Ditching the Data Center: How to Stop Worrying and Love the Edge

Peter Van Roy
Université catholique de Louvain
Coordinator, LightKone project

June 8-9, 2017
Erlang User Conference
Stockholm, Sweden

LIGHTKONE
Lightweight computation for networks at the edge

Context: LightKone and SyncFree

LIGHTKONE

Lightweight computation for networks at the edge

- **LightKone H2020 project (2017-2019)** lightkone.eu
  - Lightweight computation for networks at the edge
  - Partners: UCL, UPMC/INRIA, INESC TEC, TUKL, NOVA ID, Scality, Gluk, UPC/Guifi, Stritzinger

SYNCFREE

- **SyncFree FP7 project (2013-2016)** syncfree.lip6.fr
  - Large-scale computation without synchronisation
  - Partners: INRIA, Basho, Trifork, Rovio, UNL, UCL, Koç, TUKL

Greed – for lack of a better word – is *good*!

– Gordon Gekko, *Wall Street* (1987)

~~Greed~~ – for lack of a better word – is *good*!

Failure

– Gordon Gekko, *Wall Street* (1987)

Overview

- “Failure is good” philosophy
- Convergent computation
  - Lasp language
  - Experimenting with Lasp
- How does it work? … a little bit of theory
  - Convergent consistency
  - Convergent data structures: CRDTs
  - Hybrid gossip communication
- Where we are going
  - Lasp today and tomorrow
  - Synchronization-free services for edge computing

LIGHTKONE
Lightweight computation for networks at the edge

Failure is good

- How can failure be good?
  - Nodes go offline to save power (low-power systems)
  - Networks go down or partition (low-power systems)
  - Using low-cost nodes that fail under stress (heat) reduces costs
  - Networks grow and shrink, nodes come and go
  - Hardware and software upgrades diffuse through the system
  - Software rejuvenation (periodic restart) … like living organisms!
  - And we are not yet talking about crashes and software bugs!
- Don’t fight failure, accept it as normal
  - Use computation and communication models that live with failure

Living with failure

- Convergent computation
  - New information generalizes past information
  - Computation is always converging to a result
  - Dropped, delayed, or reordered messages are ok
  - Very little synchronization between nodes is needed

- Hybrid gossip communication
  - Efficient communication in dynamic networks
  - Nodes keep track of neighbors as network changes
  - Communication keeps working even if most nodes fail

Natural match

- Convergent computation is naturally tolerant to node and communication problems on edge networks
  - Network partitions
  - Message loss and reordering

May slow down convergence;
no other error is possible

  - Nodes going offline and online
  - Nodes crash

Correct as long as state
exists on at least one node

The big question

- Convergent computation looks very nice
- But!
  - Can it be implemented efficiently?
  - Is it easy to program?
- As we will see, the answer to both questions is a big yes
  - Our research initially went in the opposite direction: we started by investigating programming with weak synchronization models and we arrived at convergent computation

Convergent data is used today

- Many companies are already using convergent data structures
  - The following companies are using CRDTs
  - Convergent computation is still a research topic

Tapjoy LEAGUE of LEGENDS TOMTOM bet365 akka RIOT GAMES ROVIO

basho TRIFORK. ...think software SCALITY SOUNDCLOUD riak HELIUM

Convergent computation
(Lasp)

LIGHTKONE

Lightweight computation for networks at the edge

Simple Lasp program

- Lasp is a programming language for writing convergent computations
- Let us declare two sets and connect them with a map (code fragment using Erlang syntax)

```
S1=declare(set),
bind(S1, {add, [1,2,3]}),
S2=declare(set),
map(S1, fun(X) -> X*2 end, S2).
```

How Lasp executes

- A Lasp program is a graph of data structures connected by operations
  - Data structures S1 and S2 are passive
  - Operation Map is an active process
- If you update S1 by removing an element, the map will update S2 by removing a mapped element
  - Execution is functional dataflow programming
- This looks simple, but there is more than meets the eye!

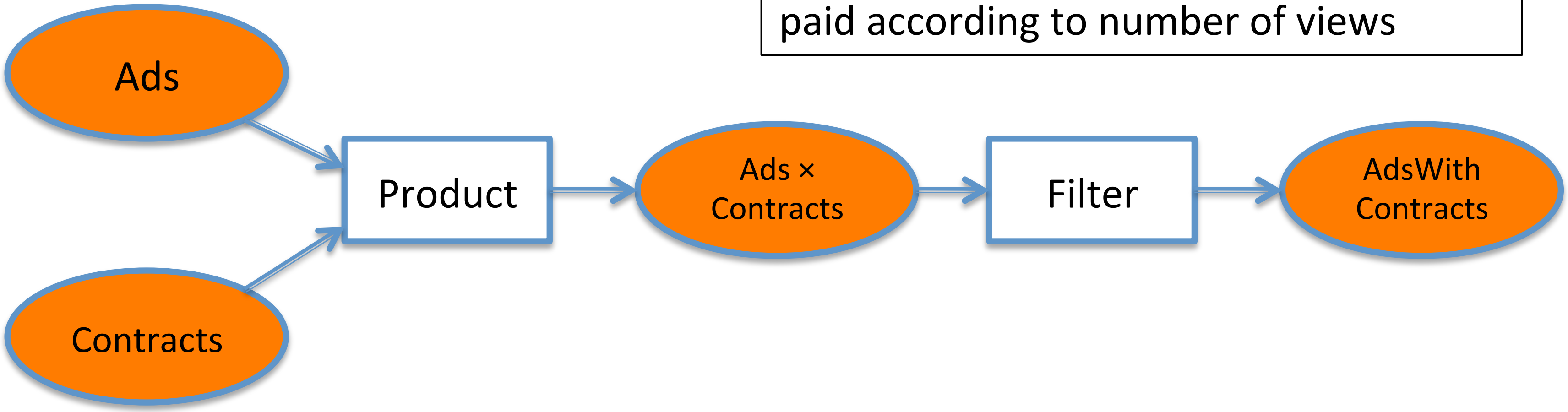
Lasp is convergent

- Lasp data structures are designed to be fault tolerant
  - They are replicated and maintain consistency between replicas
- Lasp data structures are designed to be convergent
  - An update to one replica will eventually propagate to all
- Lasp programs are also fault tolerant and convergent
  - The operations are built to guarantee this

- How does it work? It uses clever implementation techniques supported by a mathematical theory!

Lasp program: Ad counter

Ad counter: clients view advertisements, paid according to number of views



All four data structures are sets

Two processes Product and Filter

- Ads={ad(id:I counter:C), …}
  Contracts={contracts(id:I), …}

  product(Ads, Contracts, AdsContracts)

  F=fun (A C) -> A.id==C.id end
  filter(AdsContracts, F, AdsWithContracts)

Experimenting with Lasp

- Our Lasp prototype is implemented as a set of Erlang libraries
  - We invite you to try it out!
  - No special syntax yet
  - We are actively working on improving it
- Documentation
  - https://lasp-lang.org
- Code repository
  - https://github.com/lasp-lang

How does it work?
… a little bit of theory

LIGHTKONE

Lightweight computation for networks at the edge

A little bit of theory

- Convergent consistency
  - Rethinking how to build distributed systems
  - Beyond eventual consistency
- Convergent data structures
  - CRDTs and join semilattices
  - Theorem: CRDTs satisfy convergent consistency
  - Theorem: Lasp programs satisfy convergent consistency
- Hybrid gossip communication
  - Keeping the system connected even at high churn

Convergent consistency

Both easy and efficient

- One of the holy grails of distributed systems is to make them both easy to program and efficient to execute
- Strong consistency (linearizability) is easy to program with but inefficient
- Eventual consistency (operations eventually complete) is efficient to execute but hard to program with
- Can we get the best of both worlds?
  - Convergent consistency aims to combine the ease of strong consistency with the efficiency of eventual consistency
  - How can this work?

LIGHTKONE
Lightweight computation for networks at the edge

Back to basics

- A distributed system is a collection of networked computing nodes that behaves like one system (= consistency model)
- To make this work, the nodes will coordinate with each other according to well-defined rules (= synchronization algorithm)
- For example, a reliable broadcast algorithm guarantees the all-or-none property: all correct nodes deliver the message, or none do

How far can we go?

- We would like:
  - the consistency model to be as **strong** as possible (easy to program), and
  - the synchronization algorithm to be as *weak* as possible (efficient to execute)
- Let's try the extreme case: the weakest possible synchronization is no synchronization (no rules), which enforces no consistency at all!
  - So it's clear we need *some* synchronization
  - How little can we get away with?

LIGHTKONE
Lightweight computation for networks at the edge

A sweet spot: SEC

- Strong Eventual Consistency (SEC)
  - The data structure is defined so that n replicas that receive the same updates (in any order) have equivalent state
  - The only synchronization algorithm we need is eventual replica-to-replica communication

- This consistency model is surprisingly powerful
  - It supports a programming model that resembles a concurrent form of functional programming
  - It supports both nondeterminism and nonmonotonicity
  - It has an efficient, resilient implementation

Consistency models redux

- Strong consistency: the system obeys linearizability
  - Easy to program but often inefficient because of synchronization
- Eventual consistency: the system can support many concurrent operations « in flight »
  - Efficient execution but hard to program because of potential conflicts
- **Convergent consistency**: eventual consistency **plus SEC**
  - Both efficient execution and easy to program

- Not CAP but AP + ◇C = available, partition-tolerant, convergent

Convergent data structures and a couple of theorems

LIGHTKONE

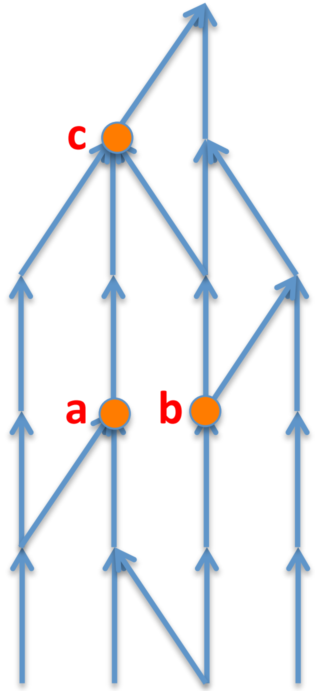
Lightweight computation for networks at the edge

CRDT definition

- How can we build convergent data structures?
  - One way is the CRDT: Conflict-free Replicated Data Type
- A *state-based CRDT* is defined as a triple $((s_1, \ldots, s_n),m,q)$:
  - $(s_1, \ldots, s_n)$ is the configuration on n replicas, with $s_i \in S$ where S is a join semilattice
  - $q_i:S \longrightarrow V$ is a query function (read operation)
  - $m_i:S \longrightarrow S$ is a mutator (update operation) such that $s \sqsubseteq m_i(s)$
  - Periodically, replicas update each other's state: $\forall i,j: s_i' = s_i \sqcup s_j$ (join)
- Why is this convergent?
  - Because the mutator always inflates the value and the periodic updates always merge using the join operation
  - This ensures that all replicas will eventually converge to the same value

Join semilattice

- The state-based CRDT is convergent because it is based on a join semilattice
- A *join semilattice* is a partially ordered set S that has a least upper bound (join) for any nonempty finite subset:
  - Partial order: ∀x, y, z ∈ S:
    Reflexivity: $x \sqsubseteq x$
    Antisymmetry: $x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x=y$
    Transitivity: $x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
  - Least upper bound (join): ∀x, y ∈ S: $x \sqcup y \in S$
    - $z = x \sqcup y$ is an upper bound
    - All other upper bounds are at least as large as z



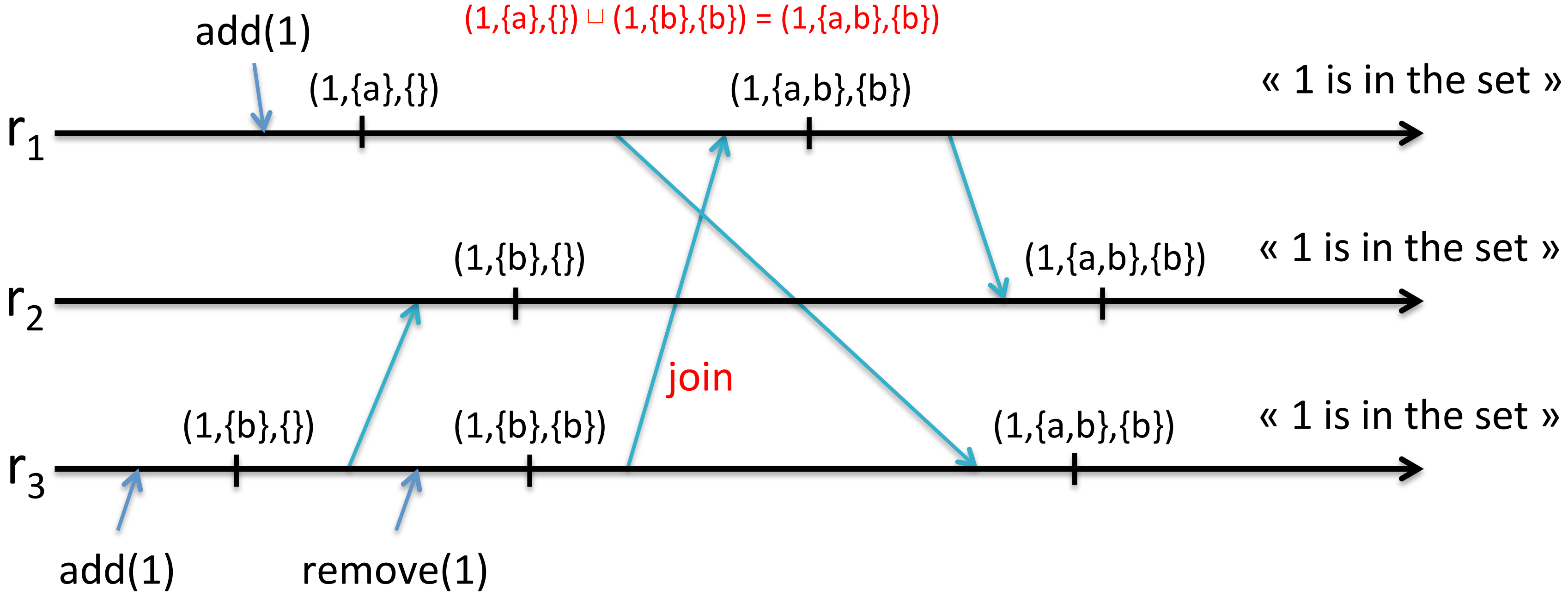
$c = a \sqcup b$

LIGHTKONE
Lightweight computation for networks at the edge

Observed-Remove Set (1)

- Many CRDTs exist, too many to present in this talk
  - We will show just one: the OR-Set
- The OR-Set supports both adding and removing elements
  - The outcome of a sequence of adds and removes conforms to the sequential specification of a set
    - In a distributed system it's more general: a remove will remove all elements in its causal history
  - In case of concurrent add and remove, the add has precedence
- The intuition is to uniquely tag each added element
  - The tag is not exposed when querying the set content
  - When removing an element, all tags are removed

Observed-Remove Set (2)



- Each replica stores triples (e,A,R) where e is the element, A is the set of adds and R is the set of removes
- If (e,A,R) with A-R≠{} then e is in the set
  - All updates (both adds and removes) cause monotonic increases in (e,A,R)

LIGHTKONE
Lightweight computation for networks at the edge

Theorem:
A state-based CRDT satisfies SEC

- Strong Eventual Consistency (SEC)
  - We assume eventual delivery: an update delivered at some correct replica is eventually delivered to all correct replicas
    - Eventual replica-to-replica communication satisfies this
  - An object is SEC if all correct replicas that have delivered the same updates have equivalent state

- Theorem: A state-based CRDT satisfies SEC
  - Proof by induction on the causal histories of deliveries at the replicas
  - Proof given in INRIA Research Report RR-7687

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski.
Conflict-free replicated data types. Technical Report RR-7687. INRIA (July 2011).

Theorem:
A simple Lasp program satisfies SEC

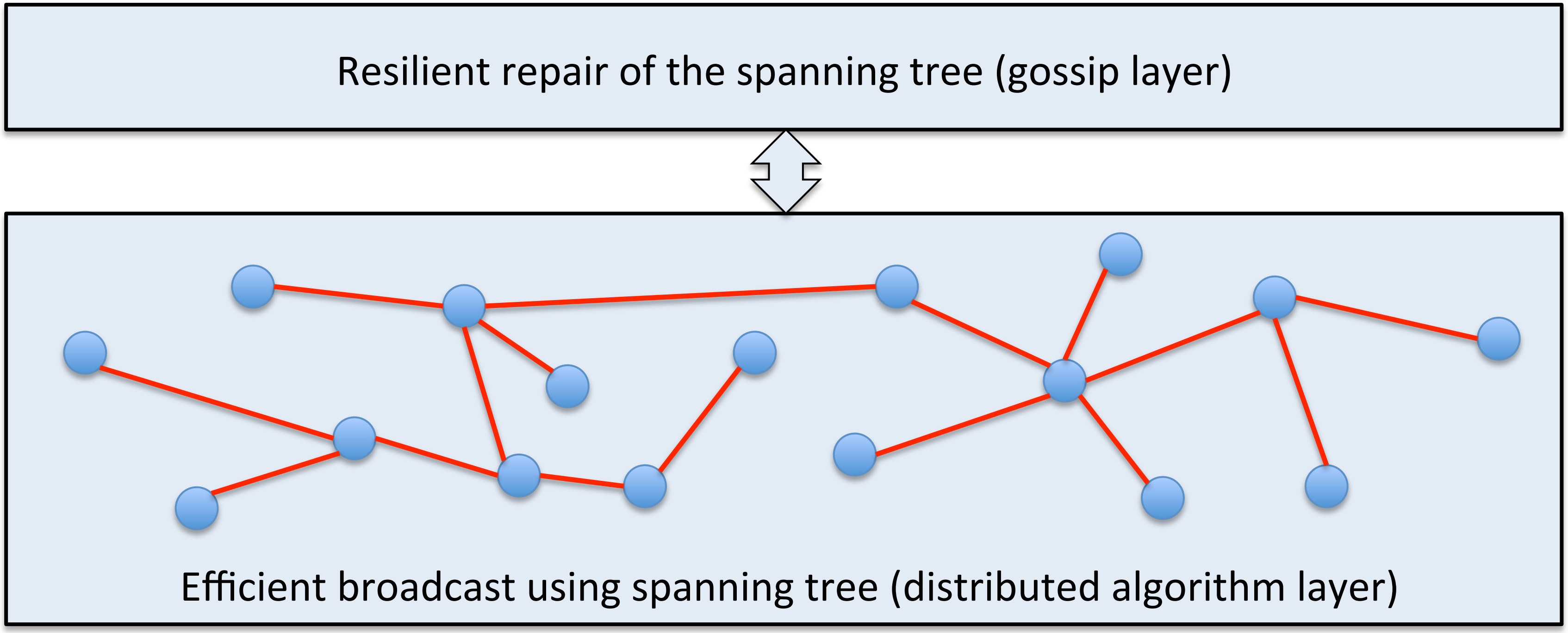
- Fault model: crash-stop, at least one replica correct
- Simple Lasp program: single CRDT instance or a Lasp process with inputs from simple Lasp programs (directed acyclic graph where node = CRDT instance, edge = Lasp process)

- Theorem: A simple Lasp program satisfies SEC
  - Proof by induction on the program graph: Lasp program execution can be reduced to a state-based CRDT execution
  - Proof given in Lasp paper published in PPDP 2015 conference

Christopher Meiklejohn and Peter Van Roy. Lasp: A language for distributed, coordination-free programming. In *Principles and Practice of Declarative Programming (PPDP 2015)*. ACM, 184–195 (July 2015).

Hybrid gossip communication

Hybrid gossip algorithms



Resilient repair of the spanning tree (gossip layer)

Efficient broadcast using spanning tree (distributed algorithm layer)

- Hybrid gossip algorithms use two layers (Plumtree shown)
  - Efficient (but fragile) broadcast using spanning tree
  - Resilient (but slow) repair of spanning tree using gossip
- The combination is both efficient and resilient

Lasp communication layer

- Lasp uses a membership protocol that is a modified version of the HyParView hybrid gossip algorithm
  - HyParView maintains a connected network with high reliability (almost 100% connectivity) even when up to 90% of nodes fail
  - HyParView stores at each node an active view and a passive view of its neighbors. The small active view maintains open connections; the larger passive view is used to increase reliability.
  - Lasp modifies HyParView to work with high churn
- The connected network is used for periodic state updates between replicas

João Leitão, José Pereira, and Luís Rodrigues. *HyParView: A membership protocol for reliable gossip-based broadcast*. Technical Report TR-07-13, Universidade de Lisboa (May 2007).

LIGHTKONE
Lightweight computation for networks at the edge

Where we are going

Today and tomorrow

Today

- We are using convergent consistency, with CRDTs and Lasp, as the basis for a programming system and a transactional database
    - Lasp language, as shown in this talk
    - Antidote database, not presented in this talk
- We are applying the approach to edge computing (Internet of Things) in the LightKone project
    - Taking nontrivial computations (analytics, machine learning) out of the data center and executing them *directly on the edge*

Tomorrow

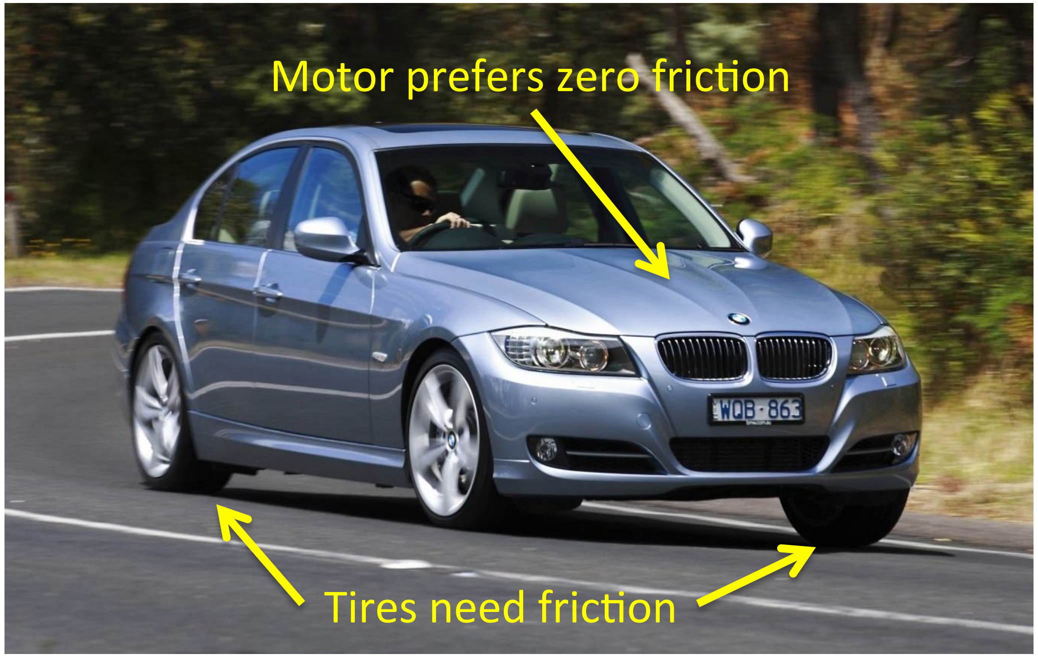
- But convergent consistency is much more
    - To explain, let me tell a story about synchronization …

LIGHTKONE

Lightweight computation for networks at the edge

Parable of the car (1)

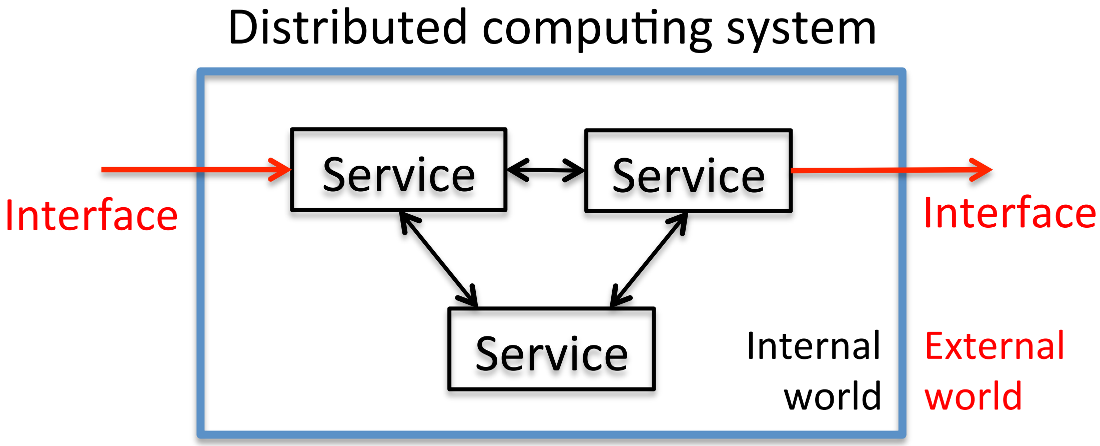
Synchronization is like friction



- Like friction, synchronization is both desirable and undesirable
- Consider a car on a highway
- The car needs friction: it moves because the tires grip the road
- But the car's motor avoids friction: the motor should be as frictionless as possible, otherwise it will heat up and wear out

Parable of the car (2)

Consider a distributed computing system made of services connected together



- Synchronization is only needed at the interface with the external world
- Internally, services avoid synchronization (they use convergent computation)

Friction is only needed externally, so the tires can grip the road

Internally, the motor avoids friction

LIGHTKONE
Lightweight computation for networks at the edge

Synchronization-free services

- The system has a synchronization boundary
  - Inside the boundary, all services use weak synchronization
  - Strong synchronization is only needed at the boundary

- Services are inside the boundary
  - Each service does convergent computation
  - Service API has asynchronous streams, in and out

LIGHTKONE
Lightweight computation for networks at the edge

Conclusion

- We have introduced convergent consistency and programming with weak synchronization
  - We presented data structures (CRDTs) and a programming language (Lasp) for convergent computation

- Our current work is focused on edge computing and synchronization-free services
  - LightKone H2020 project (lightkone.eu)
  - The project uses Lasp and Antidote as starting points

LIGHTKONE

Lightweight computation for networks at the edge