Elements of a unified semantics for synchronization-free programming based on Lasp and Antidote

March 1, 2018
Dagstuhl seminar 18091

Peter Van Roy
Université catholique de Louvain

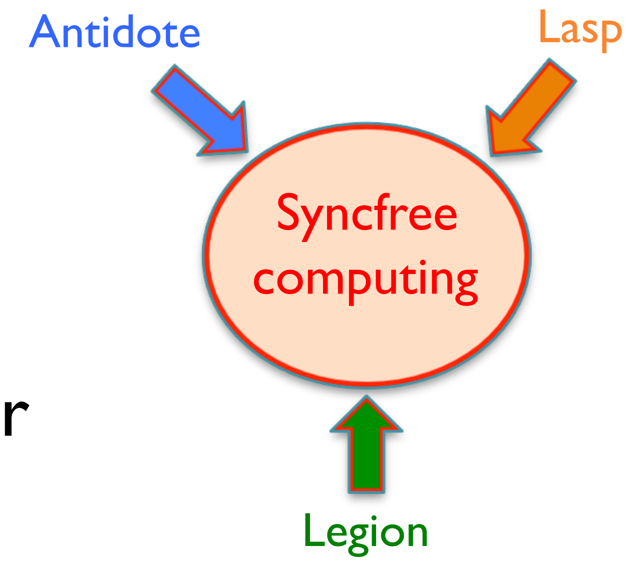
Work in progress; inspired by the document by Peter Zeller, Annette Bieniusa, Mathias Weber, Christopher Meiklejohn, Peter Van Roy, Nuno Preguiça, and Carla Ferreira

LightKone and SyncFree projects

- **LightKone H2020 project (2017-2019)**
  lightkone.eu
  - Lightweight computation for networks at the edge
  - Partners: UCL, UPMC/INRIA, INESC TEC/UMinho, TUKL, NOVA ID/UNL, Scality, Gluk, UPC/Guifi, Stritzinger
- **SyncFree FP7 project (2013-2016)**
  syncfree.lip6.fr
  - Large-scale computation without synchronisation
  - Partners: INRIA, Basho, Trifork, Rovio, UNL, UCL, Koç, TUKL

Three systems from SyncFree

- Lasp provides dataflow composition of CRDTs
- Antidote provides causal transactional CRDT storage
- Legion provides peer-to-peer CRDT interaction between clients



⇒ Each explores a different part of the space

LIGHTKONE

There can be only one!

– Connor MacLeod, *Highlander* (1986)

There can be only one semantics! (*)

– Prof. Dr. Ir. Connor MacLeod, *Hochländer* (1986)

(*) **Es kann nur eine Semantik geben!**

LIGHTKONE

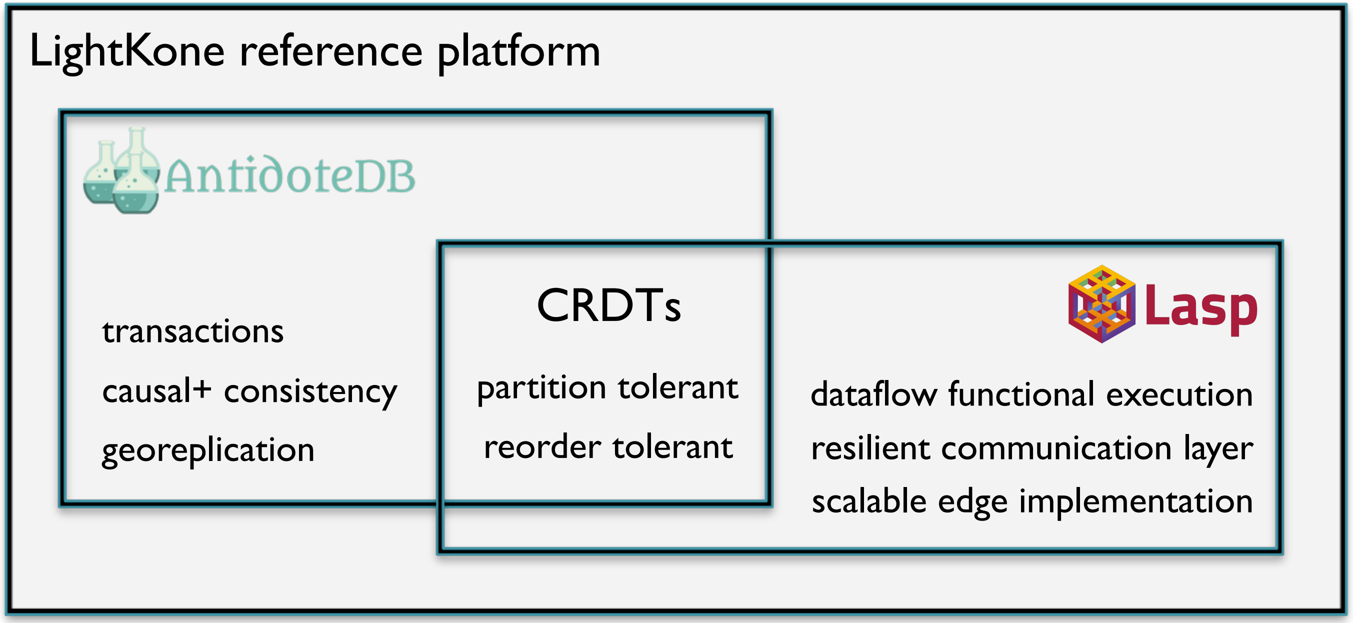
Lasp and Antidote

- Lasp
  - Deterministic dataflow functional semantics
  - Graph of CRDTs connected by operations
  - Resilient communication with hybrid gossip targeting unreliable networks (e.g., edge networks)
- Antidote
  - Georeplicated data store with low latency and high availability
  - Transactional causal+ consistency on CRDTs
- Both based on CRDTs
  - Both provide consistency with weak synchronization
  - Both tolerate partitioning and message reordering

Combining Lasp and Antidote

- Both are distributed programming models based on weak synchronization
- Lasp and Antidote were invented separately
  - Both use CRDTs as their data structures
  - Both provide important functionality
  - But they have very different implementations
- We would like to combine them
  - Define one semantics that can express both
  - Allow the implementations to interoperate correctly

The LightKone reference platform



- Reference platform defined by the unified semantics
- Antidote and Lasp are partial implementations

ABSTRACT EXECUTIONS

Abstract executions

- We describe systems in terms of events and their visibility
  - This defines observable behavior between clients and the system
  - An abstract execution is an event graph that satisfies certain correctness conditions that we explain in the next two slides
    - For full definitions see S. Burckhardt, *Principles of Eventual Consistency*, 2014
- Event $e \in E$: uniquely identifies objects and their operations
  - Key: $key(e) \in Keys$
    - Objects are uniquely identified by their key k
  - Operation: $op(e) \in Ops$
  - Result value: $res(e) \in V$
- Visibility relation $vis \subset E \times E$: defines what events can see
  - We write $e_1 <_{vis} e_2$ when $(e_1, e_2) \in vis$
  - e_1 can be observed by e_2
- Arbitration relation $ar \subset E \times E$: breaks ties for concurrency

LIGHTKONE

Data types

- Each data type T is defined by a function F_T
  - Each object k has a type defined by type(k)
- Value of an object is defined for each event e
  - Value depends on e's *context*, i.e., all the object's events that are visible to e (we do not represent the object state explicitly)
- Context $c=ctxt(e) = (E', op_{|E'}, vis_{|E'}, ar_{|E'})$ where $E'=\{e' \in E \mid e' \prec_{vis} e\}$
  - We can restrict the context to key k: $c_{|k}= (E, op, vis, ar)_{|k} = (E', op_{|E'}, vis_{|E'}, ar_{|E'})$ where $E'=\{e \in E \mid key(e)=k\}$
- Value $v= F_{type(key(e))}(ctxt(e)_{|key(e)}) \in V$

LIGHTKONE

Correct execution

- A correct execution satisfies the conditions:
  - Acyclic visibility: no cycles in vis
  - Total arbitration: ar is a total order
  - Per-object eventual consistency
    - All of an object's events are seen by all other events on that object (except for a finite number)
    - For all keys k: $\forall e \in E_k . \{e' \in E_k \mid e \not<_{vis} e'\}$ is finite
      where $E_k = \{e \mid key(e)=k\}$
  - Correct results (definition of res)
    - $\forall e \in E.\ res(e) = F_{type(key(e))}(ctxt(e)_{|key(e)})$
  - Causality
    - Per-object causal consistency: $\forall k$: $vis_{|Ek}$ is transitive
    - Causal consistency: vis is transitive

LIGHTKONE

LASP SEMANTICS

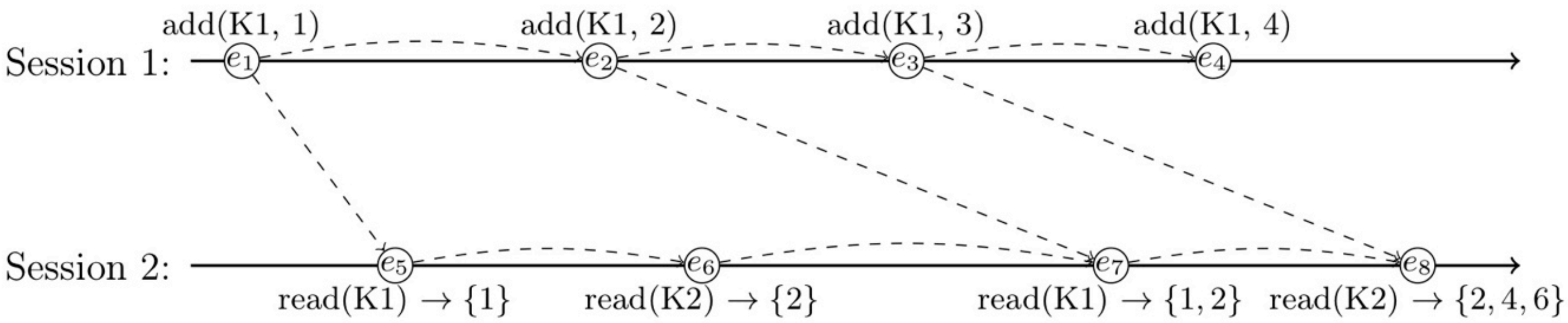
Lasp

- Sets connected with a map:


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

- Deterministic dataflow functional semantics
  - Graph of CRDTs connected by operations
  - Operations: Map, filter, fold, product, intersect, union, join
- Efficient resilient implementation
  - Ensures consistency with weak synchronization
  - Tolerates node and communication failures
  - Uses a communication layer based on hybrid gossip

Example Lasp program



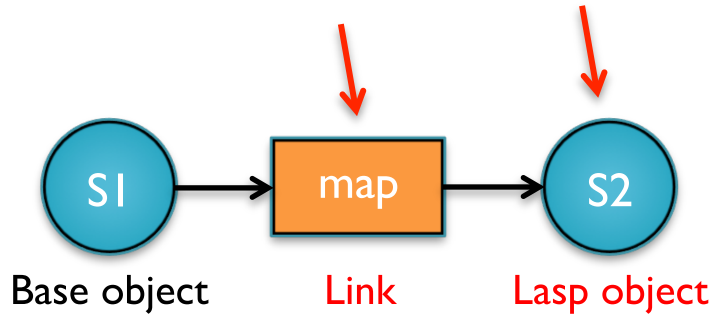
- Consider a Lasp program with two objects k_1 and k_2 and a map between them:

```
K1 = declare(set),
K2 = declare(set),
map(K1, fun(X) -> X*2 end, K2).
```

- Let's calculate $res(e_8) = \{2,4,6\}$
  - Set of visible events for e_8: $E' = \{e_1, e_2, e_3, e_5, e_6, e_7\}$
  - $res(e_8) = R(k_2, ctxt(e_8)) = (\lambda S \rightarrow \{x \cdot 2 \mid x \in V\})(R(k_1, ctxt(e_8))$ where $R(k_1, ctxt(e_8)) = F_{aw\text{-}set}(ctxt(e_8)) = \{1,2,3\}$
  - $res(e_8) = R(k_2, ctxt(e_8)) = \{2,4,6\}$

LIGHTKONE

Lasp semantics



- To specify Lasp semantics, we add two concepts:
  - Lasp objects and links
- Lasp object: we partition the key space into base objects and Lasp objects
  - LaspKeys ⊂ Keys
  - Base objects have both read and update events, whereas Lasp objects have only read events
- Link: Each Lasp object k is *linked* from n objects
  - link(k)=([k_1, …, k_n], f)
  - The function f defines the read operation on k, which depends on k_1, …, k_n

LIGHTKONE

Lasp operations

- Lasp operations are defined by their links
  - Each Lasp operation has its own link
  - On this slide, we assume all objects have set values
- Lasp (as defined in PPDP 2015 (*)) provides:
  - Map: ([k], λ V→{f(x) | x∈V})
  - Product: ([k_1,k_2], λ V_1,V_2→(V_1×V_2))
  - Intersection: ([k_1,k_2], λ V_1,V_2→($V_1 \cap V_2$))
  - Union: ([k_1,k_2], λ V_1,V_2→($V_1 \cup V_2$))
  - Filter: ([k], λ V→{x | x∈V∧P(x)})
  - Fold: ([k], $fold_{f,z}$) where

    $fold_{f,z}${}=z and $fold_{f,z}$({x}∪V)=f(x, $fold_{f,z}$(V))

(*) 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).

LIGHTKONE

Eventual consistency of linked objects

- If a Lasp object k_1 depends on a base object k_2, then there is eventual consistency between the two objects
- First define all the objects that a Lasp object depends on (dependsOn function):
  - There are direct dependencies and transitive dependencies
  - If $link(k) = ([k_1, \ldots, k_n], f)$ then $\{k_1, \ldots, k_n\} \subseteq dependsOn(k)$
  - If $k_a \in dependsOn(k_b)$ and $k_b \in dependsOn(k_c)$ then $k_a \in dependsOn(k_c)$
- Then all base events e are seen by all but a finite number of dependent Lasp events e':
  - $\forall e \in E. \{e' \in E \mid key(e) \in dependsOn(key(e')) \land e \not\prec_{vis} e'\}$ is finite
  - This definition is similar to eventual consistency on one object, but here it concerns two objects

LIGHTKONE

Reading from Lasp objects (1)

- Base objects can be read and updated
  - The value of a base object at event e is defined by the *context* of e: all events that are visible to e
  - The value can be updated because the context depends on e
- Lasp objects can only be read
  - Value of a Lasp object e is defined by the *link*, which defines a function of the base objects that the Lasp object depends on
  - No update is possible on e since the value does not depend on the context of e

LIGHTKONE

Reading from Lasp objects (2)

- Result value is written res(e) for event e
  - Event e can be for a base object or a Lasp object
  - We assume res(e)=R(key(e), ctxt(e)) with R as follows

- Read from base objects
  - For base objects, R is defined by F_{type} definition
  - $R(k, c) = F_{type(k)}(c_{|k})$
- Read from Lasp objects
  - For Lasp objects, R is defined by the link
  - Assume that $link(k)=([k_1, \ldots, k_n],f)$
  - $R(k, c) = f(R(k_1,c), \ldots, R(k_n,c))$

LIGHTKONE

CONVERGENT CONSISTENCY (WORK IN PROGRESS)

From eventual to convergent (1)

- So far we have defined eventual consistency for single objects and for linked (Lasp) objects
- Eventual consistency for single objects
  - All events e are seen by all but finite number of events e' on the same object
  - $\forall e \in E: \{e' \in E \mid key(e)=key(e') \wedge e \not<_{vis} e'\}$ is finite
- Eventual consistency for linked objects
  - Base events e are seen by all but finite number of dependent Lasp events e'
  - $\forall e \in E: \{e' \in E \mid key(e) \in dependsOn(key(e')) \wedge e \not<_{vis} e'\}$ is finite
- But CRDTs do more than eventual consistency!

LIGHTKONE

From eventual to convergent (2)

- Eventual consistency leaves out a key property of CRDT and Lasp execution
  - Eventual consistency says only that every event will be taken into account always after a sufficiently long time, but there is a finite interval during which the event can have erratic visibility
  - In CRDTs and Lasp, computations are always based on a strictly growing set of events (once added, an event is never forgotten)

- Lasp computations are always converging to the result
  - Every update eventually appears on all replicas
  - Each replica has a strictly growing set of updates
  - This is a monotonicity property

Convergent consistency

- Consider the definition of monotonic reads
  - $\forall e_1, e_2, e_3 \in E: e_1 \prec_{vis} e_2 \land e_2 \prec_{so} e_3 \Rightarrow e_1 \prec_{vis} e_3$
  - A (read) event once visible in a session is always visible in the session
- Convergent consistency between two objects
  - An event e of object k_1 once visible to object k_2 is always visible to k_2
  - $\forall e \in E_{k1}, \forall e', e'' \in E_{k2} : e \prec_{vis} e' \land e' \prec_{vis} e'' \Rightarrow e \prec_{vis} e''$
- Convergent consistency for Lasp objects
  - Add the condition $k_1 \in dependsOn(k_2)$
  - If a base event is seen by a dependent Lasp event, then it is seen by all further events of the same Lasp object

LIGHTKONE

Convergence and CRDTs

- Convergent consistency
  - Each event adds information permanently in a single step
- Strong eventual consistency
  - n replicas that receive the same updates (in any order) have equivalent state
  - A state-based CRDT satisfies SEC
  - An acyclic Lasp program satisfies SEC
- State-based CRDTs
  - State-based CRDT ensures SEC and CC

LIGHTKONE

ANTIDOTE SEMANTICS

Antidote semantics

- Antidote provides the following guarantees
  - Acyclic visibility, total arbitration, eventual consistency
  - Causal consistency
  - Atomic visibility
  - Min snapshot
- Antidote provides a series of datatypes, such as:
  - Add-Wins Set:

    $F_{aw\text{-}set}(ctxt) = F_{aw\text{-}set}(E, op, vis, ar) =$

    let E' = filterResets(E, op, vis) **in**

    $\{x \mid (\exists a \in E'.\ op(a)=add(x))$

    $\wedge\ \forall r \in E'.\ op(r)=remove(x) \rightarrow \exists a \in E'.op(a)=add(x)\ \wedge r <_{vis} a\}$
  - Auxiliary filterResets(E, op, vis) returns events not affected by reset

Transactions

- To specify transactions, we add one concept
  - An event e is associated with a transaction t=tx(e)
- We assume all transactions are committed
  - Our model does not include time
  - We do not define isolation levels
- Atomic visibility
  - Given two transactions t_1 and t_2
  - $\forall e_1, e_1', e_2, e_2' \in E$:

    $tx(e_1)=tx(e_1')=t_1 \land tx(e_2)=tx(e_2')=t_2 \Rightarrow e_1 \prec_{vis} e_2 \leftrightarrow e_1' \prec_{vis} e_2'$

LIGHTKONE

Versioned store extension

- Assume that each event e has a version(e)
  - A version is a set of events
  - User can provide a version for each event, if none then version(e)=$\bot$
- Min snapshot
  - $\forall e, e': e' \in version(e) \Rightarrow e' <_{vis} e$
- Precise snapshot
  - $\forall e, e': e' \in version(e) \Leftrightarrow e' <_{vis} e$

LIGHTKONE

CONCLUSIONS AND FURTHER WORK

Concrete semantics

- The concrete semantics refines the abstract semantics by adding nodes, node states, and messages between nodes
  - Burckhardt gives a general framework for concrete executions
  - In this framework we define node and communication failures
- Given a concrete execution, we can derive an observable history by considering events related to calls from a client
  - A history records the interactions between clients and the system
  - An abstract execution is a history that satisfies the correctness conditions given previously
  - If the observable history can be extended to a valid abstract execution (with vis and ar), then the concrete execution is correct
- With this approach, we can prove that Lasp and Antidote protocols satisfy the abstract semantics

LIGHTKONE

Conclusions

- We now have a first unified semantics that explains both Lasp and Antidote in a single framework
  - This is a step toward a general-purpose semantics for synchronization-free programming
  - In further development of both Lasp, Antidote, and Legion we will commit to respecting this semantics

- Much work remains to be done
  - We have an abstract execution semantics that explains the observable behavior, but does not model distribution or failure
  - We need to extend this to a concrete semantics that understands nodes and their interactions
  - For continued work on the programming model, the unified semantics needs to be extended with programming concepts such as modularity and functional abstraction

LIGHTKONE

ADDITIONAL SLIDES

Session guarantees

- We assume a session order so $\subset$ E×E that orders events from the same session
  - $e_1 \prec_{so} e_2$ if e_1 was submitted before e_2 in the same session
- We distinguish read and write operations
  - isRead(e) and isWrite(e) predicates
- Read Your Writes
  - $e_1 \prec_{so} e_2 \wedge isWrite(e_1) \wedge isRead(e_2) \rightarrow e_1 \prec_{vis} e_2$
- Monotonic Reads
  - $e_1 \prec_{so} e_2 \wedge isRead(e_1) \wedge isRead(e_2) \rightarrow (\forall e'.\ e' \prec_{vis} e_1 \rightarrow e' \prec_{vis} e_2)$
- Writes Follow Reads
  - $e_1 \prec_{so} e_2 \wedge isRead(e_1) \wedge isWrite(e_2) \rightarrow (\forall e'.\ e' \prec_{vis} e_1 \rightarrow e' \prec_{vis} e_2)$
- General Session Guarantee
  - $e_1 \prec_{so} e_2 \rightarrow e_1 \prec_{vis} e_2$