LASP
DISTRIBUTED
EVENTUALLY CONSISTENT COMPUTATIONS
EN TAL AV
CHRISTOPHER MEIKLEJOHN
RESEARCH WITH:

PETER VAN ROY (UCL)
MOTIVATION
SYNCHRONIZATION IS EXPENSIVE
SYNCHRONIZATION IS SOMETIMES IMPractical
MOBILE GAMES:

SHARED STATE BETWEEN CLIENTS

CLIENTS GO OFFLINE

INTERNET OF THINGS:

DISJOINT STATE AGGREGATED UPSTREAM
CLIENTS GO OFFLINE

NO TOTAL ORDER:

REPLICATED SHARED STATE WITH OFFLINE CLIENTS
CLIENTS NEED TO MAKE PROGRESS

WALL CLOCKS:

UNRELIABLE AT BEST
NON-DETERMINISTIC IF USED IN COMPUTATIONS

CONCURRENCY
RECONCILED BY USER
CRDTs
CRDTs PROVIDE

DETERMINISTIC RESOLUTION
CRDTs:

MAPS, SETS, COUNTERS, REGISTERS, GRAPHS
DETERMINISTIC RESOLUTION
CRDTs REALIZE
STRONG EVENTUAL CONSISTENCY
“CORRECT REPLICAS THAT HAVE DELIVERED THE SAME UPDATES HAVE EQUIVALENT STATE”

CRDTs EXAMPLE

‘MAX’ REGISTER
CRDTs EXAMPLE

‘ORSET’ SET
COMPOSITION IS NONTRIVIAL
fun(X) -> 2 end

F(\(R_C\))

\(R_C\)

\(\{1\}\)

\(\{\}\)

\(\{1\}\)

\(\{\}\)

\(\{\}\)

\(\{2\}\)

\(\{\}\)

\(\{2\}\)

\(\{\}\)

\(\{a, b\}, \{b\}\)

\(\{a, b\}, \{b\}\)

\(\{a, b\}, \{b\}\)
\[ R_A \quad F(R_A) \quad R_C \quad F(R_C) \]
\[ F(R_A) \]

\[ F(R_C) \]
COMPOSITION:

USER OBSERVABLE VALUE VS. STATE METADATA MAPPING IS NONTRIVIAL WITHOUT MAPPING METADATA; UNMERGABLE
LASP
WHAT IS
LASP
LASP

LATTICE PROCESSING
CENTRALIZED SEMANTICS:

DATATYPE COMPOSITION
fun(X) -> 2 end
DISTRIBUTED SEMANTICS:

DATATYPE COMPOSITION
DISTRIBUTED SEMANTICS:
SYSTEM COMPOSITION
CENTRALIZED RUNTIME:

SINGLE NODE MODEL
DISTRIBUTED RUNTIME:

MULTI-NODE MODEL
SEMANTICS
STREAMS
STREAMS:

CENTRALIZED EXECUTION
STREAMS:

DISTRIBUTED EXECUTION
STREAMS:

CONCURRENCY; FAILURE; ANTI-ENTROPY
STREAMS:

DISTRIBUTED EXECUTION DETERMINISTIC
REQUIRES EVENTUAL DELIVERY; ANTI-ENTROPY
DECLARE:

CREATE A STREAM OF A GIVEN CRDT TYPE
INITIALIZE STATE AT BOTTOM VALUE
BIND:

ASSIGN A VALUE TO THE STREAM
MERGE OF CURRENT AND NEW VALUE
MONOTONIC READ
MONOTONIC READ:

ENSURES FORWARD PROGRESS
BLOCKS ON INFLATIONS (OR STRICT INFLATIONS)
COMPOSITIONS ARE PROCESSES:

BLOCK FOR CHANGE IN INPUT
COMPUTE CHANGE
PROPAGATE CHANGE TO OUTPUT
FUNCTIONAL
FUNCTIONAL:

MAP; FILTER; FOLD
SET-THEORETIC
SET-THEORETIC:

PRODUCT; INTERSECTION; UNION
ARCHITECTURE
LASP STORE

SHARED VARIABLE STORE
LASP BACKENDS
LEVELDB, BITCASK, ETS
LASP CRDTs

PROVIDED BY RIAK DT
LASP ARCHITECTURE

CENTRALIZED SEMANTICS
STORE:

SHARED VARIABLE STORE
PROCESSES SYNCHRONIZE ON VARIABLES
LASP ARCHITECTURE
DISTRIBUTED SEMANTICS
STORE:

REPLICATED, SHARDED VARIABLE STORE
PROCESSES SYNCHRONIZE ON VARIABLES
REPLICATED:

DISTRIBUTED WITH RIAK CORE
QUORUM REQUESTS; ANTI-ENTROPY PROTOCOL
HYBRID:

DISTRIBUTE PROGRAMS; R/W WITH LOCAL STORE
CENTRALIZED EXECUTION
EXAMPLES
%%% Create initial set.
{ok, S1} = `lasp:declare`(Type),

%%% Add elements to initial set and update.
{ok, _} = `lasp:update`(S1, {add_all, [1,2,3]}, a),

%%% Create second set.
{ok, S2} = `lasp:declare`(Type),

%%% Apply map.
ok = `lasp:map`(S1, fun(X) -> X * 2 end, S2),
%% Create initial set.
{ok, S1} = lasp_core:declare(Type, Store),

%% Add elements to initial set and update.
{ok, _} = lasp_core:update(S1, {add_all, [1,2,3]}, a, Store),

%% Create second set.
{ok, S2} = lasp_core:declare(Type, Store),

%% Apply map.
ok = lasp_core:map(S1, fun(X) -> X * 2 end, S2, Store),
AD COUNTER
AD COUNTER:

TRACKS AD IMPRESSIONS
PUSHES ADVERTISEMENTS TO THE CLIENT
DISABLES AD AT 50,000+ IMPRESSIONS
CLIENTS DISPLAY ADS WHEN OFFLINE
AD COUNTER

INFORMATION FLOW
Increment

Rovio Ad Counter

Read ≥ 50,000

Remove

Rovio Ads

Union

Ads

Product

Contracts

Ads

Contracts

Filter

Ads With Contracts

Clients

Read

Clients

Clients

Clients

Lasp Operation

User-Maintained CRDT

Lasp-Maintained CRDT

Client Process
INFORMATION FLOW:

MONOTONIC
METADATA TO PREVENT DUPLICATE PROPAGATION
AD COUNTER

DISTRIBUTION
DISTRIBUTION BOUNDARIES:

ARBITRARY

ALLOWS COMPOSITION OF ENTIRE SYSTEM
AD COUNTER

EXAMPLE CODE
%% @doc Client process; standard recursive looping server.
client(Id, AdsWithContracts, PreviousValue) ->
  receive
    view_ad ->
      %% Get current ad list.
      {ok, {_, _, AdList0}} = lasp:read(AdsWithContracts, PreviousValue),
      AdList = riak_dt_orset:value(AdList0),
      case length(AdList) of
        0 ->
          %% No advertisements left to display; ignore
          %% message.
          client(Id, AdsWithContracts, AdList0);
        _ ->
          %% Select a random advertisement from the list of
          %% active advertisements.
          {#ad{counter=Ad}, _} = lists:nth(
            random:uniform(length(AdList)), AdList),
          %% Increment it.
          {ok, _} = lasp:update(Ad, increment, Id),
          lager:info("Incremented ad counter: -p", [Ad]),
          client(Id, AdsWithContracts, AdList0)
      end
  end
end.
%% @doc Server functions for the advertisement counter. After 5 views,
%% disable the advertisement.
%%
server({#ad(counter=Counter)=Ad, _}, Ads) ->
    %% Blocking threshold read for 5 advertisement impressions.
    {ok, _} = lasp:read(Counter, 5),

    %% Remove the advertisement.
    {ok, _} = lasp:update(Ads, {remove, Ad}, Ad),

    lager:info("Removing ad: -p", [Ad]).
%% Generate a series of unique identifiers.
RovioAdIds = lists:map(fun(_, _ = druid:vu4()) -> lists:seq(i, 10),
   lager:info("Rovio Ad Identifiers are: -p", [RovioAdIds]),
   TriforkAdIds = lists:map(fun(_, _ = druid:vu4()) -> lists:seq(i, 10),
   lager:info("Trifork Ad Identifiers are: -p", [TriforkAdIds]),
   lds = RovioAdIds ++ TriforkAdIds,
   lager:info("Ad Identifiers are: -p", [lds]),
   %% Generate Rovio's advertisements.
   (ok, RovioAds) = lasp:declare(?SET),
   lists:map(fun(id) ->
     %% Generate a G-Counter.
     (ok, CounterId) = lasp:declare(?COUNTER),
     %% Add it to the advertisement set.
     (ok, _) = lasp:update(RovioAds,
       {add, #ad(id=Id, counter=CounterId)},
       undefined)
   end, RovioAdIds),
   %% Generate Trifork's advertisements.
   (ok, TriforkAds) = lasp:declare(?SET),
   lists:map(fun(id) ->
     %% Generate a G-Counter.
     (ok, CounterId) = lasp:declare(?COUNTER),
     %% Add it to the advertisement set.
     (ok, _) = lasp:update(TriforkAds,
       {add, #ad(id=Id, counter=CounterId)},
       undefined)
   end, TriforkAdIds),
   %% Union ads.
   (ok, Ads) = lasp:declare(?SET),
   ok = lasp:union(RovioAds, TriforkAds, Ads),
   %% For each identifier, generate a contract.
   (ok, Contracts) = lasp:declare(?SET),
   lists:map(fun(id) ->
     (ok, _) = lasp:update(Contracts,
       {add, #contract(id=Id)},
       undefined)
   end, lds),
   %% Generate a series of advertisement sets.
   Compute the Cartesian product of both ads and contracts.
   (ok, AdsContracts) = lasp:declare(?SET),
   ok = lasp:product(Ads, Contracts, AdsContracts),
   %% Filter items by join on item it.
   (ok, AdsWithContracts) = lasp:declare(?SET),
   filterFun = fun({#ad(id=Id1, contract(id=Id2))} ->
     Id1 =<= Id2
   end,
   ok = lasp:filter(AdsContracts, filterFun, AdsWithContracts),
   %% Launch a series of client processes, each of which is responsible
   %% for displaying a particular advertisement.
   (ok, Clients) = lasp:declare(?SET),
   %% Each client takes the full list of ads when it starts, and reads
   %% from the variable store.
   lists:map(fun(id) ->
     ClientPid = spawn_link(?MODULE, client,
       [ld, AdsWithContracts, undefined]),
     (ok, _) = lasp:update(Clients,
       {add, ClientPid},
       undefined)
   end, lists:seq(1,5)),
   %% Launch a server process for each advertisement, which will block
   %% until the advertisement should be disabled.
   (ok, Servers) = lasp:declare(?SET),
   %% Get the current advertisement list.
   (ok, {_, _ = AdList0}) = lasp:read(AdsWithContracts),
   AdList = riak_dt_orset:value(AdList0),
   %% For each advertisement, launch one server for tracking it’s
   %% impressions and wait to disable.
   lists:map(fun(Ad) ->
     ServerPid = spawn_link(?MODULE, server, [Ad, AdList]),
     (ok, _) = lasp:update(Servers,
       {add, ServerPid},
       undefined)
   end, AdList),
% Generate a series of unique identifiers.
RovioAdIds = lists:map(fun(_) -> druid:idv4() end, lists:seq(1, 10)),
lager:info("Rovio Ad Identifiers are: -p", [RovioAdIds]),

TriforkAdIds = lists:map(fun(_) -> druid:idv4() end, lists:seq(1, 10)),
lager:info("Trifork Ad Identifiers are: -p", [TriforkAdIds]),

lds = RovioAdIds ++ TriforkAdIds,
lager:info("Ad Identifiers are: -p", [lds]).

% Generate Rovio's advertisements.
(ok, RovioAds) = lasp:declare(?SET),
lists:map(fun(id) ->
    % Generate a G-Counter.
    ok = lasp:declare(?COUNTER),
    % Add it to the advertisement set.
    ok = lasp:append(RovioAds, #ad{id=id}, counter=CounterId),
    undefined
  end, RovioAdIds),

% Generate Trifork's advertisements.
(ok, TriforkAds) = lasp:declare(?SET),
lists:map(fun(id) ->
    % Generate a G-Counter.
    ok = lasp:declare(?COUNTER),
    % Add it to the advertisement set.
    ok = lasp:append(TriforkAds, #ad{id=id}, counter=CounterId),
    undefined
  end, TriforkAdIds),

% Union ads.
(ok, Ads) = lasp:declare(?SET),
ok = lasp:union(RovioAds, TriforkAds, Ads),

% For each identifier, generate a contract.
(ok, Contracts) = lasp:declare(?SET),
lists:map(fun(id) ->
    ok = lasp:append(Contracts, #contract{id=id}),
    undefined
  end, lds),

% Compute the Cartesian product of both ads and contracts.
(ok, AdsContracts) = lasp:product(Ads, Contracts, AdsContracts),
ok = lasp:product(AdsContracts, AdsContracts),

% Filter items by join on item it.
(ok, AdsWithContracts) = lasp:declare(?SET),
filterfun = fun({#ad{id=id1}, #contract{id=id2}}) ->
    ok = lasp:filter(AdsContracts, FilterFun, AdsWithContracts),

% Launch a series of client processes, each of which is responsible
% for displaying a particular advertisement.

% Generate a OR-set for tracking clients.
(ok, Clients) = lasp:declare(?SET),

% Each client takes the full list of ads when it starts, and reads
% from the variable store.
lists:map(fun(id) ->
    ClientPid = spawn_link(?MODULE, client,
    [id, AdsWithContracts, undefined]),
    ok = lasp:append(Clients, ClientPid),
    undefined
  end, lists:seq(1,5)),

% Launch a server process for each advertisement, which will block
% until the advertisement should be disabled.

% Create a OR-set for the server list.
(ok, Servers) = lasp:declare(?SET),

% Get the current advertisement list.
(ok, (AdList)) = lasp:read(AdsWithContracts),
AdList = riax:dt_orset:read(value(AdList0)),

% For each advertisement, launch one server for tracking it’s
% impressions and wait to disable.
lists:map(fun(Ad) ->
    ServerPid = spawn_link(?MODULE, server, [Ad, Ads]),
    ok = lasp:append(Servers, ServerPid),
    undefined
  end, AdList),
%% Generate a series of unique identifiers.
RovioAdIds = lists:map(fun(_,) -> druid:uuidv4() end, lists:seq(1, 10)),
lager:info("Rovio Ad Identifiers are: -p', [RovioAdIds]),
TriforkAdIds = lists:map(fun(_,) -> druid:uuidv4() end, lists:seq(1, 10)),
lager:info("Trifork Ad Identifiers are: -p', [TriforkAdIds]),
Ids = RovioAdIds ++ TriforkAdIds,
lager:info("Ad Identifiers are: -p', [Ids]),

%% Generate Rovio's advertisements.
(ok, RovioAds) = lasp:declare(?SET),
lists:map(fun(Id) ->
%% Generate a G-Counter.
  (ok, CounterId) = lasp:declare(?COUNTER),
  %% Add it to the advertisement set.
  (ok, _) = lasp:update(RovioAds,
    {add, #ad(id=Id, counter=CounterId)},
    undefined)
end, RovioAdIds),

%% Generate Trifork's advertisements.
(ok, TriforkAds) = lasp:declare(?SET),
lists:map(fun(Id) ->
%% Generate a G-Counter.
  (ok, CounterId) = lasp:declare(?COUNTER),
  %% Add it to the advertisement set.
  (ok, _) = lasp:update(TriforkAds,
    {add, #ad(id=Id, counter=CounterId)},
    undefined)
end, TriforkAdIds),

%% Union ads.
(ok, Ads) = lasp:declare(?SET),
ok = lasp:union(RovioAds, TriforkAds, Ads),

%% For each identifier, generate a contract.
(ok, Contracts) = lasp:declare(?SET),
lists:map(fun(Ad) ->
  (ok, _) = lasp:update(Contracts,
    {add, #contract(id=Id)},
    undefined)
end, Ads),

%% Compute the Cartesian product of both ads and contracts.
(ok, AdsContracts) = lasp:declare(?SET),
ok = lasp:product(Ads, Contracts, AdsContracts),

%% Filter items by join on item its.
(ok, AdsWithContracts) = lasp:declare(?SET),
filterfun = fun({#ad(id=Id), #contract(id=Id2)} ->
    Id1 =:= Id2
end,
ok = lasp:filter(AdsContracts, FilterFun, AdsWithContracts),

%% Launch a series of client processes, each of which is responsible
%% for displaying a particular advertisement.

%% Generate a OR-set for tracking clients.
(ok, Clients) = lasp:declare(?SET),

%% Each client takes the full list of ads when it starts, and reads
%% from the variable store.
lists:map(fun(Id) ->
  ClientPid = spawn_link(?MODULE, client,
    [Id, AdsWithContracts, undefined]),
  (ok, _) = lasp:update(Clients,
    {add, ClientPid},
    undefined)
end, lists:seq(1, 5)),

%% Launch a server process for each advertisement, which will block
%% until the advertisement should be disabled.

%% Create a OR-set for the server list.
(ok, Servers) = lasp:declare(?SET),

%% Get the current advertisement list.
(ok, {_, AdsList0}) = lasp:read(AdsWithContracts),
AdsList = rtak_dt_orset:value(AdsList0),

%% For each advertisement, launch one server for tracking it's
%% impressions and wait to disable.
lists:map(fun(Ad) ->
  ServerPid = spawn_link(?MODULE, server, [Ad, Ads]),
  (ok, _) = lasp:update(Servers,
    {add, ServerPid},
    undefined)
end, AdsList),
Generate a series of unique identifiers. 
\[ \text{RovioAdIds} = \text{lists:map}(\text{fun}(_) \rightarrow \text{druid:dv4()} \ \text{end}, \text{lists:seq}(1, \text{10})), \]
\[ \text{lager:info('Rovio Ad Identifiers are: \_p', [RovioAdIds]).} \]

\[ \text{TriforkAdIds} = \text{lists:map}(\text{fun}(_) \rightarrow \text{druid:dv4()} \ \text{end}, \text{lists:seq}(1, \text{10})), \]
\[ \text{lager:info('Trifork Ad Identifiers are: \_p', [TriforkAdIds]).} \]

\[ \text{lids = RovioAdIds ++ TriforkAdIds,} \]
\[ \text{lager:info('Ad Identifiers are: \_p', [lids]).} \]

Generate Rovio's advertisements. 
\[ \text{(ok, RovioAds) = \text{lasp:declare}(\text{SET}),} \]
\[ \text{lists:map}(\text{fun}(\text{id}) \rightarrow \]
\[ \%\% \text{Generate a G-Counter.} \]
\[ \text{(ok, CounterId) = \text{lasp:declare}(\text{COUNTER})}, \]
\[ \%\% \text{Add it to the advertisement set.} \]
\[ \text{(ok, \_)} = \text{lasp:update}(\text{RovioAds,}} \]
\[ \%\% \text{Add it to the advertisement set.} \]
\[ (\text{add, \#ad(id=ld, counter=CounterId))}, \]
\[ \text{\text{undefined})} \]
\[ \text{end, RovioAdIds),} \]

Generate Trifork's advertisements. 
\[ \text{(ok, TriforkAds) = \text{lasp:declare}(\text{SET}),} \]
\[ \text{lists:map}(\text{fun}(\text{id}) \rightarrow \]
\[ \%\% \text{Generate a G-Counter.} \]
\[ \text{(ok, CounterId) = \text{lasp:declare}(\text{COUNTER})}, \]
\[ \%\% \text{Add it to the advertisement set.} \]
\[ (\text{ok, \_)} = \text{lasp:update}(\text{TriforkAds,}} \]
\[ \%\% \text{Add it to the advertisement set.} \]
\[ (\text{add, \#ad(id=ld, counter=CounterId))}, \]
\[ \text{\text{undefined})} \]
\[ \text{end, TriforkAdIds),} \]

Union ads. 
\[ \text{(ok, Ads) = \text{lasp:declare}(\text{SET}),} \]
\[ \text{ok = \text{lasp:union}(\text{RovioAds, TriforkAds, Ads}),} \]

For each identifier, generate a contract. 
\[ \text{(ok, Contracts) = \text{lasp:declare}(\text{SET}),} \]
\[ \text{lists:map}(\text{fun}(\text{id}) \rightarrow \]
\[ \text{(ok, \_)} = \text{lasp:update}(\text{Contracts,}} \]
\[ \text{(add, \#contract(id=ld))}, \]
\[ \text{\text{undefined})} \]
\[ \text{end, \text{Ads}),} \]

Compute the Cartesian product of both ads and contracts. 
\[ \text{(ok, AdsContracts) = \text{lasp:declare}(\text{SET}),} \]
\[ \text{ok = \text{lasp:product}(\text{Ads, Contracts, AdsContracts}),} \]

Filter items by join on item it. 
\[ \text{(ok, AdsWithContracts) = \text{lasp:declare}(\text{SET}),} \]
\[ \text{filterfun = \text{fun}(\text{\#ad(id=ldl, \#contract(id=id2)}) \rightarrow \}
\[ \text{ld1 ::= ld2} \]
\[ \text{end,} \]
\[ \text{ok = \text{lasp:filter}(\text{AdsContracts, FilterFun, AdsWithContracts}),} \]

Launch a series of client processes, each of which is responsible for displaying a particular advertisement. 

Generate a OR-set for tracking clients. 
\[ \text{(ok, Clients) = \text{lasp:declare}(\text{SET}),} \]

Each client takes the full list of ads when it starts, and reads from the variable store. 
\[ \text{lists:map}(\text{fun}(\text{id}) \rightarrow \]
\[ \text{\text{ClientPid = spawn_link(\text{MODULE, client,}} \]
\[ \text{[id, AdsWithContracts, undefined])}}, \]
\[ \text{(ok, \_)} = \text{lasp:update}(\text{Clients,}} \]
\[ \text{(add, ClientPid),} \]
\[ \text{\text{undefined})} \]
\[ \text{end, lists:seq(1,5))}, \]

Launch a server process for each advertisement, which will block until the advertisement should be disabled. 

Create a OR-set for the server list. 
\[ \text{(ok, Servers) = \text{lasp:declare}(\text{SET}),} \]

Get the current advertisement list. 
\[ \text{(ok, (_:_, AdList0)) = \text{lasp:read}(\text{AdsWithContracts}),} \]
\[ \text{AdList = riak_dt_orset:value(AdList0),} \]

For each advertisement, launch one server for tracking it's impressions and wait to disable. 
\[ \text{lists:map}(\text{fun}(\text{Ad}) \rightarrow \]
\[ \text{\text{ServerPid = spawn_link(\text{MODULE, server, [Ad, Ads]),}} \]
\[ \text{(ok, \_)} = \text{lasp:update}(\text{Servers,}} \]
\[ \text{(add, ServerPid),} \]
\[ \text{\text{undefined})} \]
\[ \text{end, AdList),} \]

Launch a server process for each advertisement, which will block until the advertisement should be disabled. 

Create a OR-set for the server list. 
\[ \text{(ok, Servers) = \text{lasp:declare}(\text{SET}),} \]

Get the current advertisement list. 
\[ \text{(ok, (_:_, AdList0)) = \text{lasp:read}(\text{AdsWithContracts}),} \]
\[ \text{AdList = riak_dt_orset:value(AdList0),} \]

For each advertisement, launch one server for tracking it's impressions and wait to disable. 
\[ \text{lists:map}(\text{fun}(\text{Ad}) \rightarrow \]
\[ \text{\text{ServerPid = spawn_link(\text{MODULE, server, [Ad, Ads]),}} \]
\[ \text{(ok, \_)} = \text{lasp:update}(\text{Servers,}} \]
\[ \text{(add, ServerPid),} \]
\[ \text{\text{undefined})} \]
\[ \text{end, AdList),} \]

Launch a server process for each advertisement, which will block until the advertisement should be disabled. 

Create a OR-set for the server list. 
\[ \text{(ok, Servers) = \text{lasp:declare}(\text{SET}),} \]

Get the current advertisement list. 
\[ \text{(ok, (_:_, AdList0)) = \text{lasp:read}(\text{AdsWithContracts}),} \]
\[ \text{AdList = riak_dt_orset:value(AdList0),} \]

For each advertisement, launch one server for tracking it's impressions and wait to disable. 
\[ \text{lists:map}(\text{fun}(\text{Ad}) \rightarrow \]
\[ \text{\text{ServerPid = spawn_link(\text{MODULE, server, [Ad, Ads]),}} \]
\[ \text{(ok, \_)} = \text{lasp:update}(\text{Servers,}} \]
\[ \text{(add, ServerPid),} \]
\[ \text{\text{undefined})} \]
\[ \text{end, AdList),} \]
%% Generate a series of unique identifiers.
RovioAdIds = lists:map(fun(_) -> druid:idv4() end, lists:seq(1, 10)),
lager:info("Rovio Ad Identifiers are: [RovioAdIds].")

TriforkAdIds = lists:map(fun(_) -> druid:idv4() end, lists:seq(1, 10)),
lager:info("Trifork Ad Identifiers are: [TriforkAdIds].")

Ids = RovioAdIds ++ TriforkAdIds,
lager:info("Ad Identifiers are: [Ids].")

%% Generate Rovio's advertisements.
(ok, RovioAds) = lasp:declare(?SET),
lists:map(fun(Id) ->
    %% Generate a G-Counter.
    (ok, CounterId) = lasp:declare(?COUNTER),
    %% Add it to the advertisement set.
    (ok, _) = lasp:update(RovioAds,
        [add, #ad(id=Id, counter=CounterId)],
        undefined)
    end, RovioAdIds),

%% Generate Trifork's advertisements.
(ok, TriforkAds) = lasp:declare(?SET),
lists:map(fun(Id) ->
    %% Generate a G-Counter.
    (ok, CounterId) = lasp:declare(?COUNTER),
    %% Add it to the advertisement set.
    (ok, _) = lasp:update(TriforkAds,
        [add, #ad(id=Id, counter=CounterId)],
        undefined)
    end, TriforkAdIds),

%% Union ads.
(ok, Ads) = lasp:declare(?SET),
ok = lasp:union(RovioAds, TriforkAds, Ads),

%% For each identifier, generate a contract.
(ok, Contracts) = lasp:declare(?SET),
lists:map(fun(Id) ->
    (ok, ...) = lasp:update(Contracts,
        [add, #contract(id=Id)],
        undefined)
    end, Ids),

%% Compute the Cartesian product of both ads and contracts.
(ok, AdsContracts) = lasp:declare(?SET),
ok = lasp:product(Ads, Contracts, AdsContracts),

%% Filter items by join on item it.
(ok, AdsWithContracts) = lasp:declare(?SET),
filterfun = fun(#ad(id=Id1), #contract(id=Id2)) ->
    Id1 =/= Id2
    end,
ok = lasp:filter(AdsContracts, FilterFun, AdsWithContracts),

%% Launch a series of client processes, each of which is responsible
%% for displaying a particular advertisement.

%% Generate a OR-set for tracking clients.
(ok, Clients) = lasp:declare(?SET),

%% Each client takes the full list of ads when it starts, and reads
%% from the variable store.
lists:map(fun(Id) ->
    ClientPid = spawn_link(?MODULE, client,
        [Id, AdsWithContracts, undefined]),
    (ok, _) = lasp:update(Clients,
        [add, ClientPid],
        undefined)
    end, lists:seq(1, 5)),

%% Launch a server process for each advertisement, which will block
%% until the advertisement should be disabled.

%% Create a OR-set for the server list.
(ok, Servers) = lasp:declare(?SET),

%% Get the current advertisement list.
(ok, (_Orset)) = lasp:read(AdsWithContracts),
Orset = rtab:dr_orset:value(AdList0),

%% For each advertisement, launch one server for tracking it's
%% impressions and wait to disable.
lists:map(fun(Ad) ->
    ServerPid = spawn_link(?MODULE, server, [Ad, Orset]),
    (ok, _) = lasp:wait(ServerPid),
    (add, ServerPid),
    undefined)
    end, AdList).
RELATED WORK
RELATED WORK

DISTRIBUTED OZ
RELATED WORK

DERFLOW_L
RELATED WORK

BLOOM^L
RELATED WORK
LVARS
RELATED WORK

D-STREAMS
RELATED WORK

SUMMING BIRD
FUTURE WORK
FUTURE WORK

INVARIANT PRESERVATION
FUTURE WORK

CAUSAL+ CONSISTENCY
FUTURE WORK

ORSWOT OPTIMIZATION
FUTURE WORK

DELTA STATE-CRDTs
FUTURE WORK

OPERATION-BASED CRDTs
FUTURE WORK

DEFORESTATION
SOURCE

GITHUB.COM/CMEIKLEJOHN/LASP
ERLANG WORKSHOP 2014

DERFLOW

DISTRIBUTED DETERMINISTIC DATAFLOW

PROGRAMMING FOR ERLANG
PAPOC / EUROSY S 2015

LASP

A LANGUAGE FOR DISTRIBUTED, EVENTUALLY CONSISTENT COMPUTATIONS WITH CRDTs
SYNCFREE

SYNCFREE IS A EUROPEAN RESEARCH PROJECT TAKING PLACE FOR 3 YEARS, STARING OCTOBER 2013, AND IS FUNDED BY THE EUROPEAN UNION, GRANT AGREEMENT N°609551.
FIN