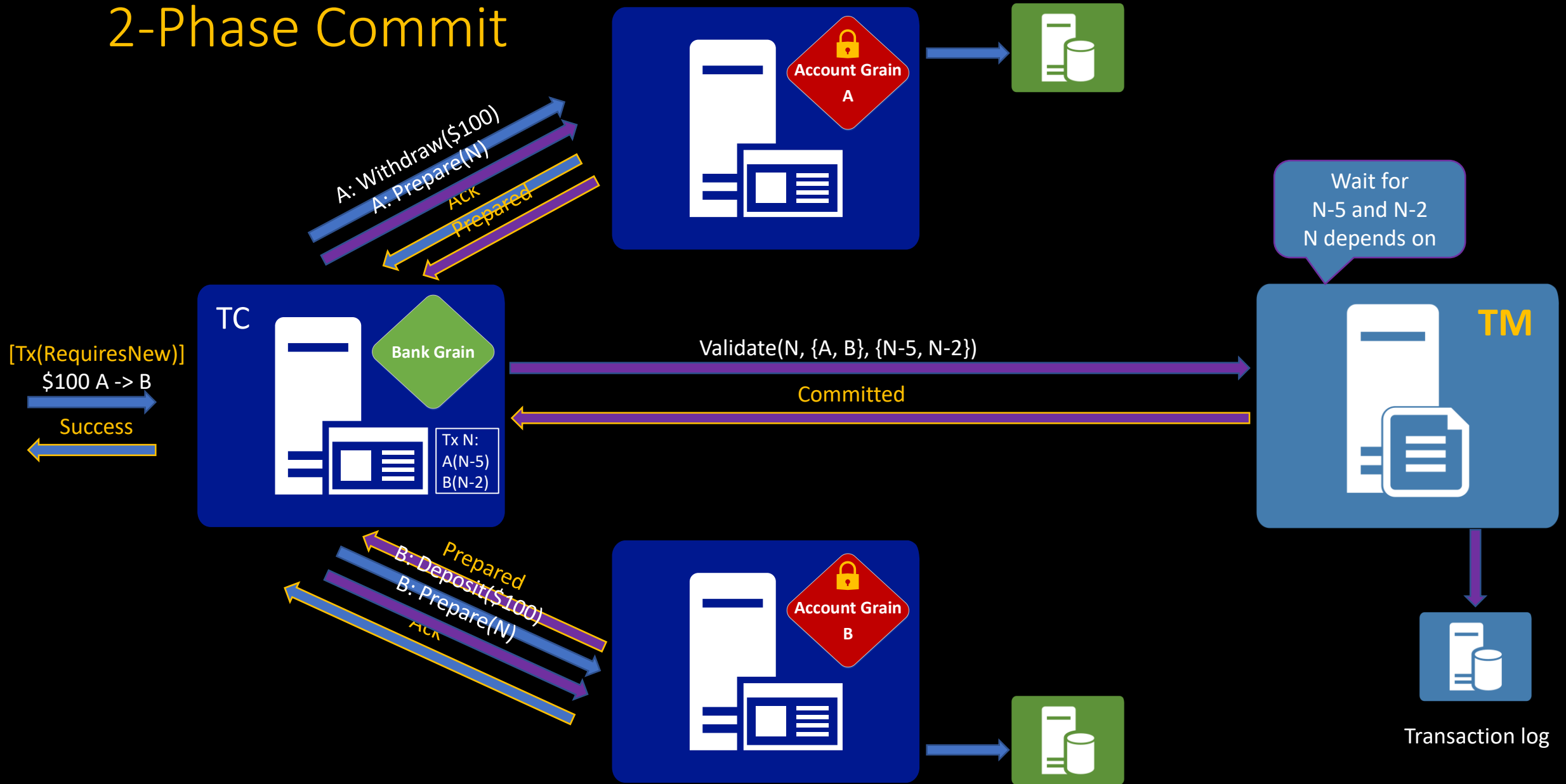
In this paper we present a novel distributed transaction system that avoids these problems by allowing a transaction to release locks early, during phase-one of the 2PC protocol. After a transaction  $T$  finishes executing, it will not acquire more locks, so holding locks after this point has no value from a 2PL perspective. But if  $T$  releases write locks before it commits, subsequent transactions can read “dirty” data that will be invalid if  $T$  aborts. To avoid this inconsistency, a transaction keeps track of its dependencies on uncommitted transactions. A central validator service, the Transaction Manager, makes sure that  $T$  commits only after all of  $T$ 's dependent transactions commit. If one of  $T$ 's dependent transactions aborts, then  $T$  will abort too.

October 31, 2016

# Архитектура

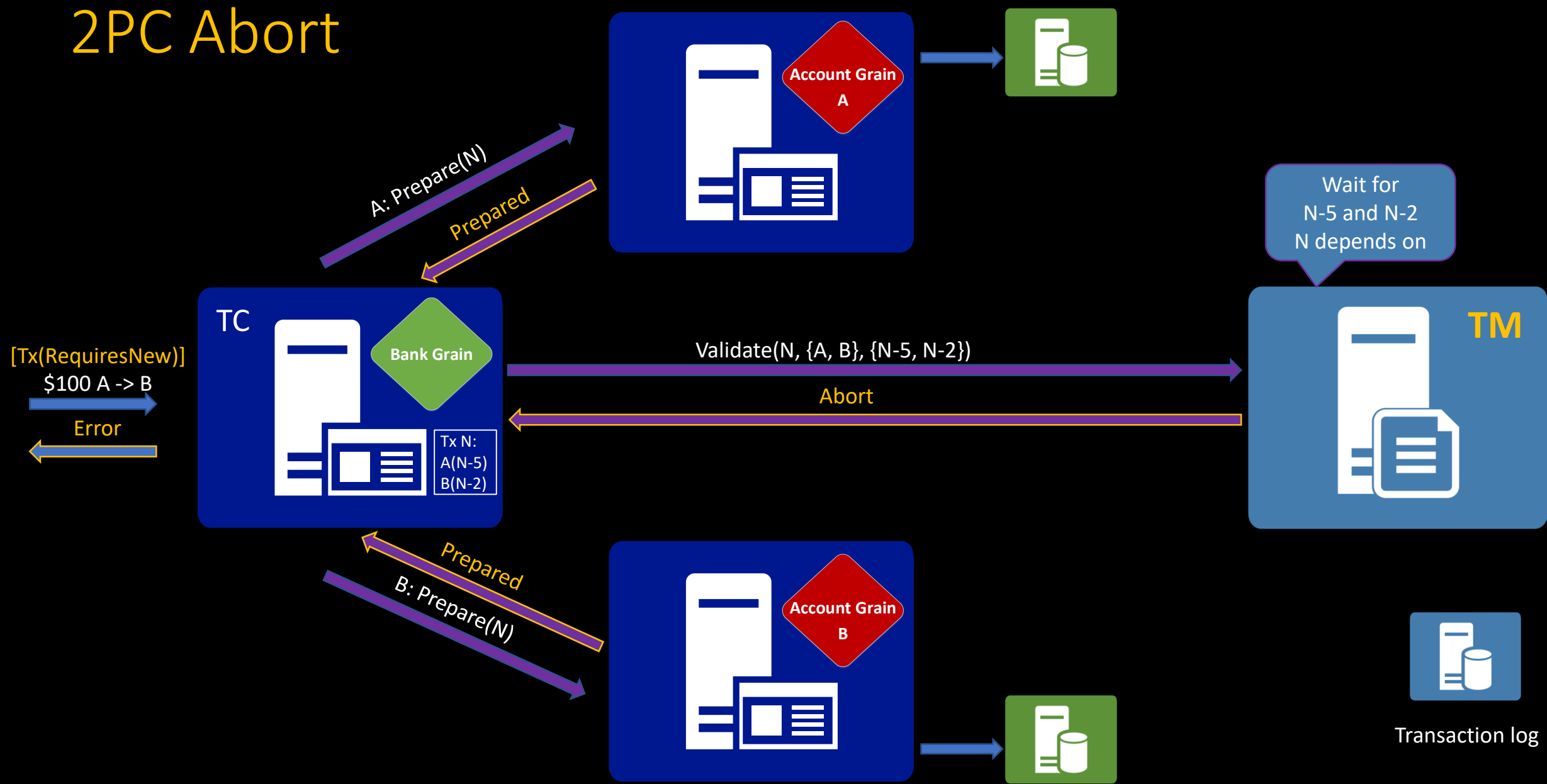


# 2-Phase Commit



Что происходит в случае сбоя?

# 2PC Abort



# Недостатки

## 1. Каскадные аборты

- Могут произойти только в случае сбоя сервера или ОС (сравнительно редко)

## 2. Одно-грейновые транзакции требуют валидации (медленно)

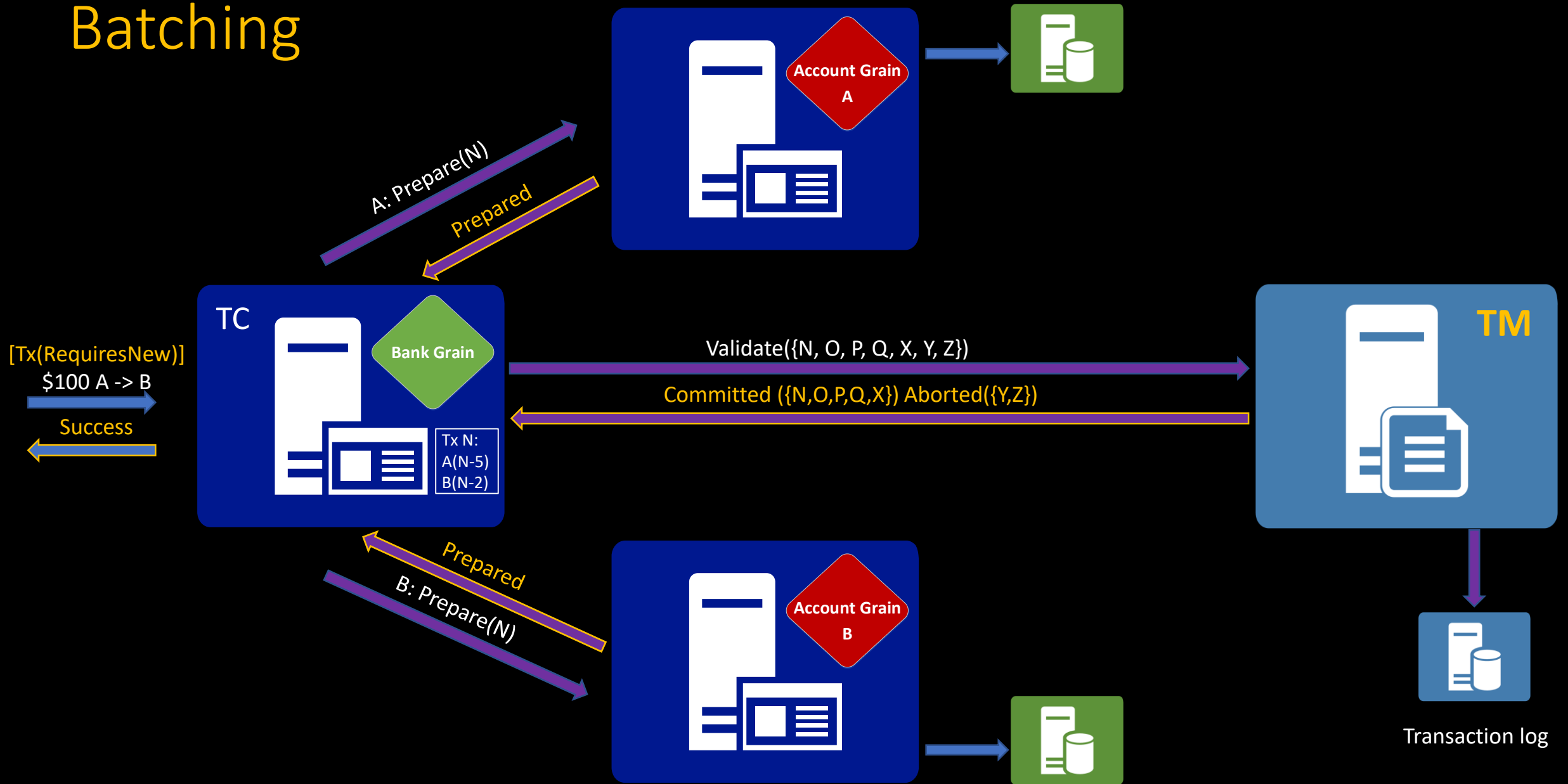
- Влияет на скорость, но не на пропускную способность

## 3. Отдельный ТМ ведёт к дополнительным операционным расходам

## 4. Единый ТМ ограничивает масштабируемость

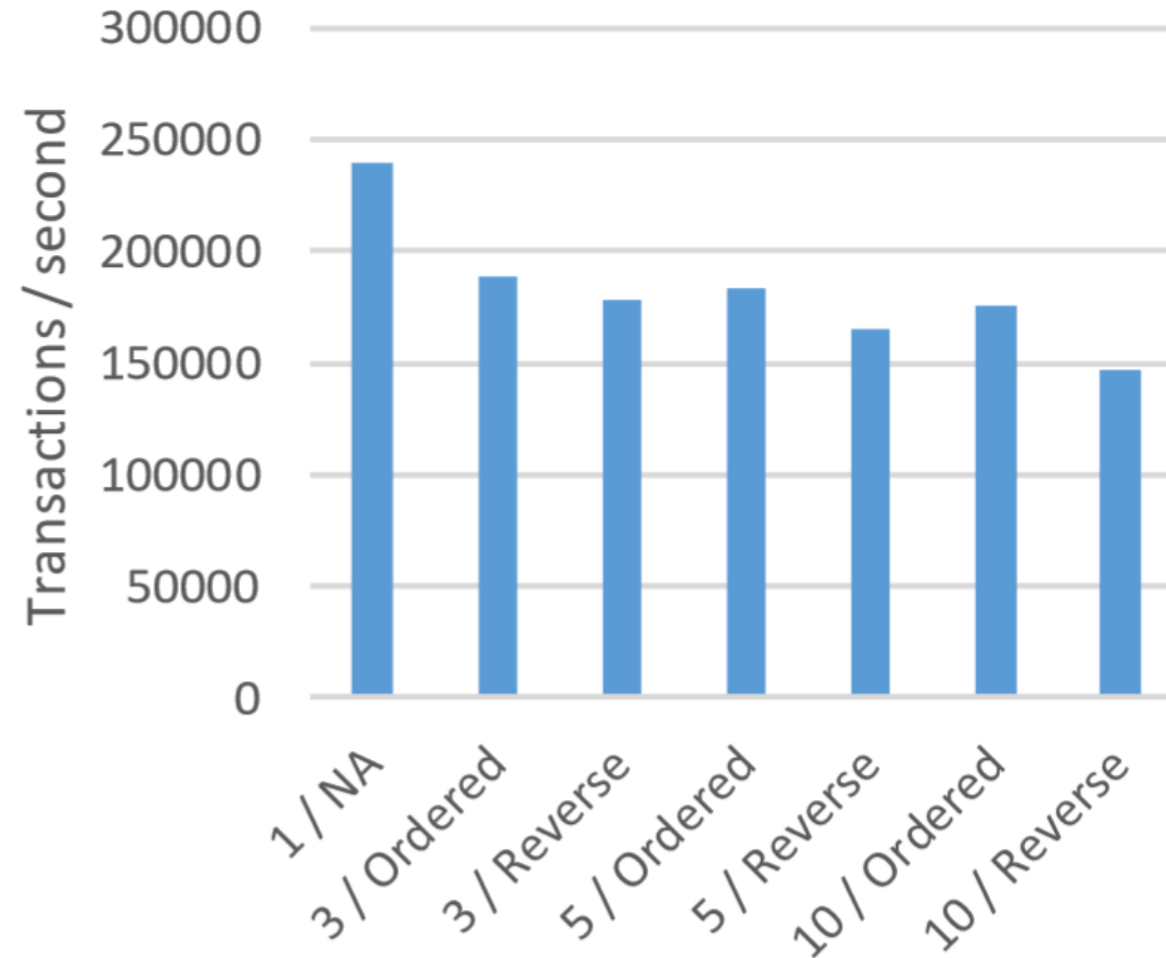
- Не самая плохая проблема
- Есть способы улучшить...

# Batching



# Производительность ТМ

8-core VM

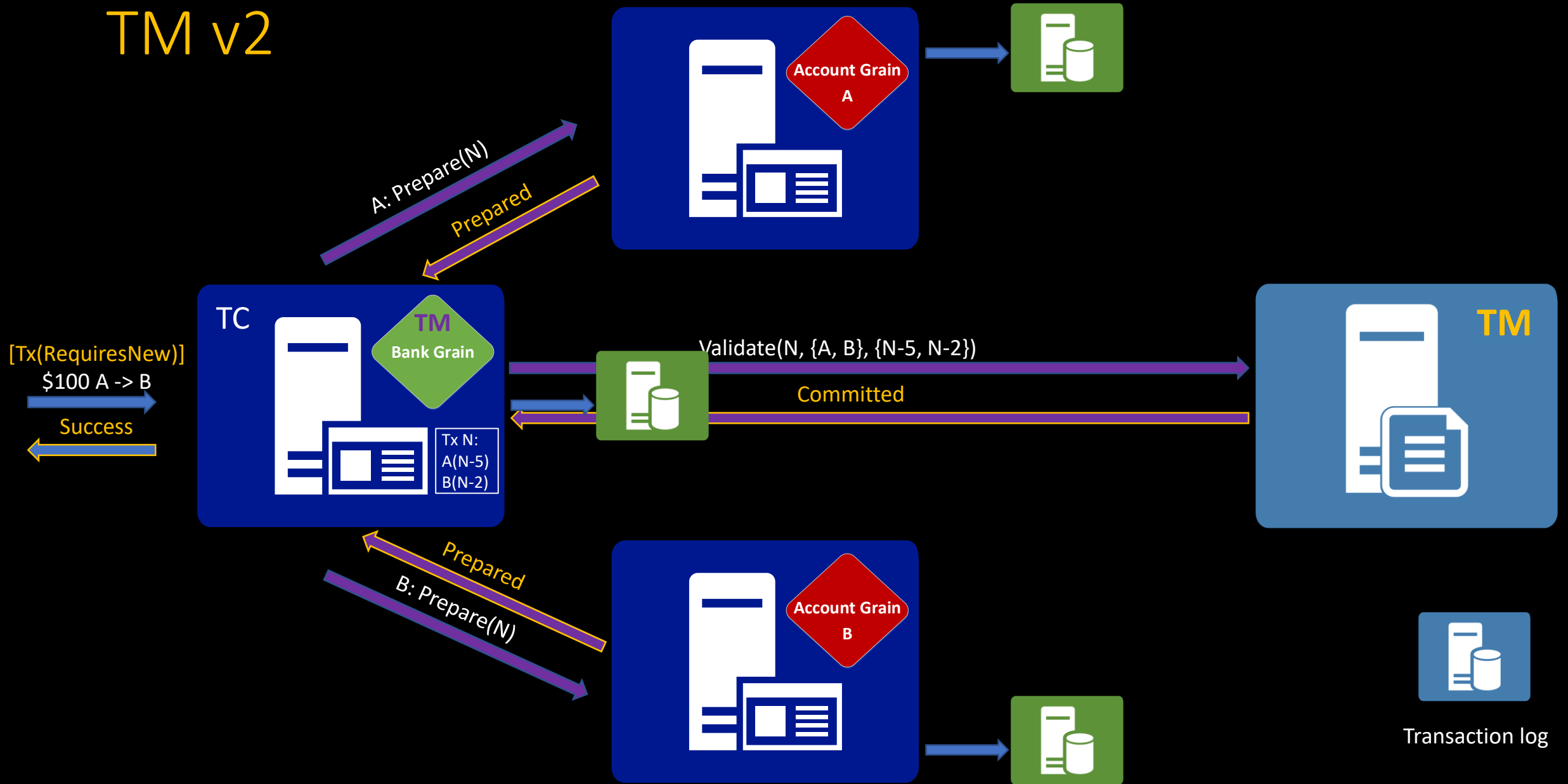


**Figure 6 TM Throughput**



И это только начало (Beta)

# TM v2



# Недостатки TM v1

## 1. Каскадные аборты

- Могут произойти только в случае сбоя сервера или ОС

## 2. Одно-грейновые транзакции требуют валидации

- Влияет на скорость, но не на пропускную способность

## 3. Отдельный TM ведёт к дополнительным оп. расходам

## 4. Единый TM ограничивает масштабируемость

- Не самая плохая проблема



Phil Bernstein



Sebastian Burckhardt



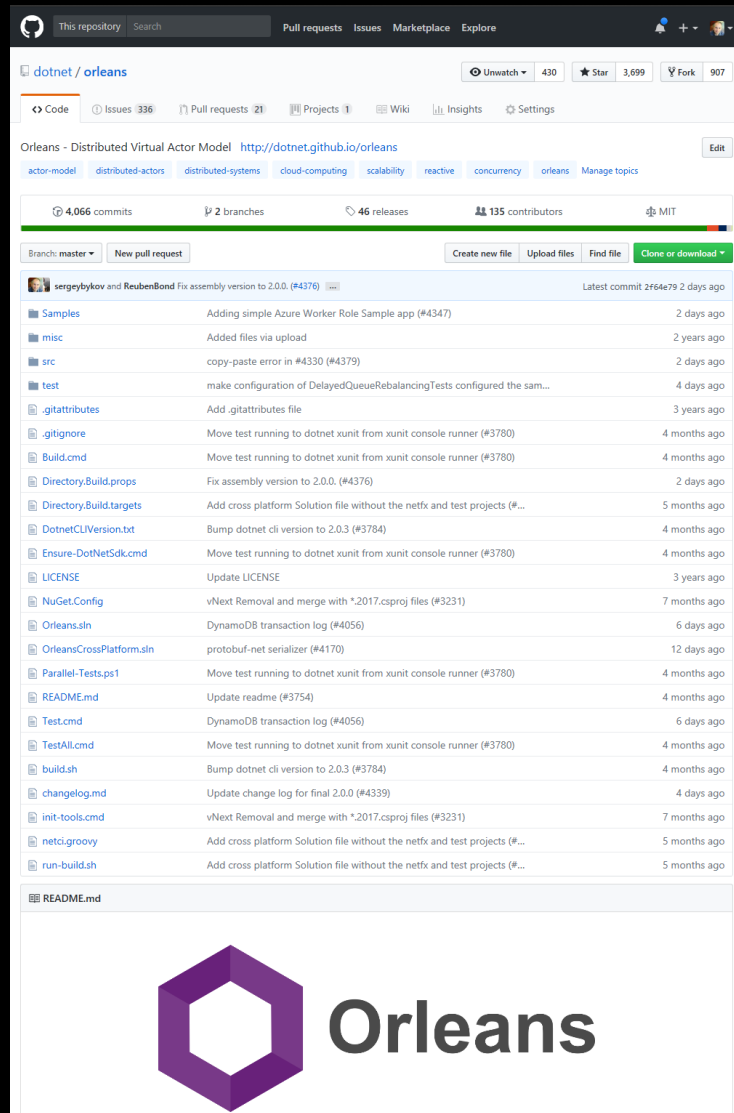
Christopher Meiklejohn



Alejandro Tomsic

Самое главное...

# Весь код живёт на GitHub



dotnet / orleans

Unwatch 430 Star 3,699 Fork 907

Code Issues 336 Pull requests 21 Projects 1 Wiki Insights Settings

Orleans - Distributed Virtual Actor Model <http://dotnet.github.io/orleans> Edit

actor-model distributed-actors distributed-systems cloud-computing scalability reactive concurrency orleans Manage topics


4,066 commits 2 branches 46 releases 135 contributors MIT

Branch: master New pull request Create new file Upload files Find file Clone or download

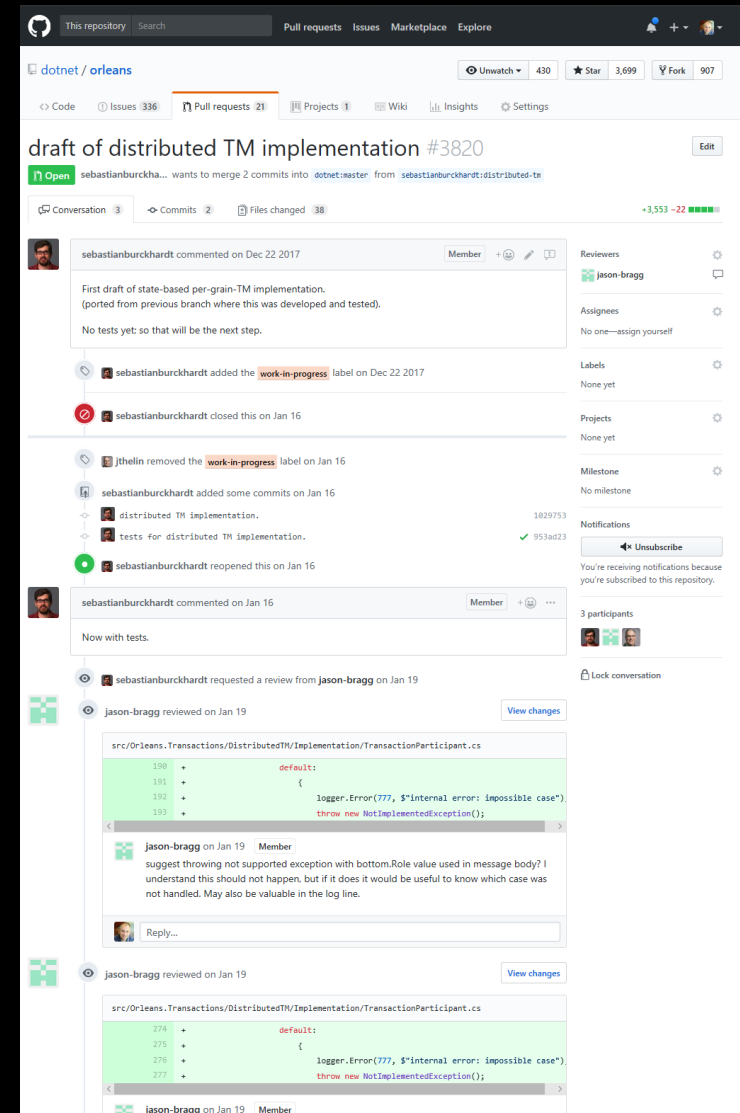
sergebykov and ReubenBond Fix assembly version to 2.0.0. (#4376) Latest commit 2f64e79 2 days ago

- Samples Adding simple Azure Worker Role Sample app (#4347) 2 days ago
- misc Added files via upload 2 years ago
- src copy-paste error in #4330 (#4379) 2 days ago
- test make configuration of DelayedQueueRebalancingTests configured the sam... 4 days ago
- .gitattributes Add .gitattributes file 3 years ago
- .gitignore Move test running to dotnet xunit from xunit console runner (#3780) 4 months ago
- Build.cmd Move test running to dotnet xunit from xunit console runner (#3780) 4 months ago
- Directory.Build.props Fix assembly version to 2.0.0. (#4376) 2 days ago
- Directory.Build.targets Add cross platform Solution file without the netfx and test projects (#... 5 months ago
- DotnetCLIVersion.txt Bump dotnet cli version to 2.0.3 (#3784) 4 months ago
- Ensure-DotNetSdk.cmd Move test running to dotnet xunit from xunit console runner (#3780) 4 months ago
- LICENSE Update LICENSE 3 years ago
- NuGet.Config vNext Removal and merge with \*2017.csproj files (#3231) 7 months ago
- Orleans.sln DynamoDB transaction log (#4056) 6 days ago
- OrleansCrossPlatform.sln protobuf-net serializer (#4170) 12 days ago
- Parallel-Tests.ps1 Move test running to dotnet xunit from xunit console runner (#3780) 4 months ago
- README.md Update readme (#3754) 4 months ago
- Test.cmd DynamoDB transaction log (#4056) 6 days ago
- TestAll.cmd Move test running to dotnet xunit from xunit console runner (#3780) 4 months ago
- build.sh Bump dotnet cli version to 2.0.3 (#3784) 4 months ago
- changelog.md Update change log for final 2.0.0 (#4339) 4 days ago
- init-tools.cmd vNext Removal and merge with \*2017.csproj files (#3231) 7 months ago
- netci.groovy Add cross platform Solution file without the netfx and test projects (#... 5 months ago
- run-build.sh Add cross platform Solution file without the netfx and test projects (#... 5 months ago

README.md



# Orleans



dotnet / orleans

Unwatch 430 Star 3,699 Fork 907

Code Issues 336 Pull requests 21 Projects 1 Wiki Insights Settings

draft of distributed TM implementation #3820 Edit

Open sebastianburckhardt wants to merge 2 commits into dotnet:master from sebastianburckhardt:distributed-tm

Conversation 3 Commits 2 Files changed 38 +3,553 -22

sebastianburckhardt commented on Dec 22 2017 Member

First draft of state-based per-grain-TM implementation. (ported from previous branch where this was developed and tested). No tests yet so that will be the next step.

sebastianburckhardt added the work-in-progress label on Dec 22 2017

sebastianburckhardt closed this on Jan 16

jthelin removed the work-in-progress label on Jan 16

sebastianburckhardt added some commits on Jan 16

- distributed TM implementation. 1829753
- tests for distributed TM implementation. 953ad23

sebastianburckhardt reopened this on Jan 16

sebastianburckhardt commented on Jan 16 Member

Now with tests.

sebastianburckhardt requested a review from jason-bragg on Jan 19

jason-bragg reviewed on Jan 19 View changes

```
src/Orleans.Transactions/DistributedTM/Implementation/TransactionParticipant.cs
198 +         default:
199 +         {
200 +             logger.Error(777, $"internal error: impossible case");
201 +             throw new NotImplementedException();
202 +         }
```

jason-bragg on Jan 19 Member

suggest throwing not supported exception with bottom.Role value used in message body? I understand this should not happen, but if it does it would be useful to know which case was not handled. May also be valuable in the log line.

Reply...

jason-bragg reviewed on Jan 19 View changes

```
src/Orleans.Transactions/DistributedTM/Implementation/TransactionParticipant.cs
274 +         default:
275 +         {
276 +             logger.Error(777, $"internal error: impossible case");
277 +             throw new NotImplementedException();
278 +         }
```

jason-bragg on Jan 19 Member

# Заключение

В жизни всегда есть место инновациям, даже в транзакциях

- SQL <-> NoSQL <-> Distributed SQL <-> Distributed Transactions

Важно иногда сомневаться в общепринятых «истинах»

- Нести «ересь» нелегко

Слой middle-tier позволяет сделать много интересного

- Не привязываясь к конкретному хранилищу данных

Open Source и никак иначе

- Присоединяйтесь!



David Fowler @davidfowl  
hey



Reuben Bond @ReubenBond  
Hey

[@davidfowl](#)



David Fowler @davidfowl  
hey

I'm here to learn about actors 😊



Reuben Bond @ReubenBond

I'm heading out at the moment (Friday night here), but I hope I can help anyway. Otherwise my DMs are open or we can Skype some time.



David Fowler @davidfowl  
ah yes

its not urgent



Reuben Bond @ReubenBond

What prompted you to look into actors?



David Fowler @davidfowl

with bedrock it now possible to grab a transport layer and a protocol layer and build an arbitrary programming model on top  
so I was looking at what it would look like if I made a class that was tied to a connection that was long lived on the server  
and somebody told me they basically do that with websockets  
they have a custom actor framework



Gutemberg Ribeiro @galvesribeiro

[@davidfowl](https://github.com/OrleansContrib/SignalR.Orleans) <https://github.com/OrleansContrib/SignalR.Orleans>

I may add the chat sample there

[@ReubenBond](#) regard the reminder method, that is meant to be called at onactivate



Gutemberg Ribeiro @galvesribeiro

[@ReubenBond](#)

Serializing delegates is not supported on this platform.

It used to work before 2.0

and the only serializer they have in the code is:

```
<Messaging>
  <SerializationProviders>
    <Provider type="Orleans.Serialization.ProtobufSerializer, OrleansGoogleUtils" />
  </SerializationProviders>
</Messaging>
```

and I added it as Singleton on DI



Jakub Konecki @jkonecki

Welcome, [@davidfowl](#) ! You've come to the right place if you're looking for actors 😊



# Спасибо!

Sergey Bykov  
Microsoft, @sergeybykov



@msftorleans  
<https://github.com/dotnet/orleans/>