Distributed Transactions on Serverless Stateful Functions

Distributed Transactions on Serverless Stateful Functions
Martijn De Heus, Kyriakos Psarakis, Marios Fragkoulis, Asterios Katsifodimos.
To appear in the proceedings of the 15th ACM International Conference on Distributed and Event-based Systems (DEBS) 2021.

TU Delft

- Transactions
- Microservices & Cloud computing
- Stateful Functions & Coordinator Functions
- Evaluation
- Conclusion & discussion

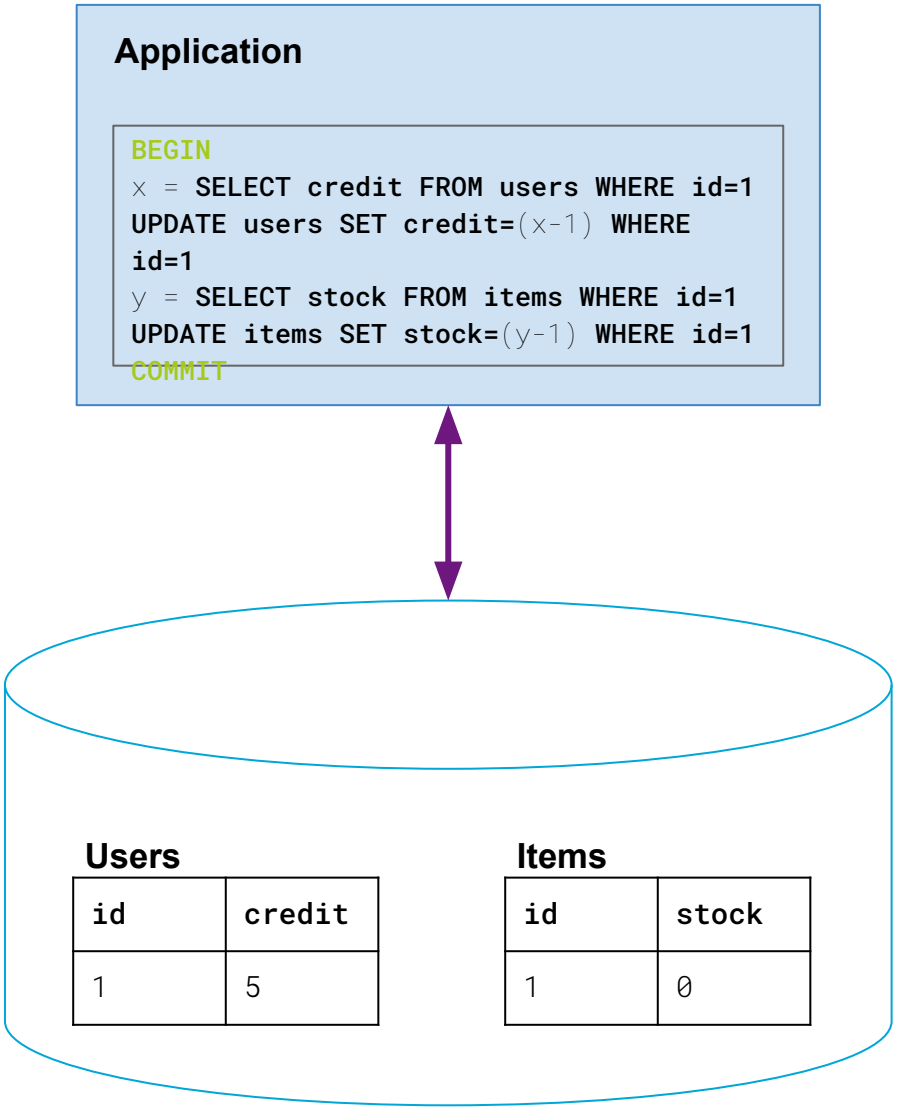
Transactions

Transactions

A transactions are a set of operations on data with 4 (**ACID**) guarantees:

- **A**tomicity
- **C**onsistency
- **I**solation
  - (*serializability*)
- **D**urability

Traditional Architecture



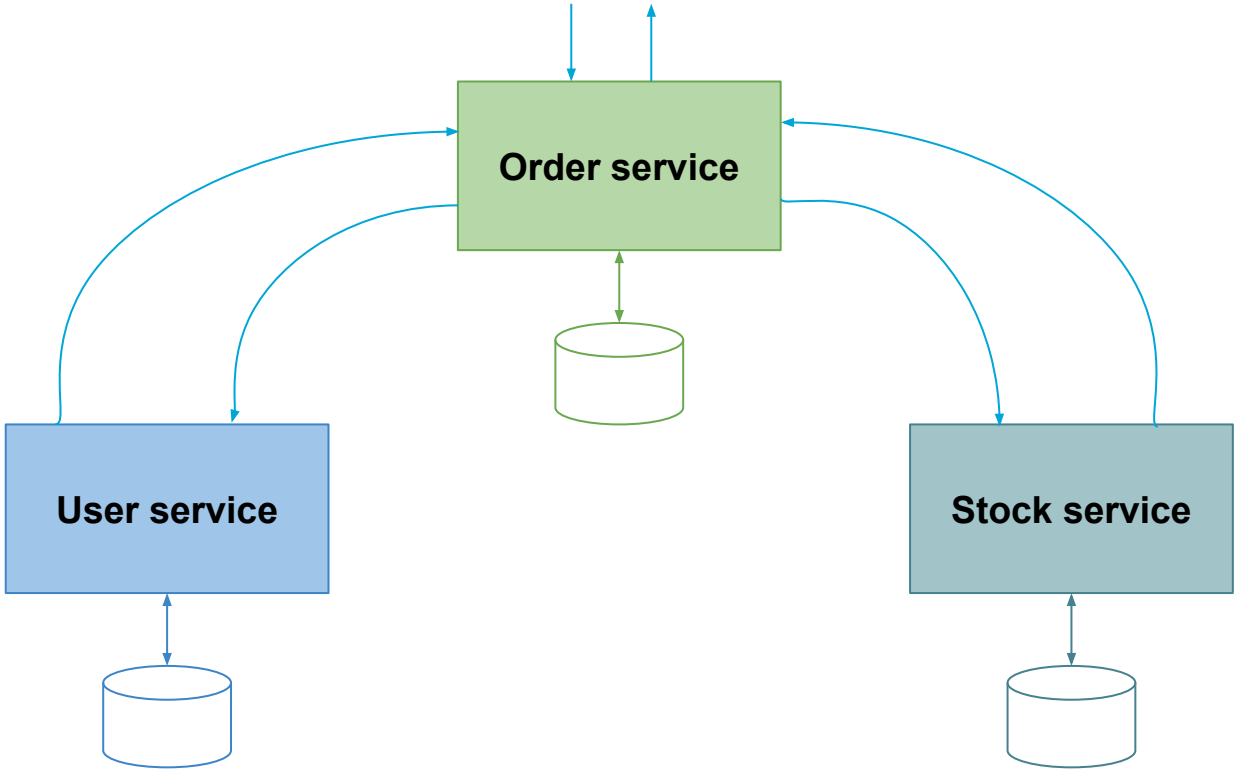
Distributed transactions

- **Two-Phase Commit** ensures **atomicity**
- **Two-Phase Locking** ensures **serializability**

Combining Two-Phase Locking and Two-Phase Commit can be used to implement **distributed** transactions

Microservice systems

Microservices

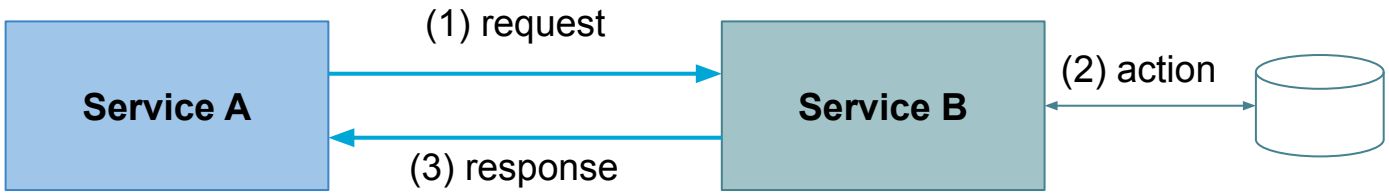


Microservices

Benefits

- Scaling components separately
- Deploying / updating components separately
- Deploying components on specific hardware
- Scaling development in organisations

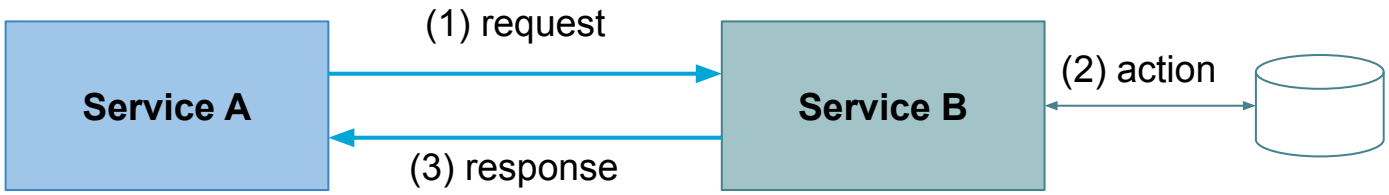
Two Generals' Problem



The network is **unreliable**, it is never certain whether a message will arrive

If **Service A** never receives message at 3, it does not know whether **Service B** has performed action at 2.

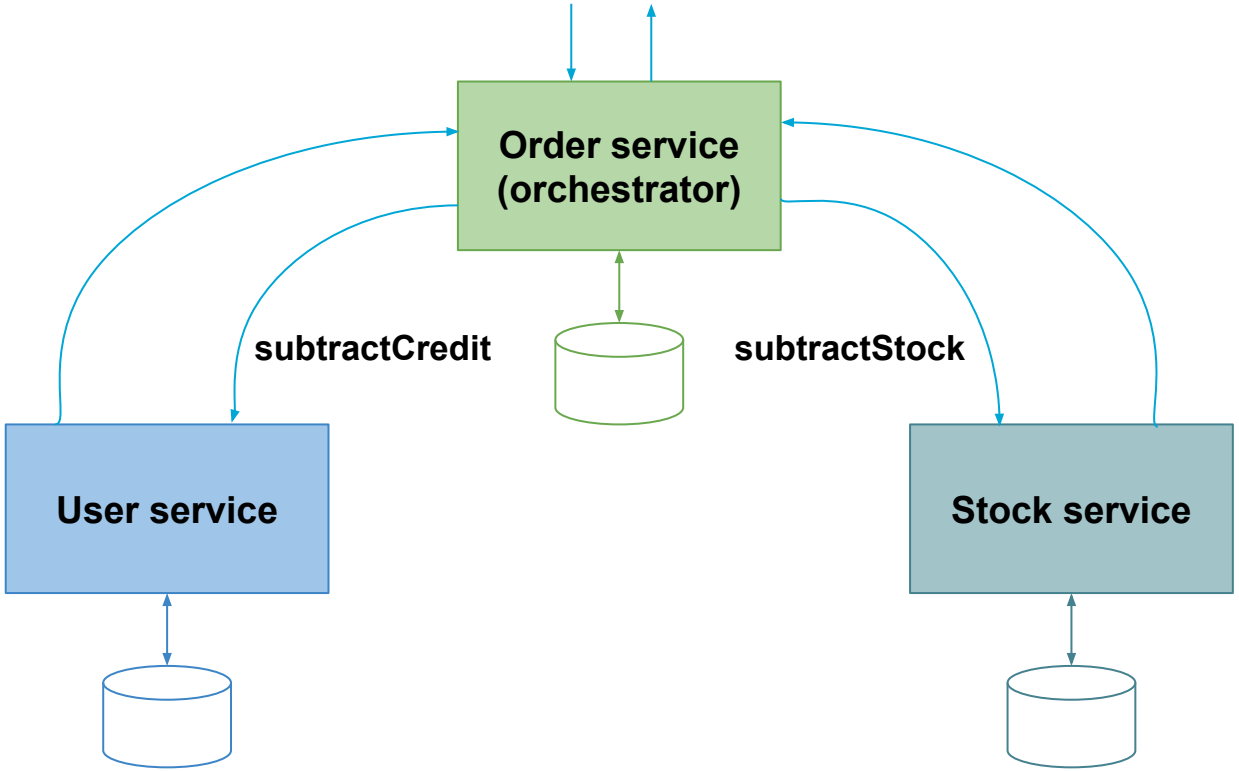
Processing guarantees



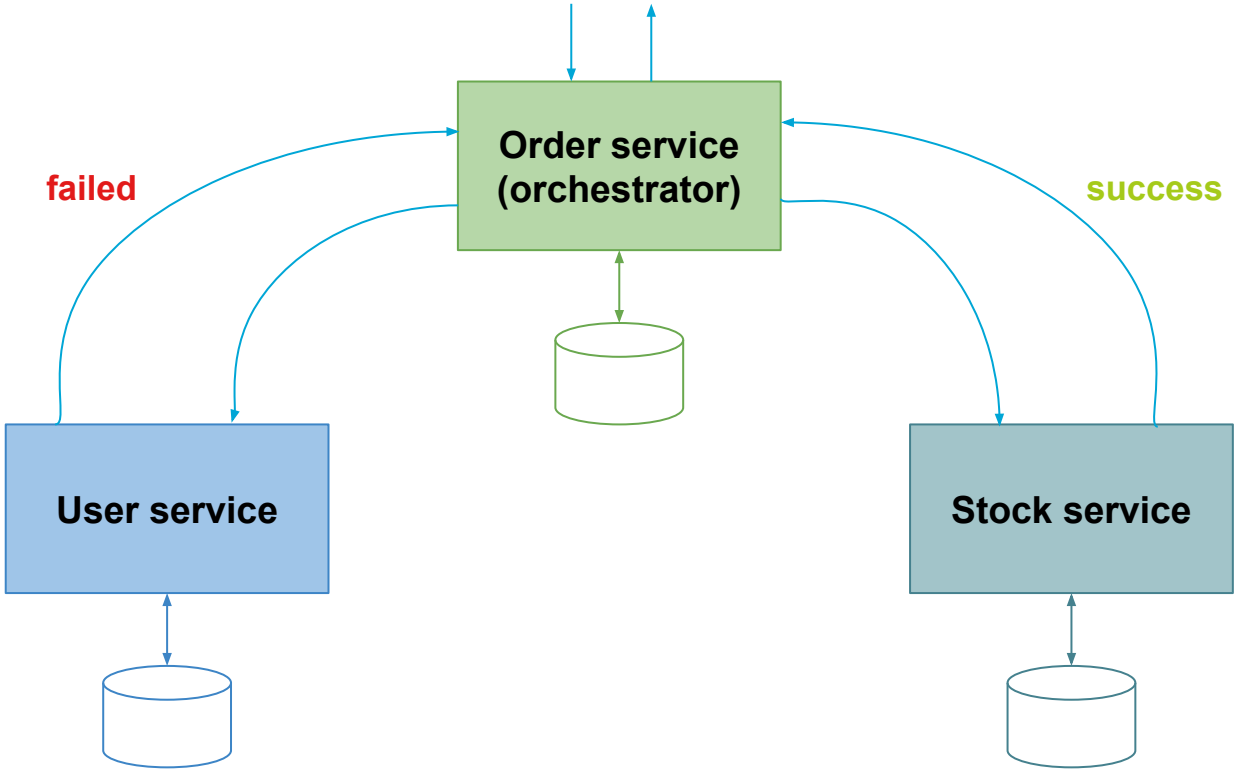
AT_MOST_ONCE: only try to send once and do not care about the results, action 2 may not be performed

AT_LEAST_ONCE: keep retrying until a response is received, action 2 may be performed multiple times

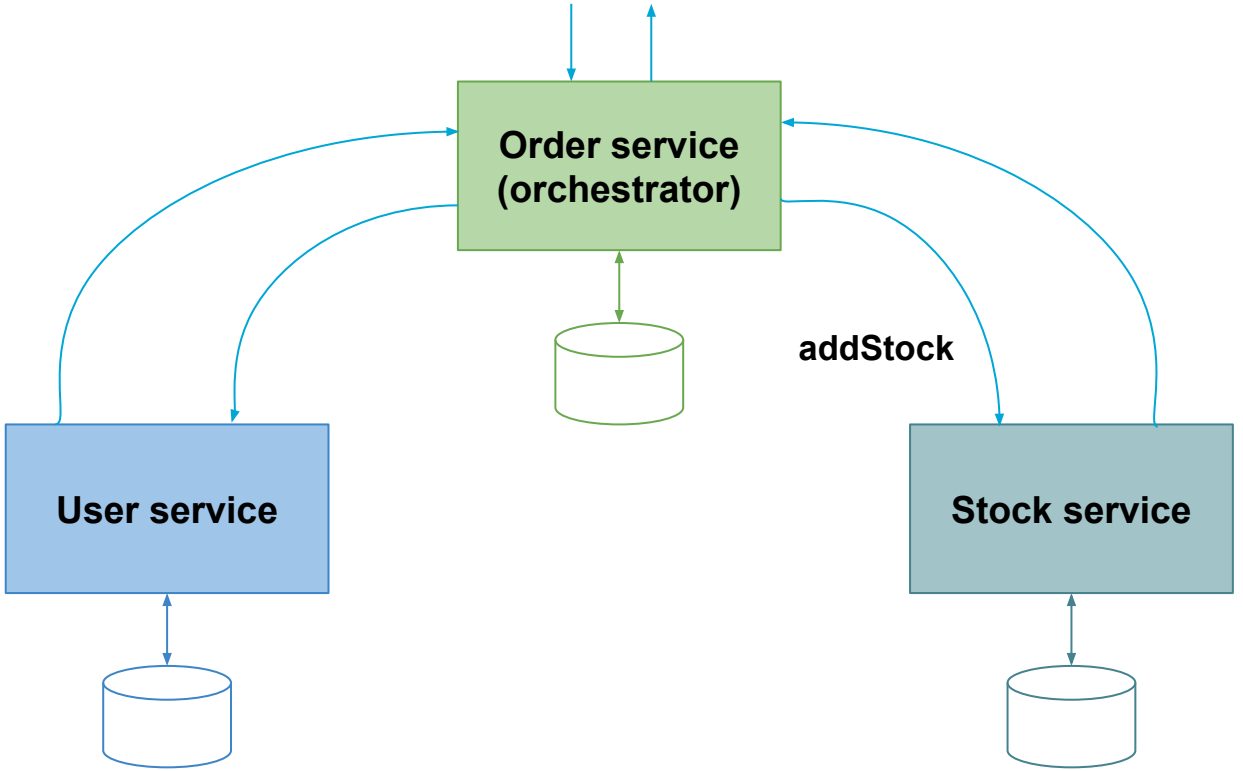
Saga orchestration



Saga orchestration



Saga orchestration



Transactions but with completely no <u>isolation</u>
<u>Atomicity</u> with of fault-tolerance is also hard to achieve

Cloud computing

Cloud computing

Make computing **accessible** for everyone

Goals:

- Eliminate up-front investment requirements and offer compute resources on demand
- Simplify distributed computing
  - Deployment and scalability
  - State management
    - Processing guarantees (exactly-once)
    - Distributed transactions

Cloud computing

Make computing **accessible** for everyone

Goals:

- ✓ Eliminate up-front investment requirements and offer compute resources on demand
- ~ Simplify distributed computing
  - Deployment and scalability
  - State management
    - Processing guarantees (exactly-once)
    - Distributed transactions

How can we improve on this?

Make computing **accessible** for everyone

Goals:

✓ Eliminate up-front investment requirements and offer compute resources on demand

~ Simplify distributed computing
- Deployment and scalability
- State management
  - Processing guarantees (exactly-once)
  - Distributed transactions

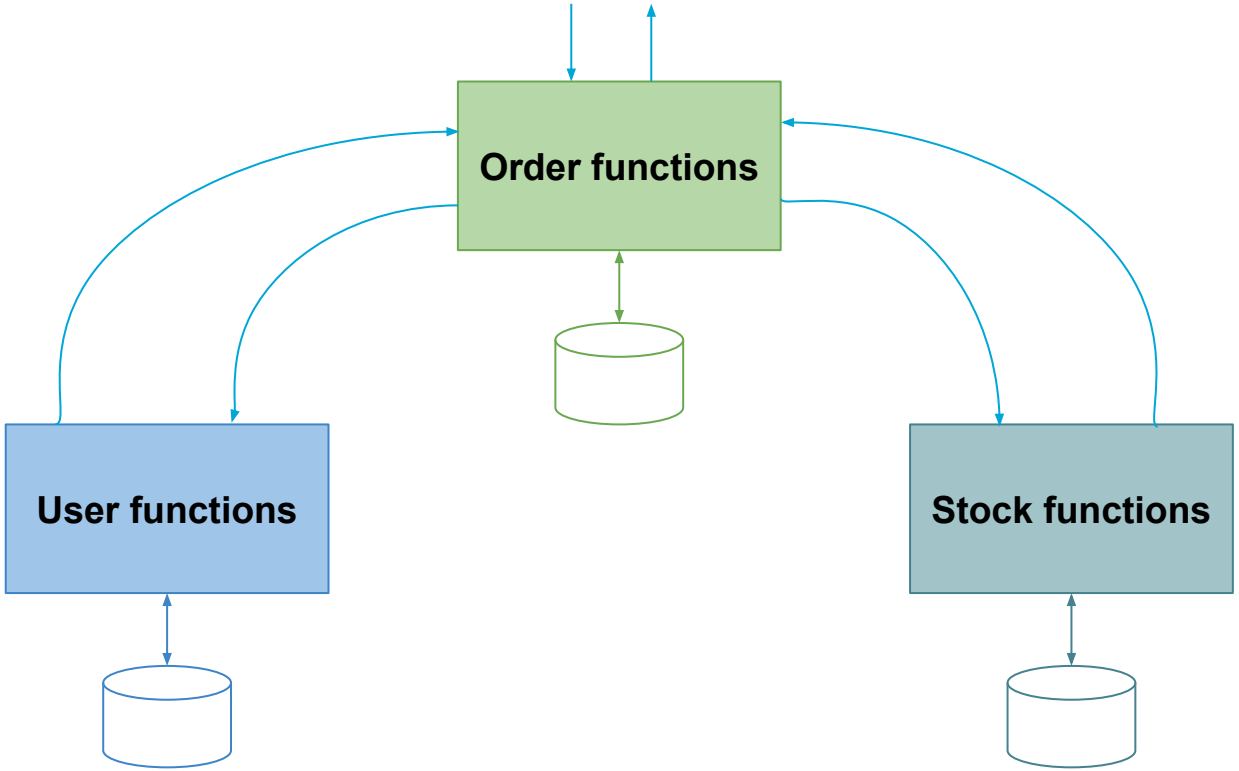
Serverless computing

Function-as-a-Service (FaaS) models as the first iteration of serverless

Goals:

- ✓ Eliminate up-front investment requirements and offer compute resources on demand
- ~ Simplify distributed computing
  - ✓ Deployment and scalability
  - State management
    - Processing guarantees (exactly-once)
    - Distributed transactions

Function-as-a-Service



Serverless computing

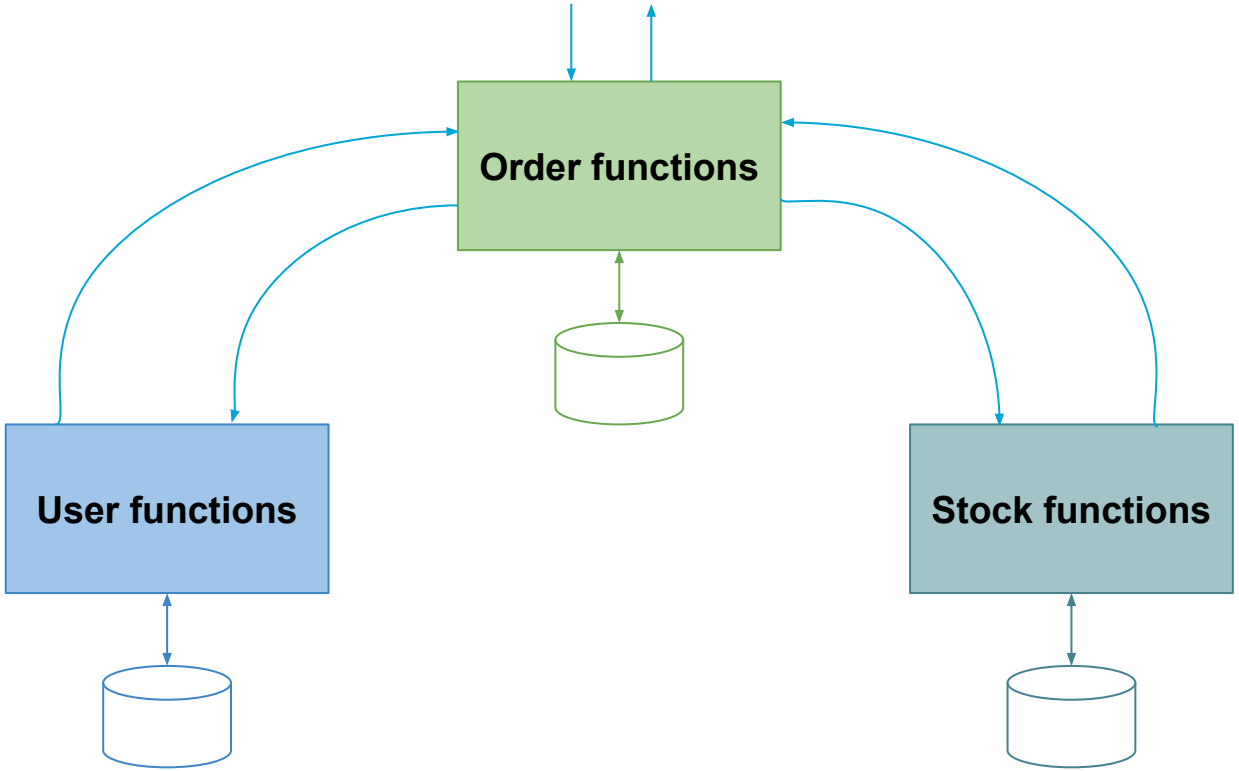
Stateful-Function-as-a-Service (SFaaS)
models as the second iteration of serverless

Goals:

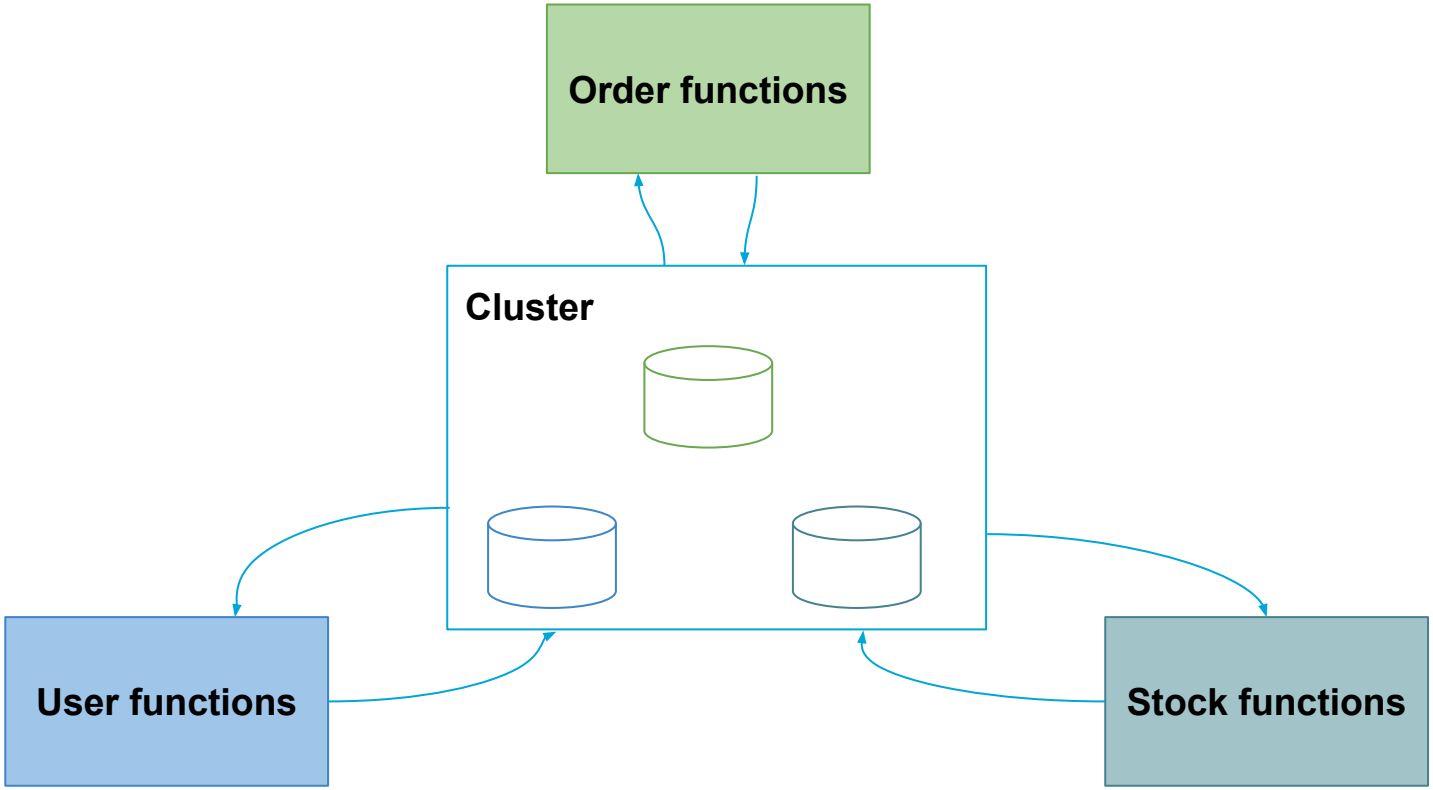
- ✓ Eliminate up-front investment requirements and offer compute resources on demand
- ~ Simplify distributed computing
  - ✓ Deployment and scalability
  - ~ State management
    - ✓ Processing guarantees (exactly-once)
    - • Distributed transactions

Flink Statefun

Functions-as-a-Service

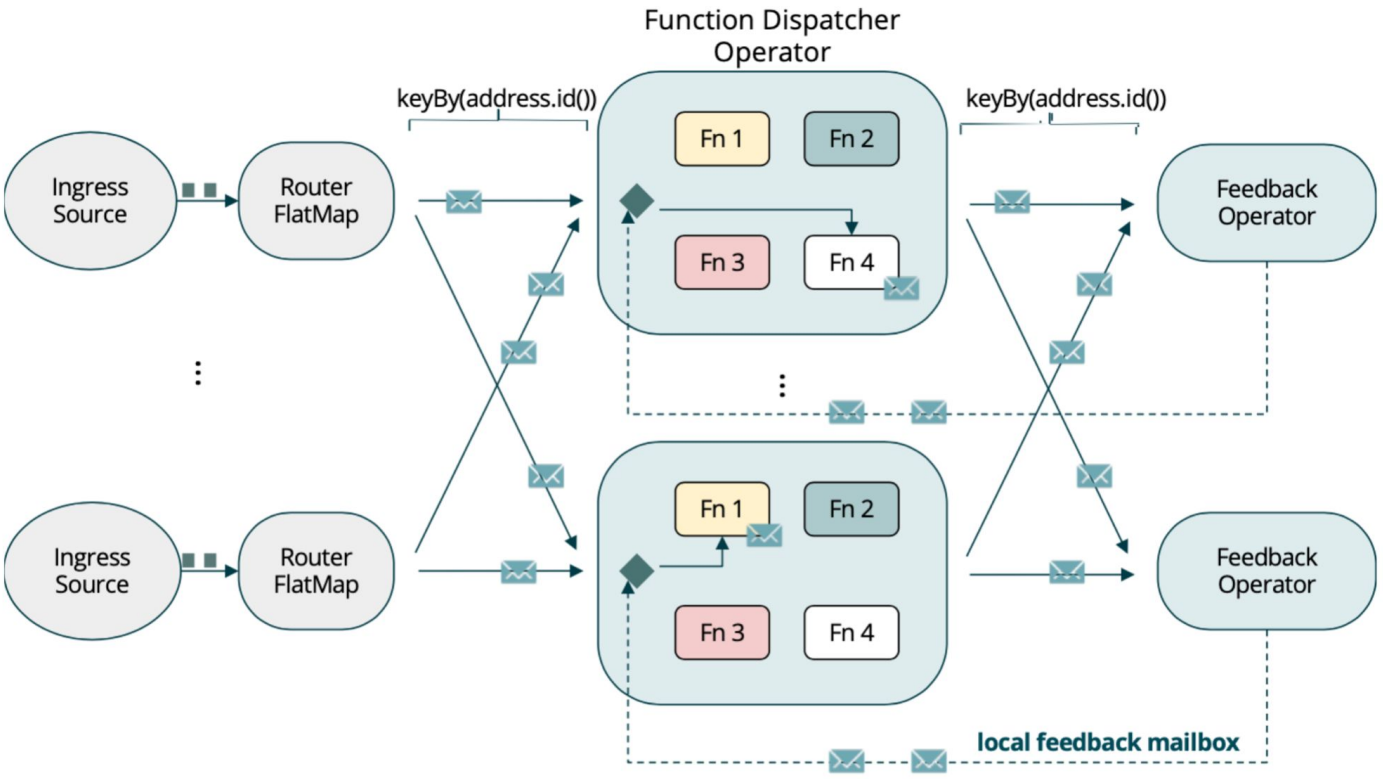


Stateful Function-as-a-Service



Couples *messaging* and *state management* to ensure exactly-once guarantees across both

Flink Statefun



Flink Statefun

Function instances encapsulate specific state based on their **address** (combination of *type* and *id*)

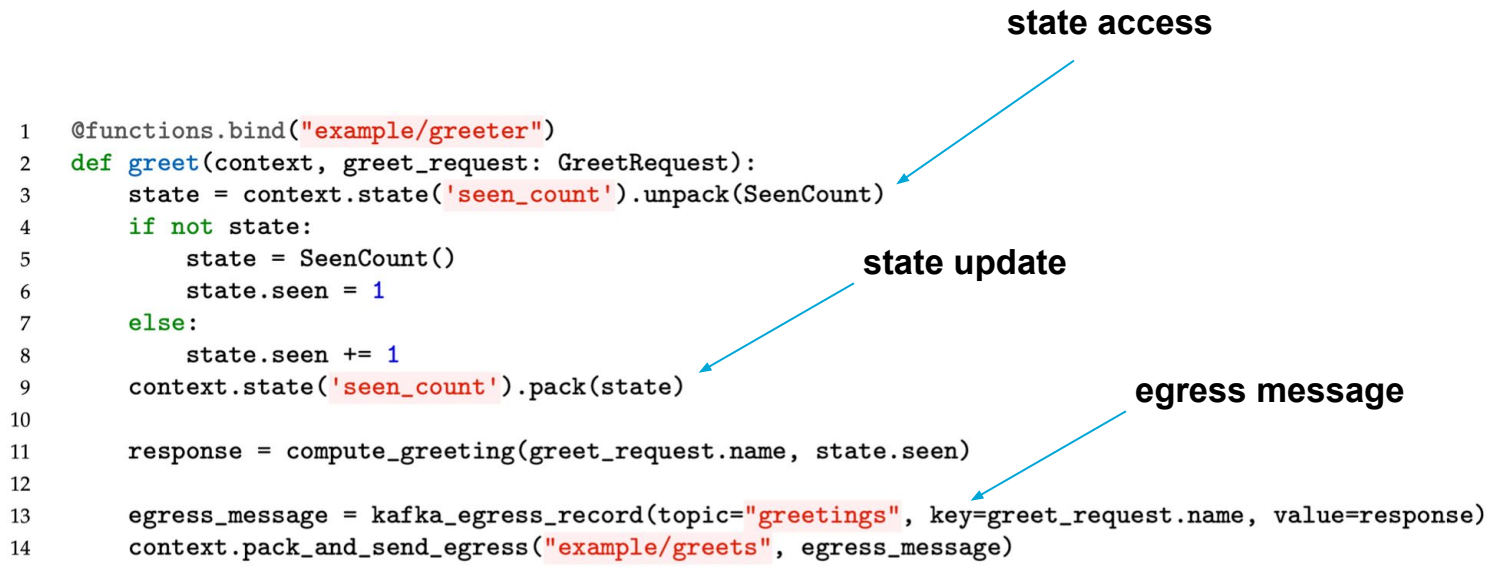
Function instances can perform **four side effects**:

- Updates to their encapsulated state
- Messages to other function instances
- Delayed messages to other function instances
- Messages to egresses

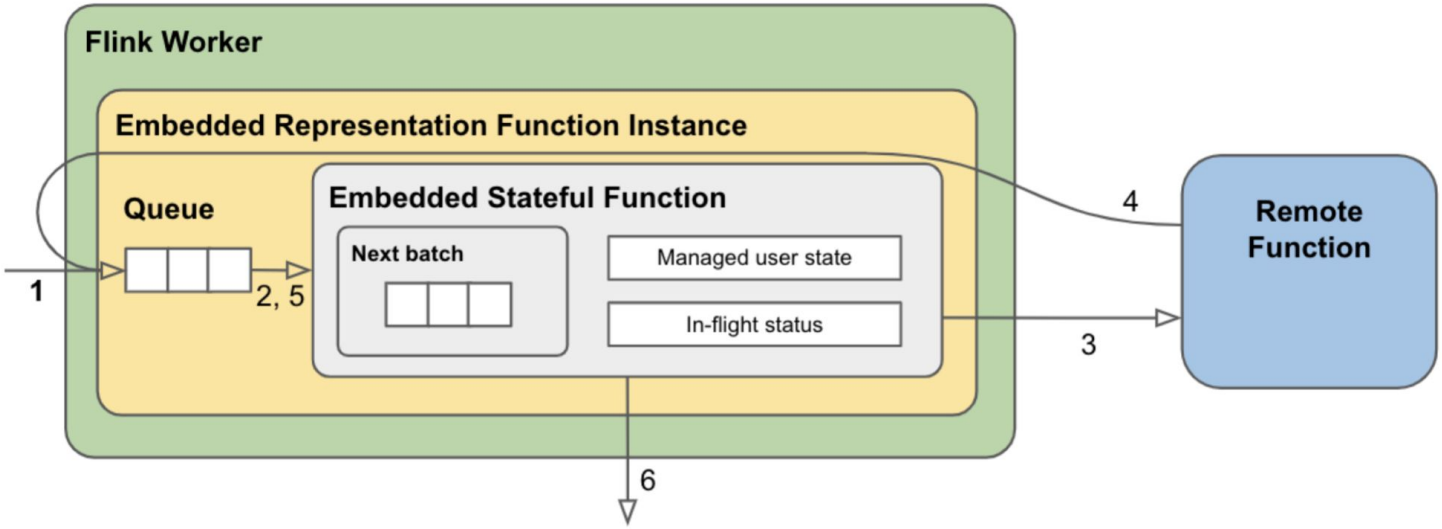
Function instances can be **invoked** by:

- Messages from other function instances
- Messages from ingresses

Flink Statefun



Flink Statefun




```
1   message InvocationBatchRequest {
2       Address target = 1;
3       repeated PersistedValue state = 2;
4       repeated Invocation invocations = 3;
5   }
6
7   message InvocationResponse {
8       repeated PersistedValueMutation state_mutations = 1;
9       repeated Invocation outgoing_messages = 2;
10      repeated DelayedInvocation delayed_invocations = 3;
11      repeated EgressMessage outgoing_egresses = 4;
12  }
```

State is sent to the remote function

The resulting side effects

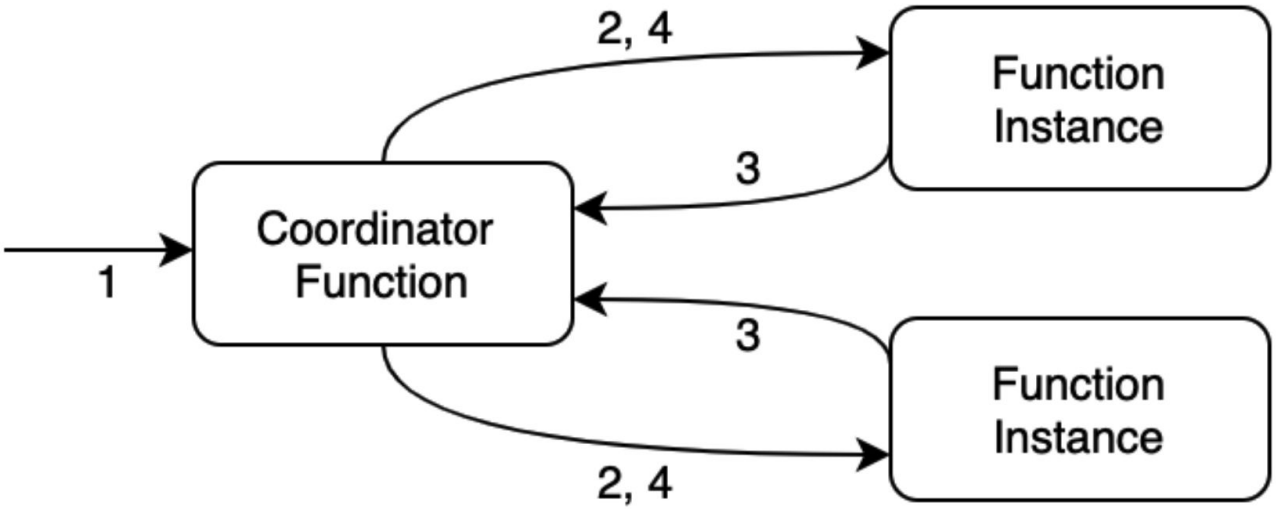
Coordinator functions for distributed transactions

Serverless computing

Goals:

- ✓ Eliminate up-front investment requirements and offer compute resources on demand
- ✓ Simplify distributed computing
  - ✓ Deployment and scalability
  - ✓ State management
    - ✓ Processing guarantees (exactly-once)
    - ✓ Distributed transactions

Coordinator functions



This relies on Statefun's exactly_once guarantees and linearizable operations on single function instances

Coordinator functions

Coordinator functions are simply *specialized function instances* to coordinate a serializable transaction or sagas

Coordinator function instances can perform **side effects**:

- Add messages to other function instances to the saga or transactions
- Add side effects to perform based on the completion of the saga

Coordinator function instances can be **invoked** by:

- Messages from other function instances
- Messages from ingresses

Coordinator functions

invocations part of the transaction

```
def handle_transfer(context, message: Transfer):
    # Send messages
    subtract_credit = SubtractCredit(amount = message.amount)
    context.pack_and_send_atomic_invocation("ycsb/account_function",
                                            message.outgoing_id,
                                            subtract_credit)
    add_credit = AddCredit(amount = message.amount)
    context.pack_and_send_atomic_invocation("ycsb/account_function",
                                            message.incoming_id,
                                            add_credit)
```

Coordinator functions

invocations part of the saga

require explicit compensating invocations

```
1   def handle_transfer(context, message: Transfer):
2       # Send messages
3       subtract_credit = SubtractCredit(amount=message.amount)
4       add_credit = AddCredit(amount=message.amount)
5       context.pack_and_send_invocation_pair("ycsb-example/account_function",
6                                             message.outgoing_id,
7                                             subtract_credit,
8                                             add_credit)
9       context.pack_and_send_invocation_pair("ycsb-example/account_function",
10                                            message.incoming_id,
11                                            add_credit,
12                                            subtract_credit)
```

Regular functions

- Functions should be able to **fail** explicitly
- **Batching** should be **adjusted** to respect **isolation** of invocations part of a **serializable transaction**
- **Locking** should be introduced
- It should participate in the protocols for sagas and serializable transactions

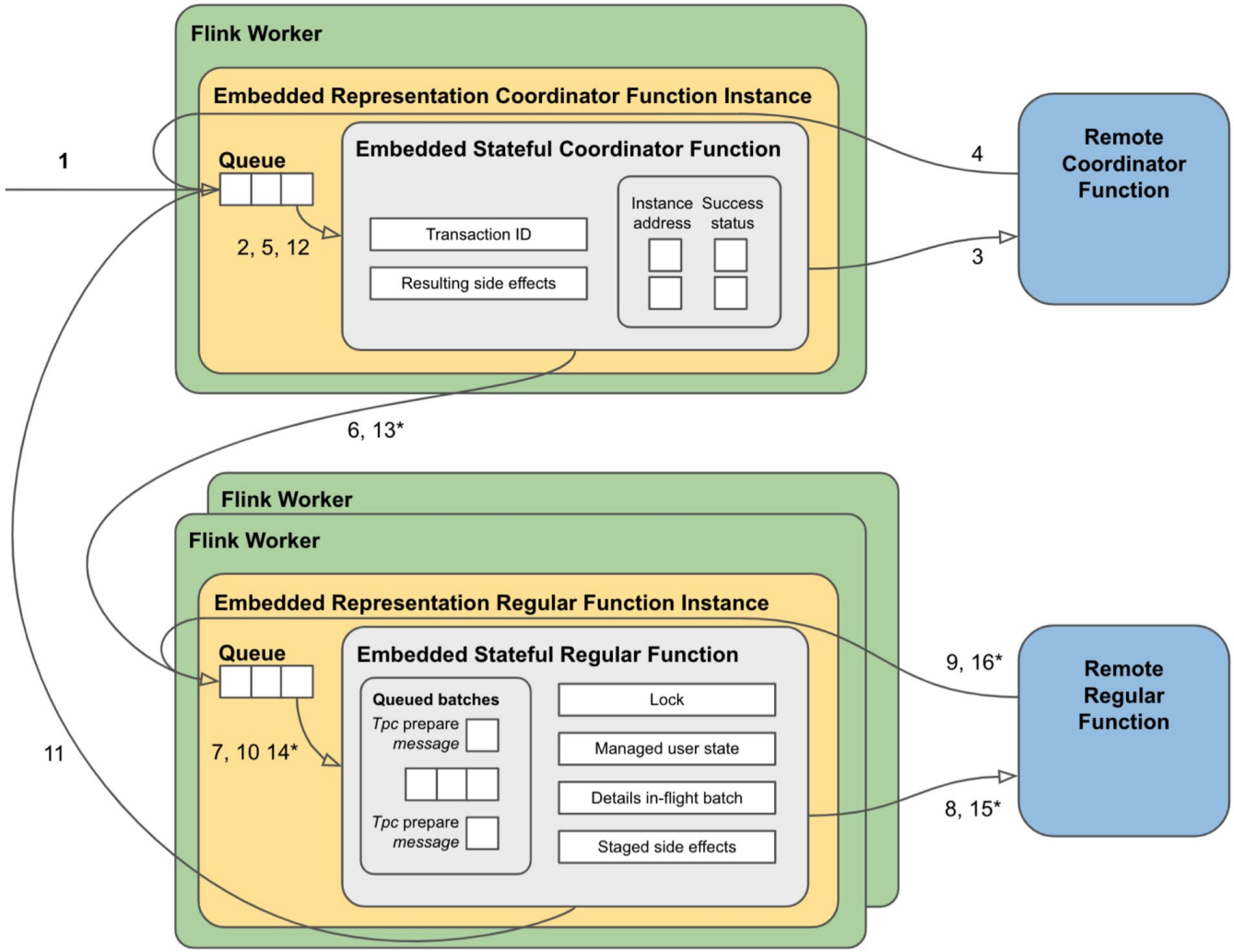
Explicit failure

```python
class NotFoundException(FunctionInvocationException):
    def __init__(self, request_id):
        super().__init__(request_id)

@functions.bind("example/user_function")
async def account_function(context, message: Read):
    # Get state
    user = context.state('user').unpack(User)

    if not user:
        raise NotFoundException(message.request_id)
    else:
        response = Response(request_id=request_id, status_code=200, state=user)
        egress_message = kafka_egress_record(topic="responses", key=request_id, value=response)
        context.pack_and_send_egress("ycsb-example/kafka-egress", egress_message)

@functions.bind_exception_handler(NotFoundException)
async def handle_not_found(context, request_id):
    response = Response(request_id=request_id, status_code=404)
    egress_message = kafka_egress_record(topic="responses", key=request_id, value=response)
    context.pack_and_send_egress("ycsb-example/kafka-egress", egress_message)
```

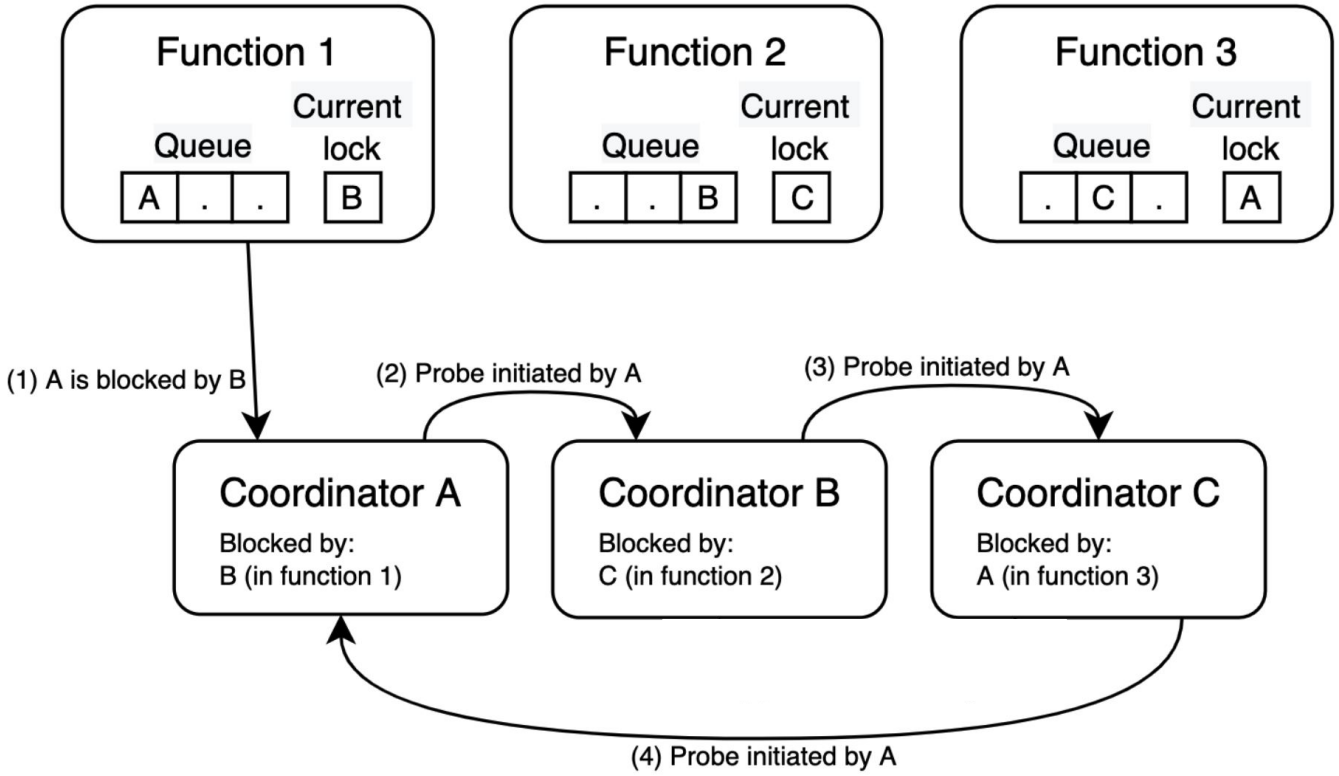
Raise an exception

Exception handler for non-transactional invocations

Coordinator functions



Deadlock detection



K. Mani Chandy, Jayadev Misra, and Laura M. Haas. 1983. Distributed Deadlock Detection. *ACM Trans. Comput. Syst.* 1, 2 (May 1983), 144–156.

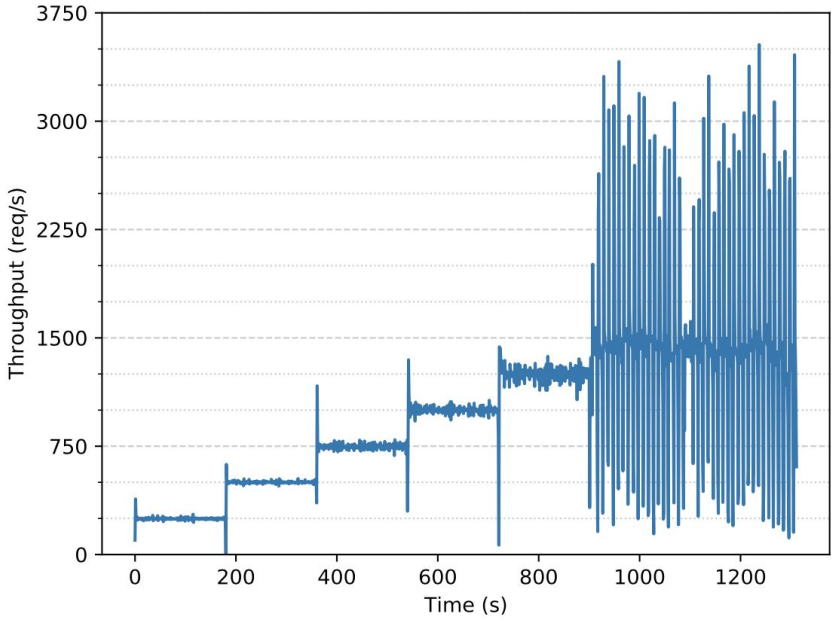
Evaluation

Evaluation

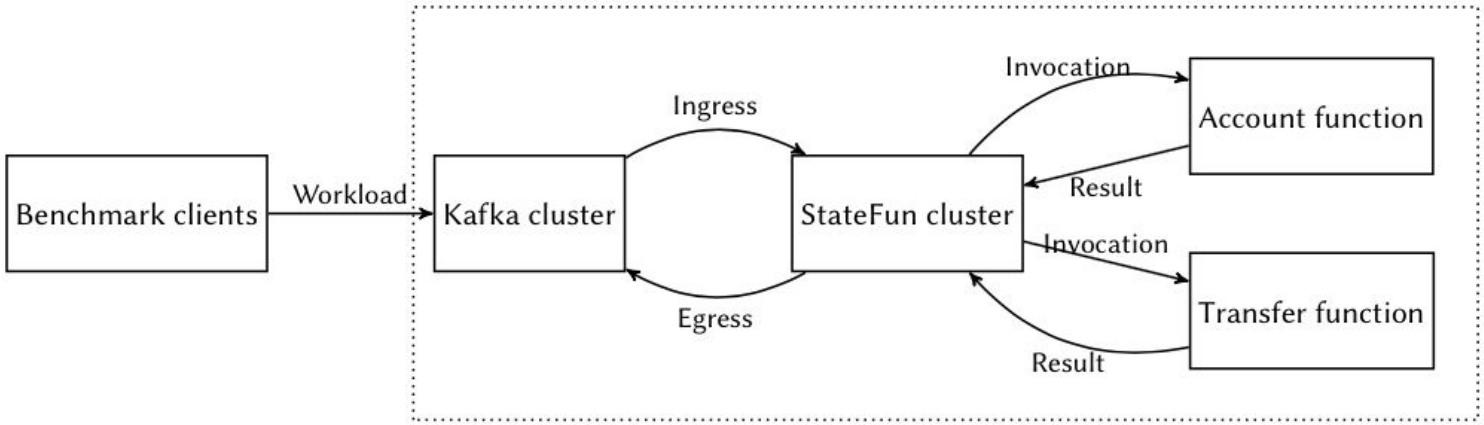
Based on an extension of **YCSB**

- Stage 1: insert *x* keys in the system
- Stage 2: perform of workload of read and write operations
  - Extended with transfer operations
  - Uniform distribution

Measured **maximum throughput** and **latency**

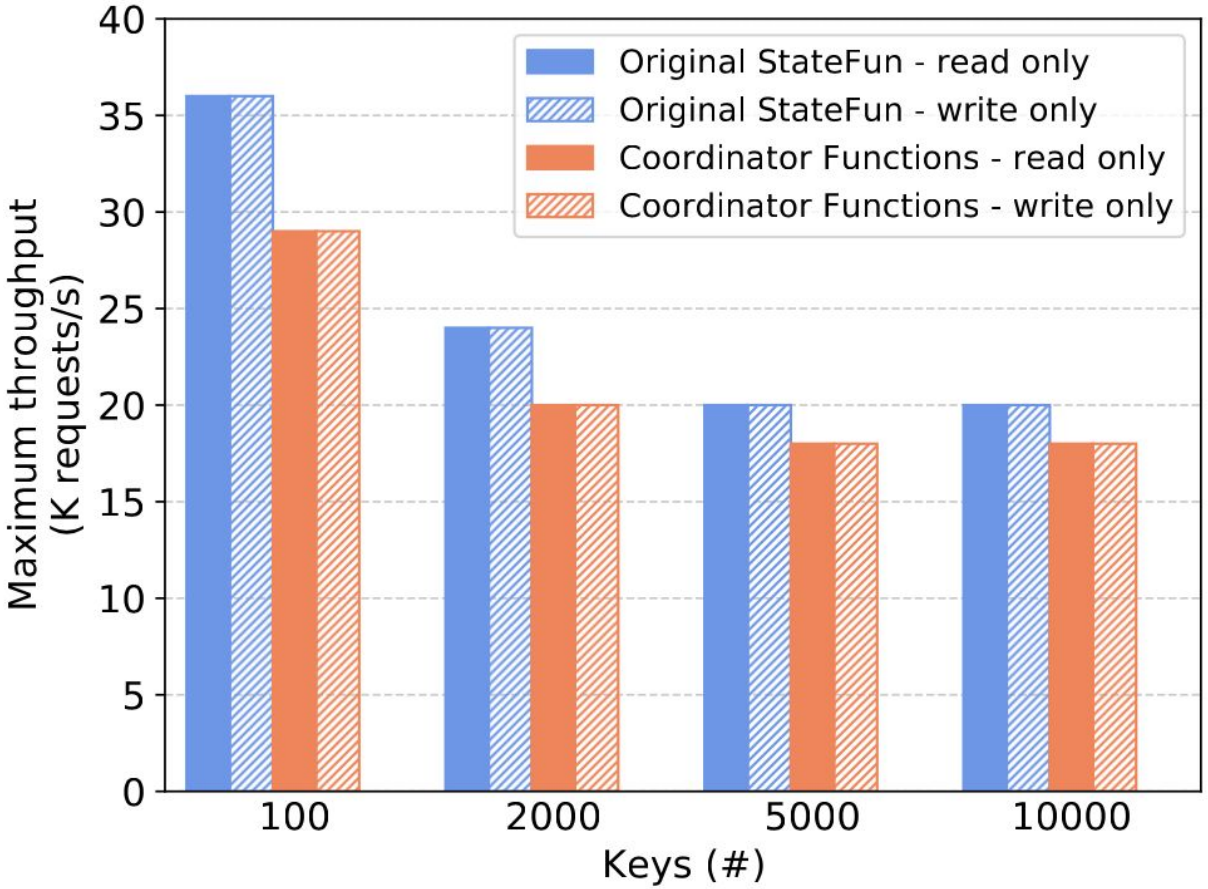


Evaluation



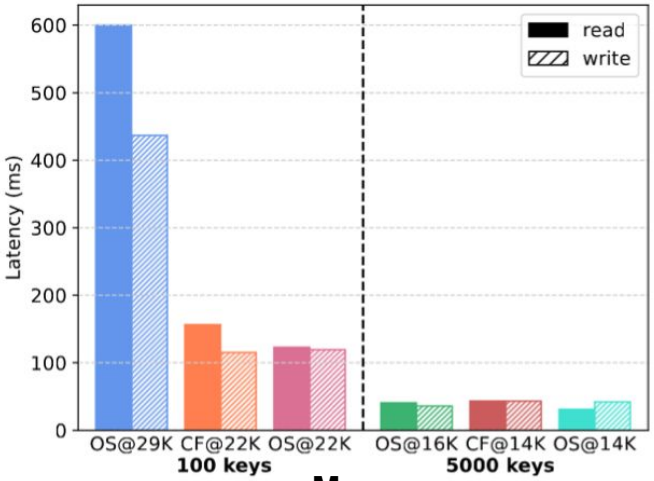
Statefun is restricted to a number of instances and CPUs other components are scaled so they are not the bottleneck

Evaluation

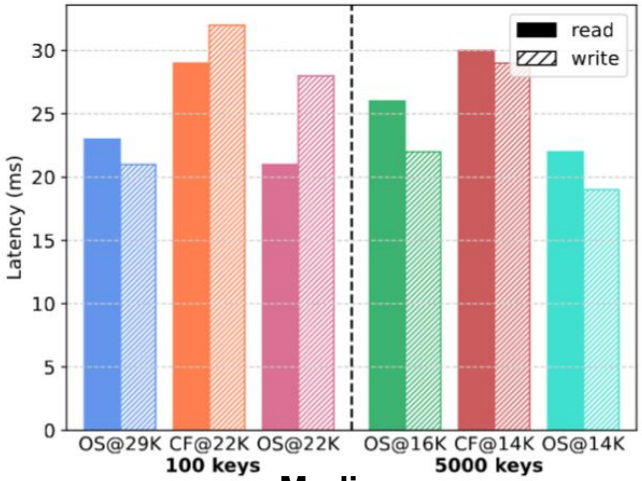


~20% drop in performance for 100 keys
~10% drop in performance for 5000+ keys

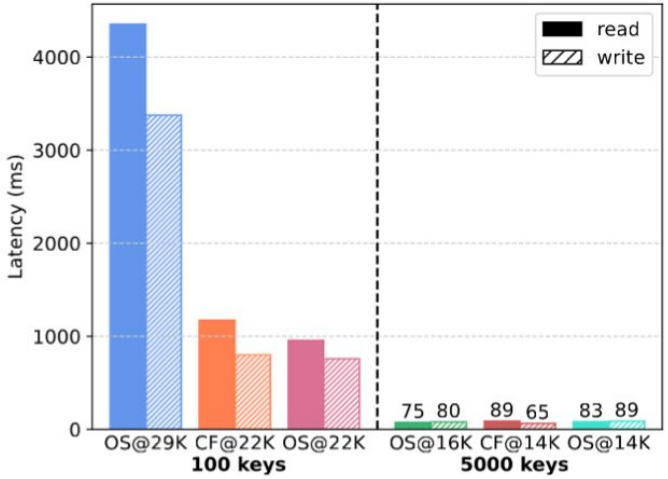
Evaluation



Mean

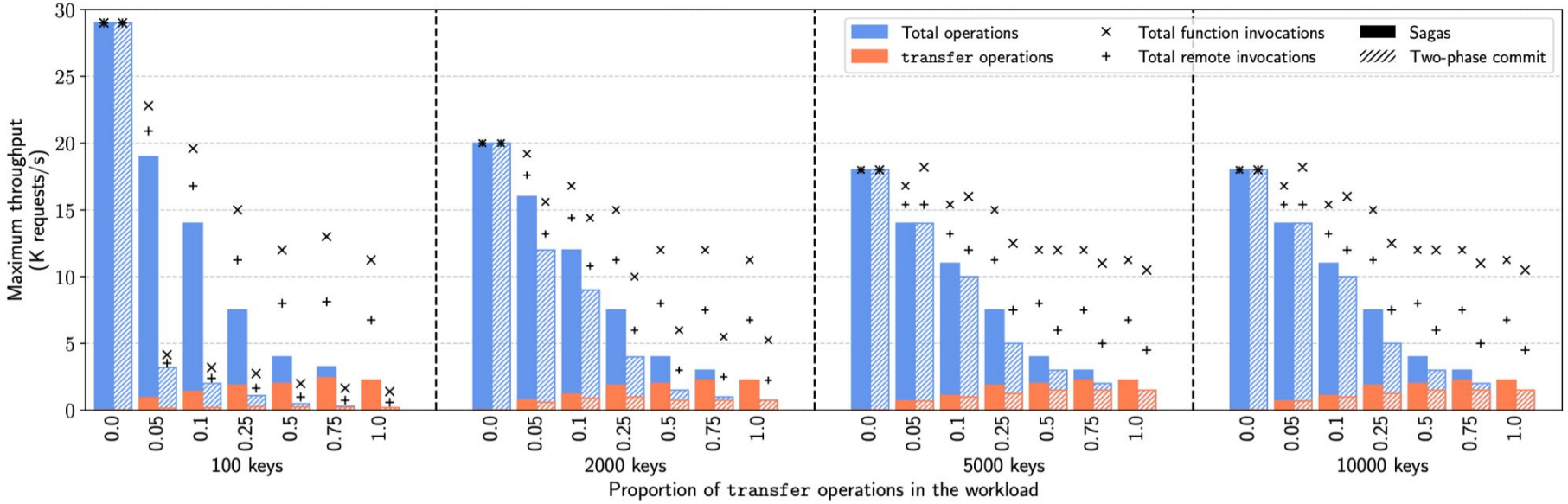


Median



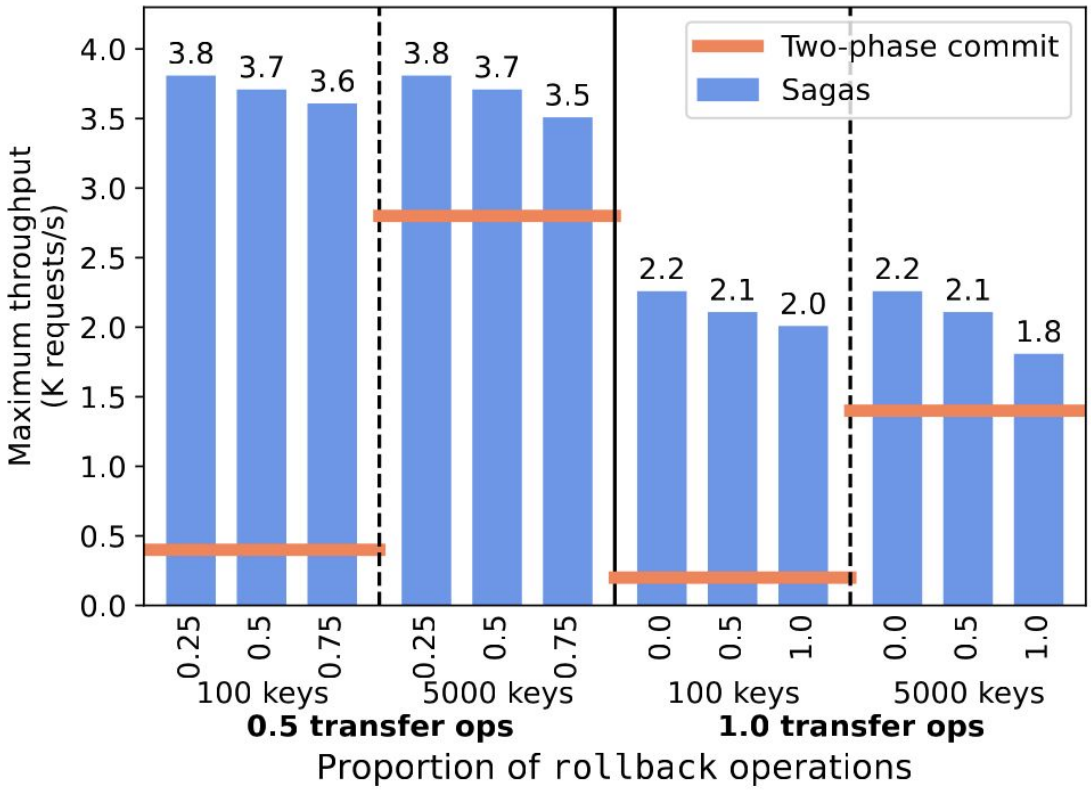
95th percentile

Evaluation



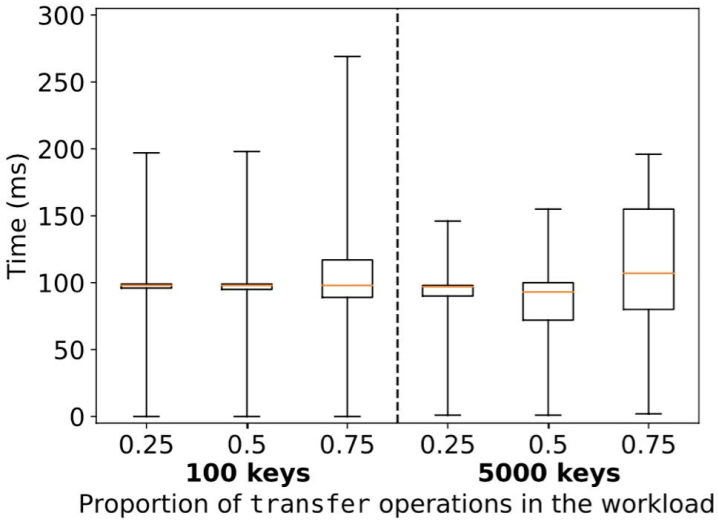
Very poor performance for serializable transaction across 100 keys
More comparable performance for 5000+ keys

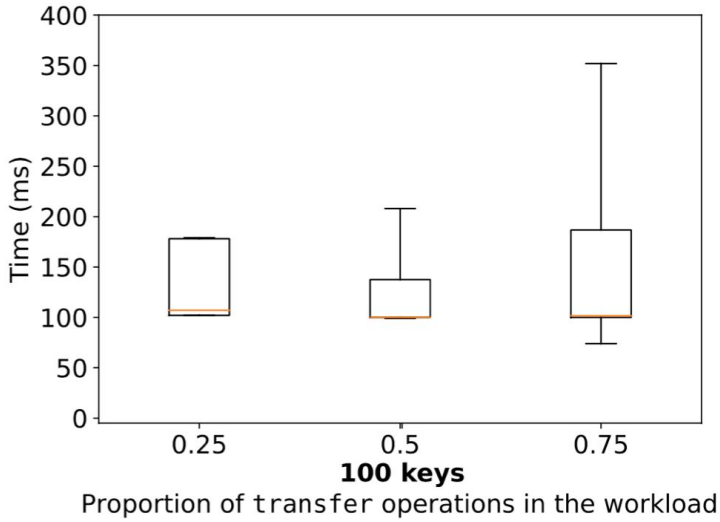
Evaluation



Rollbacks have some overhead for sagas
Sagas still perform better than serializable transactions even at 100% rollbacks

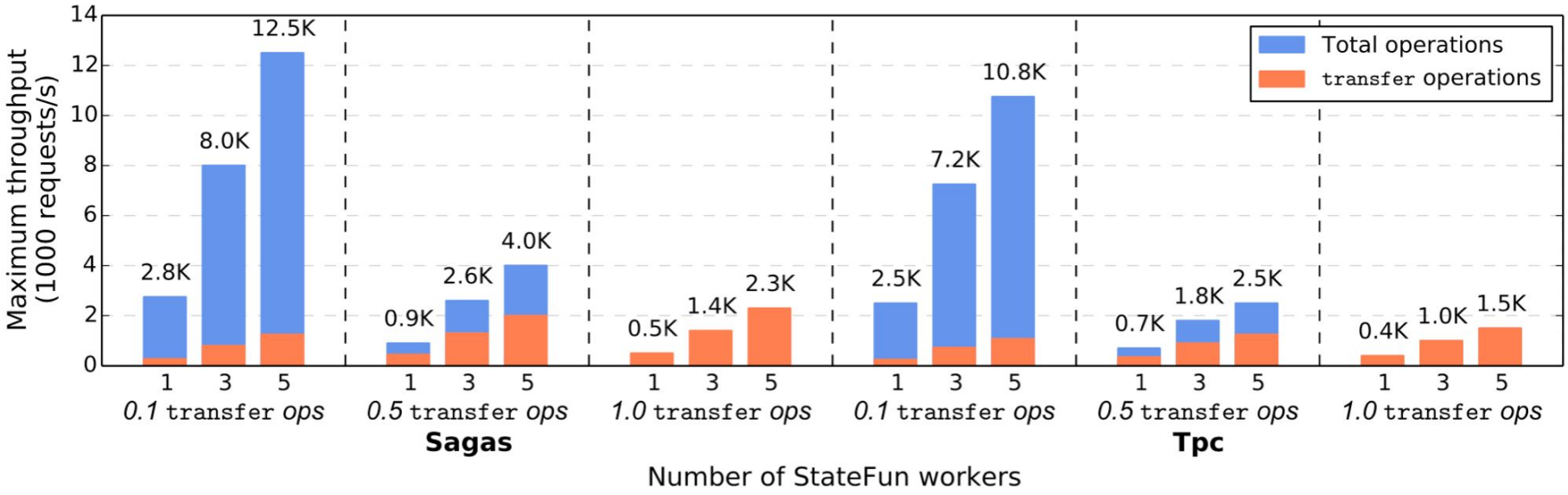
Evaluation





Keys	Transfer proportion	Deadlocks / `transfer` ops
100	0.25	9/12014 (0.07%)
	0.5	27/24107 (0.11%)
	0.75	82/35875 (0.22%)
5000	0.25	0/60121
	0.5	0/120089
	0.75	0/179794

Evaluation



Consistent **90%** scalability efficiency for **sagas**

87% scalability efficiency for **serializable transactions** at **10% transfers**
75% scalability efficiency for **serializable transactions** at **100% transfers**

Conclusion

- Simple programming model abstracting away all distributed systems concerns from the application developer
- Acceptable performance overhead on non-transactional workloads
- Good scalability for saga-based transactions and acceptable scalability for serializable transactions

Future work

Make Stateful Functions complete

- State access without specific function ID
  - Secondary indices
  - WHERE clause
- Aggregates over state across function instances
- Versioning of state / updating functions
- Research based on use cases developing improved benchmarks

Distributed Transactions on Serverless Stateful Functions

Martijn de Heus
4367839
Web Information Systems

TU Delft