

Coordination-Free Computations

Christopher Meiklejohn

LASP

DISTRIBUTED, EVENTUALLY CONSISTENT COMPUTATIONS

CHRISTOPHER MEIKLEJOHN (BASHO TECHNOLOGIES, INC.)
PETER VAN ROY (UNIVERSITÉ CATHOLIQUE DE LOUVAIN)



LASP MOTIVATION

**SYNCHRONIZATION IS
EXPENSIVE / IMPRACTICAL**

MOBILE GAMES:

SHARED STATE BETWEEN CLIENTS
CLIENTS GO OFFLINE

<http://www.rovio.com/en/news/blog/261/263-million-monthly-active-users-in-december/>

“INTERNET OF THINGS”: DISJOINT STATE AGGREGATED UPSTREAM CLIENTS GO OFFLINE

Gubbi, Jayawardhana, et al. "Internet of Things (IoT): A vision, architectural elements, and future directions." Future Generation Computer Systems 29.7 (2013): 1645-1660.

NO TOTAL ORDER:

REPLICATED SHARED STATE WITH OFFLINE CLIENTS
CLIENTS NEED TO MAKE PROGRESS

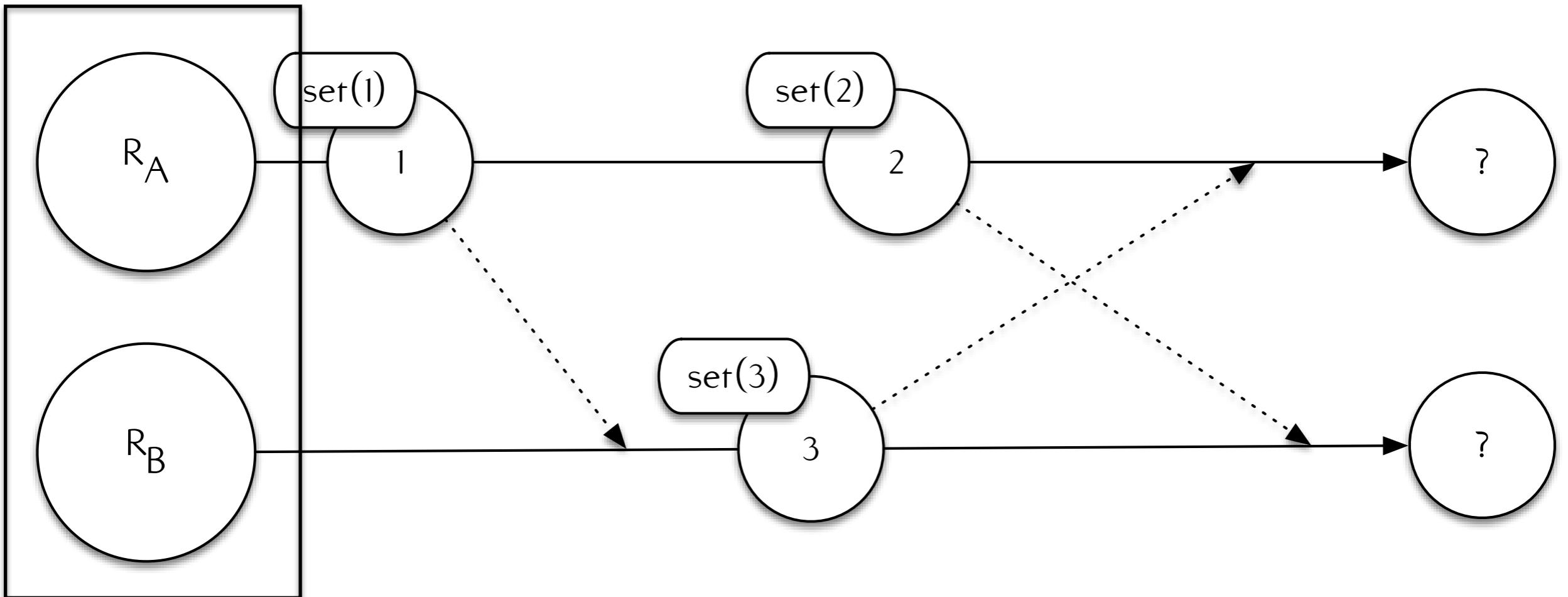
Gilbert, Seth, and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." ACM SIGACT News 33.2 (2002): 51-59.

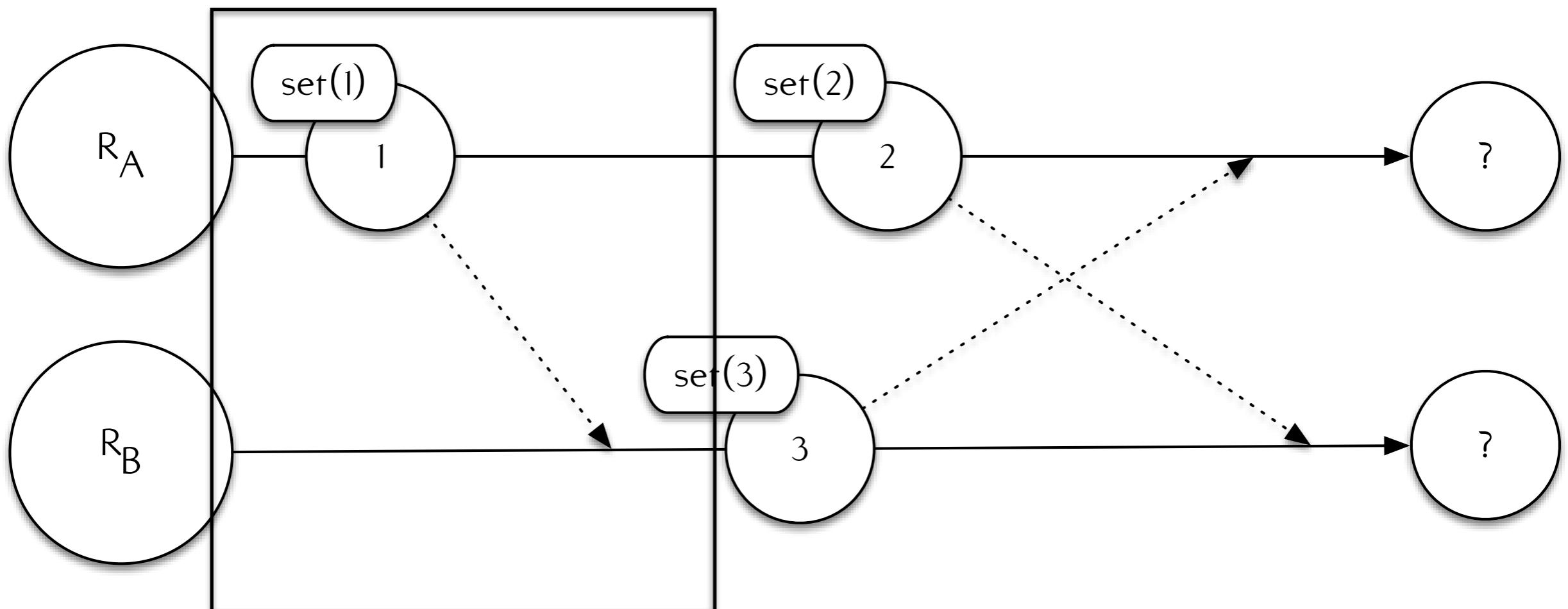
WALL CLOCKS:

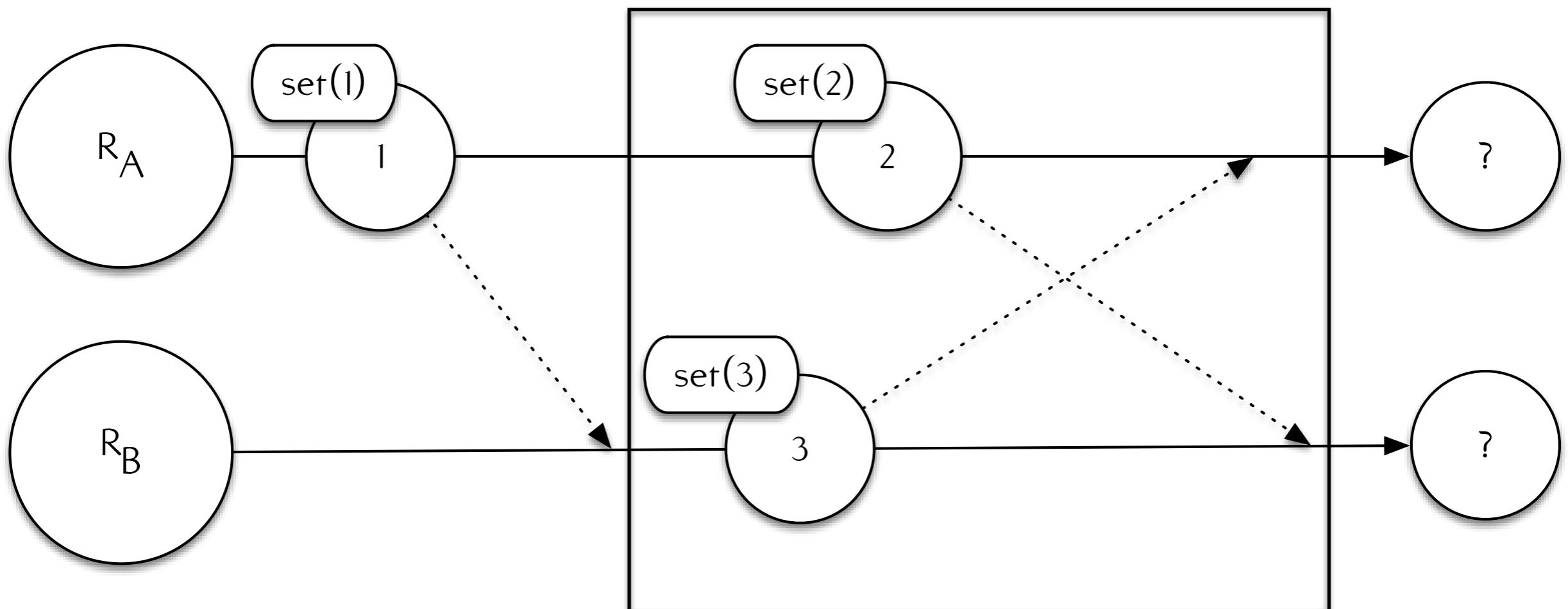
UNRELIABLE AT BEST
NON-DETERMINISTIC IF USED IN COMPUTATIONS

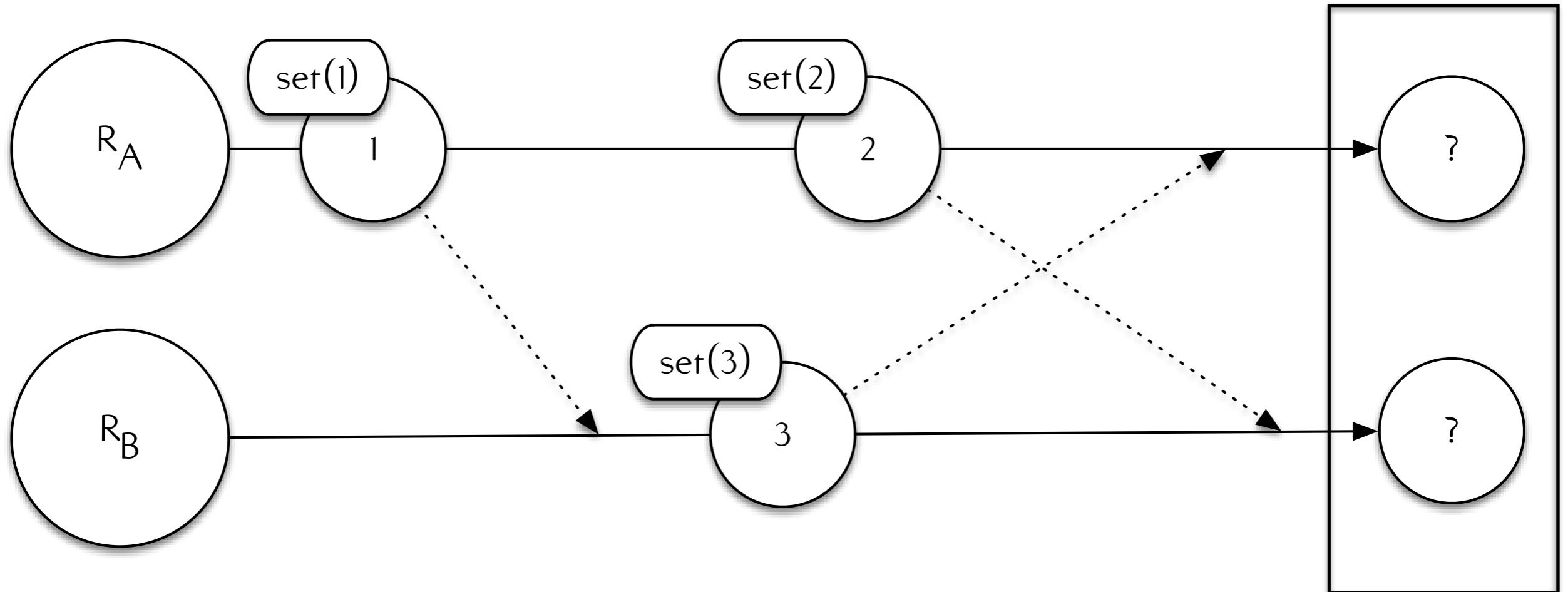
Corbett, James C., et al. "Spanner: Google's globally distributed database." ACM Transactions on Computer Systems (TOCS) 31.3 (2013): 8.

CONCURRENCY RECONCILED BY USER









CRDTs

CRDTs PROVIDE
DETERMINISTIC RESOLUTION

CRDTs:

MAPS, SETS, COUNTERS, REGISTERS, GRAPHS
DETERMINISTIC RESOLUTION

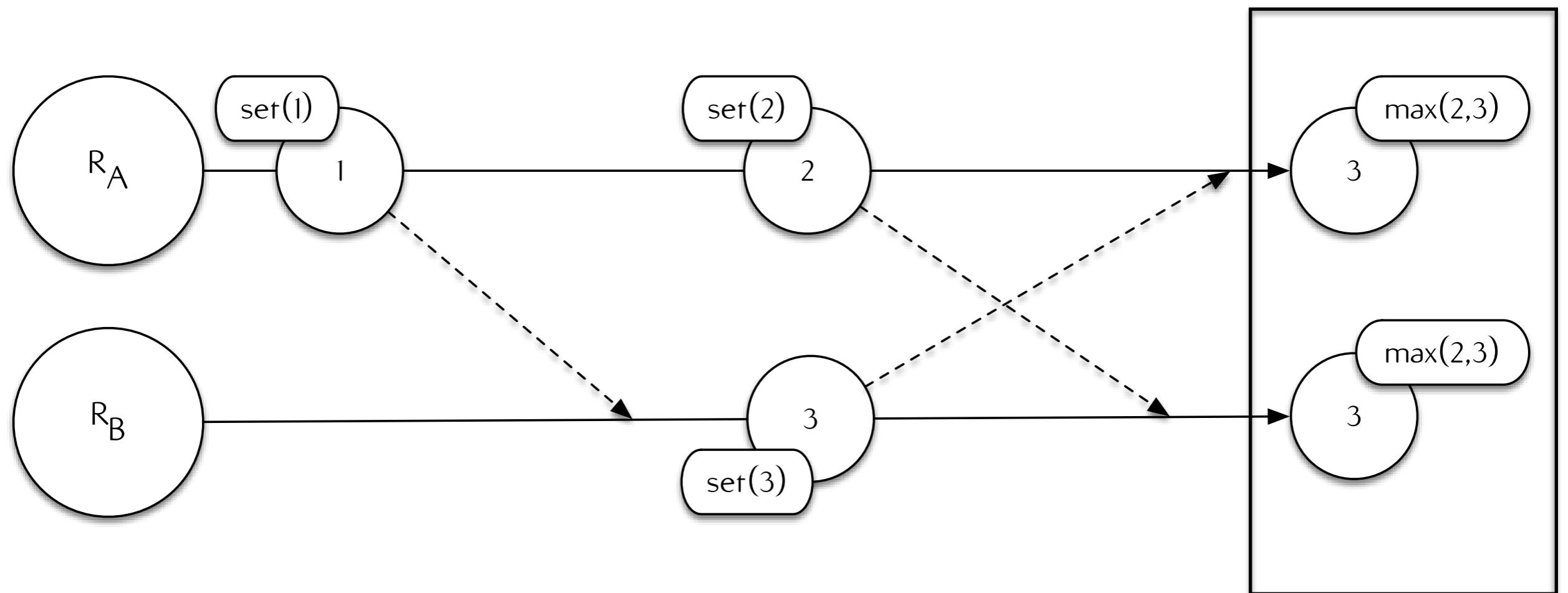
CRDTs REALIZE MONOTONIC STRONG EVENTUAL CONSISTENCY

“CORRECT REPLICAS THAT HAVE DELIVERED THE
SAME UPDATES HAVE EQUIVALENT STATE”

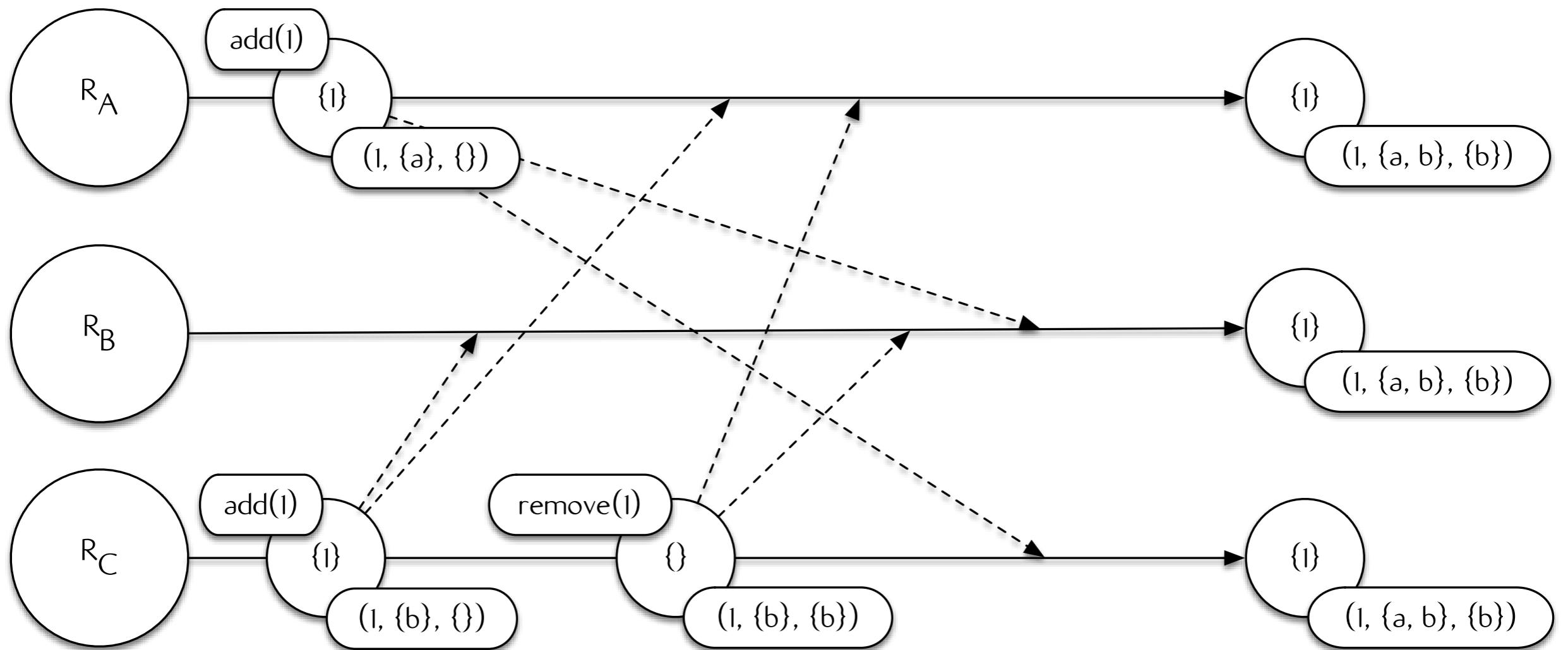
Shapiro, Marc, et al. "Conflict-free replicated data types." Stabilization, Safety, and Security of Distributed Systems. Springer Berlin Heidelberg, 2011. 386-400.

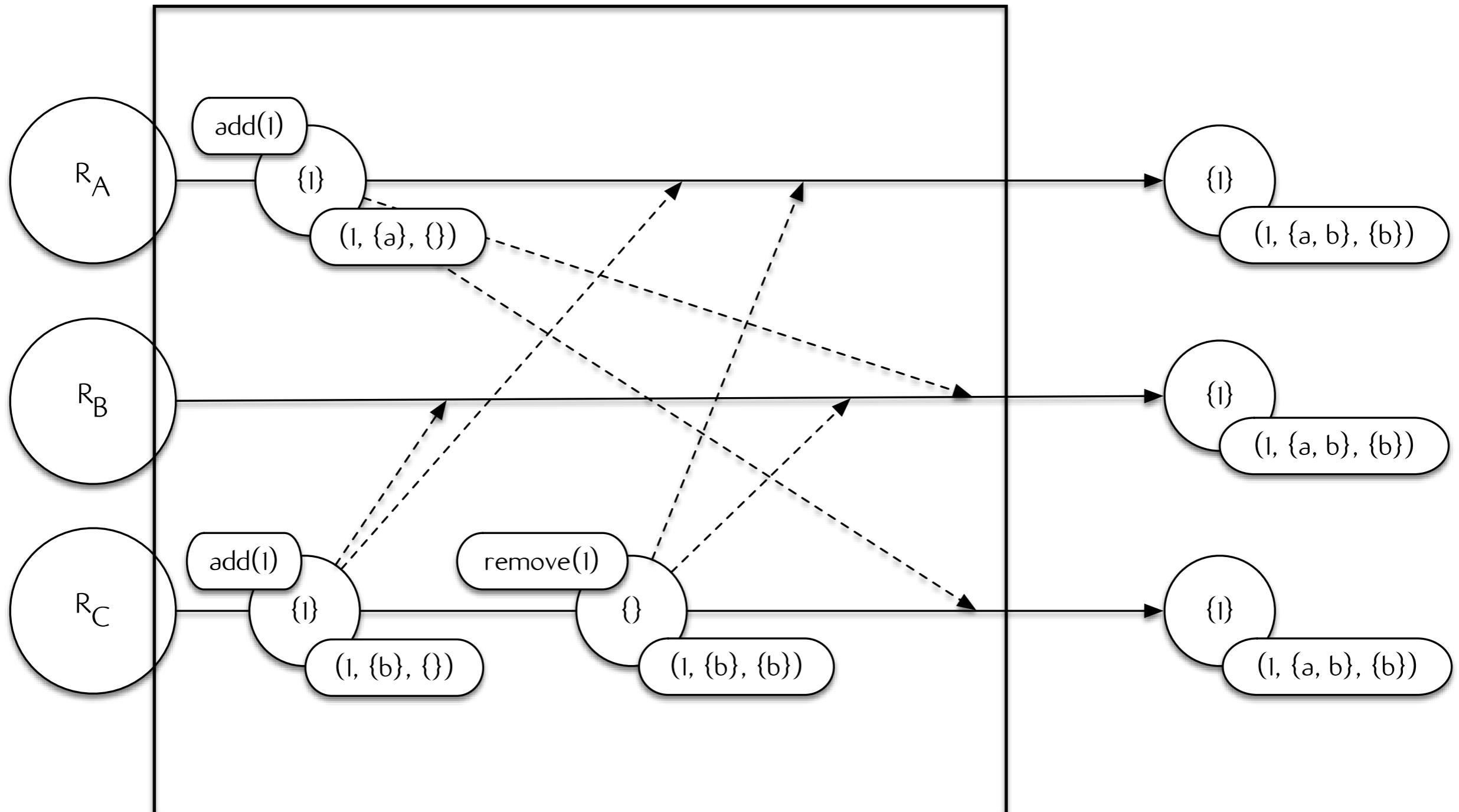
CRDTs EXAMPLE

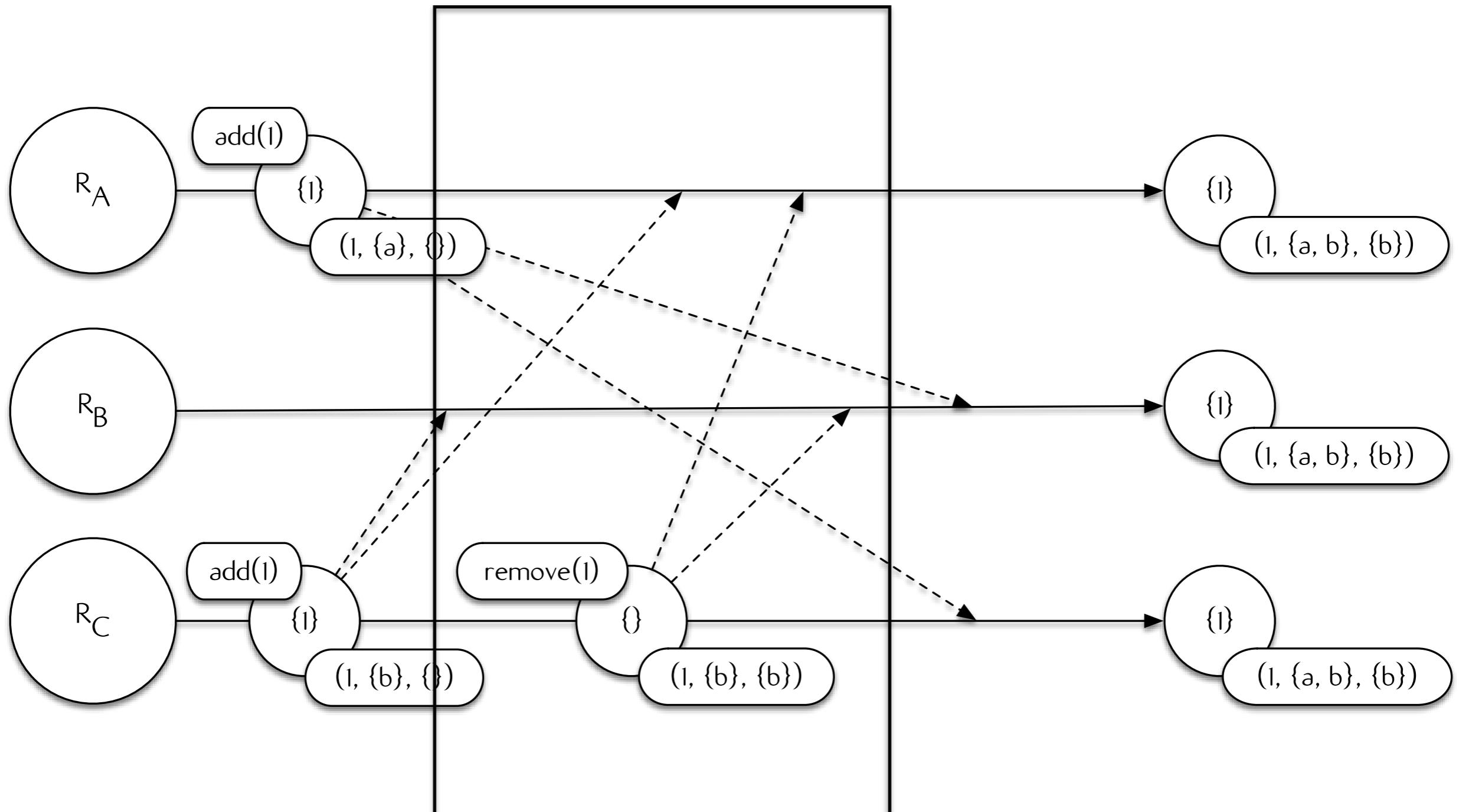
‘MAX’ REGISTER

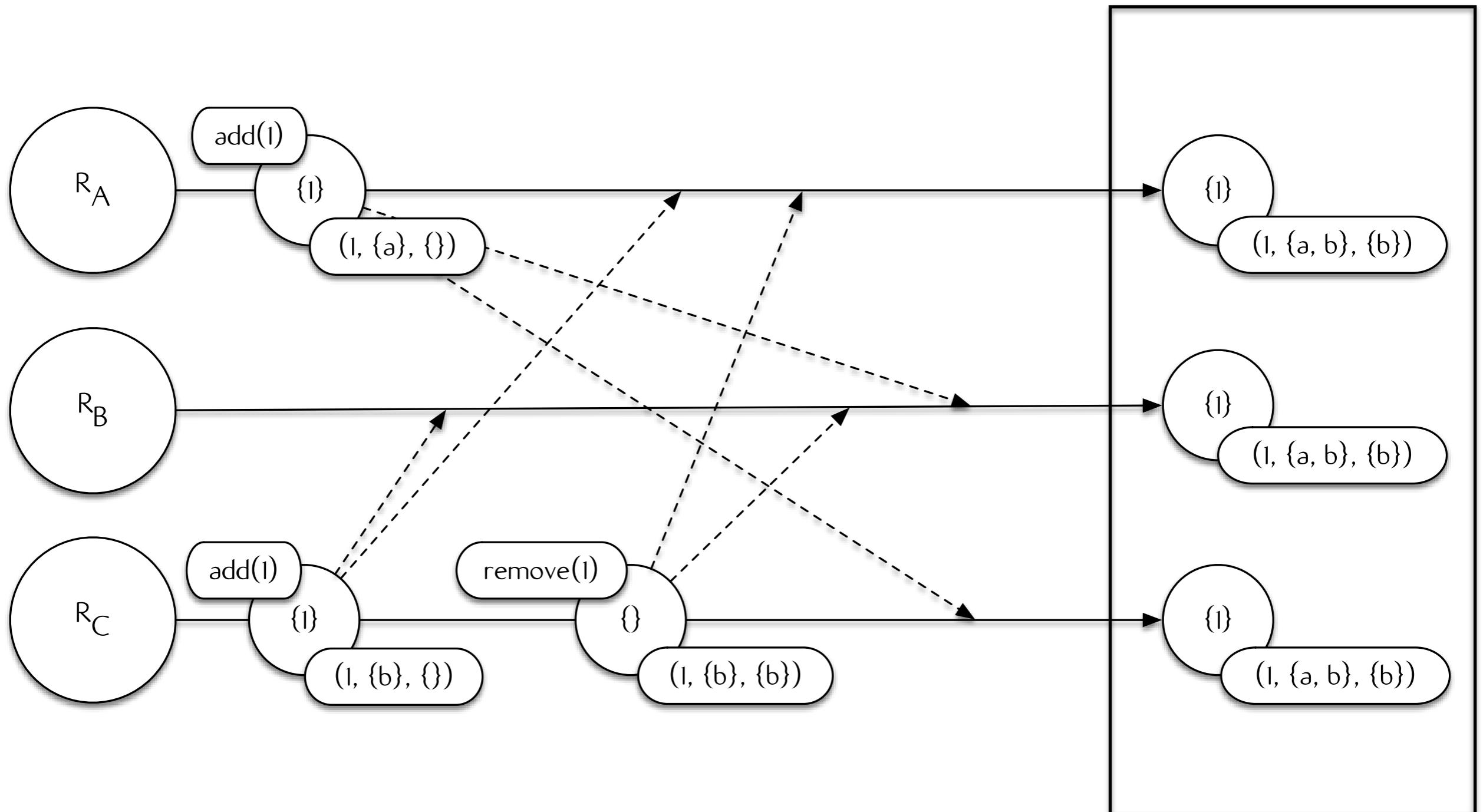


CRDTs EXAMPLE ‘ORSET’ SET



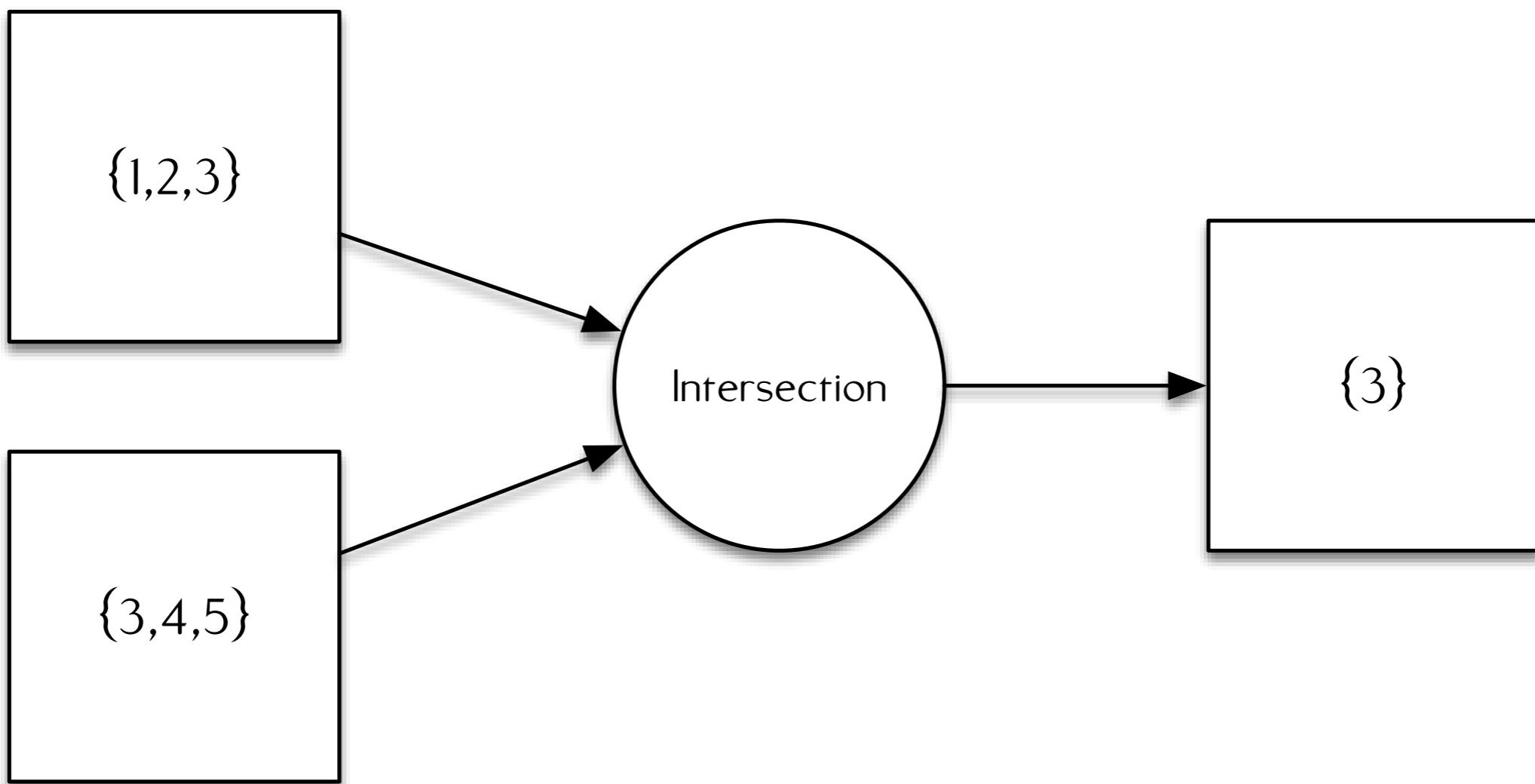


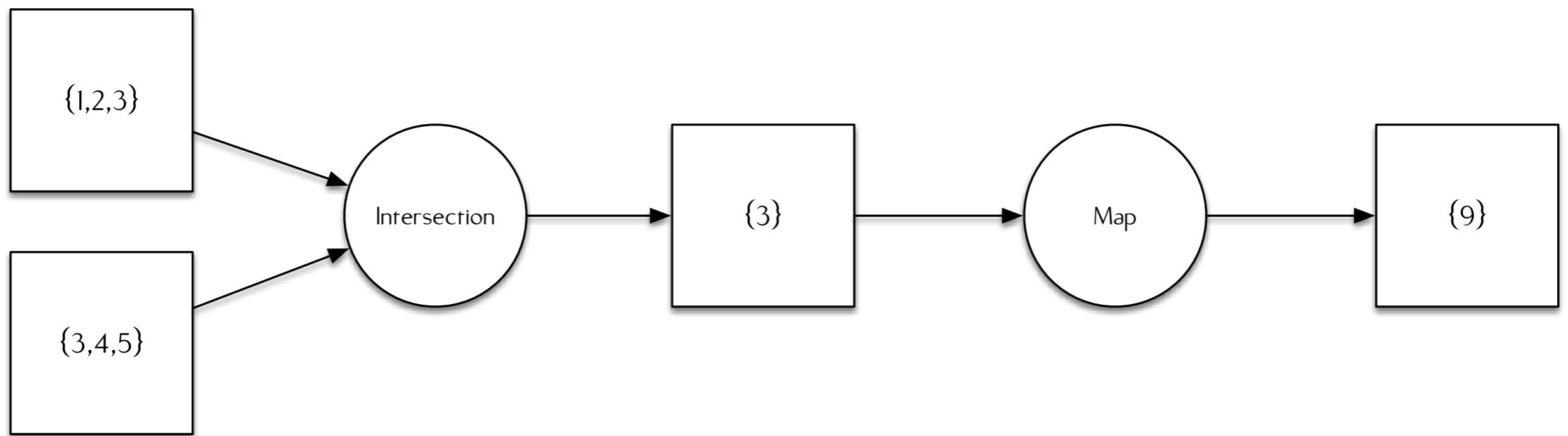


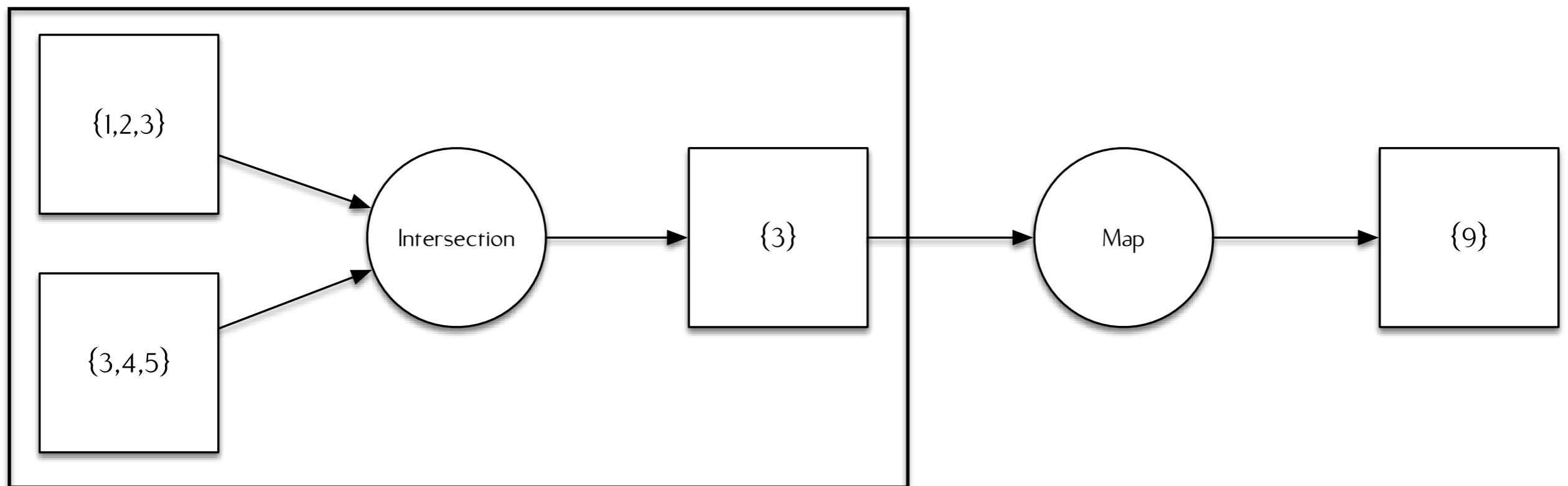


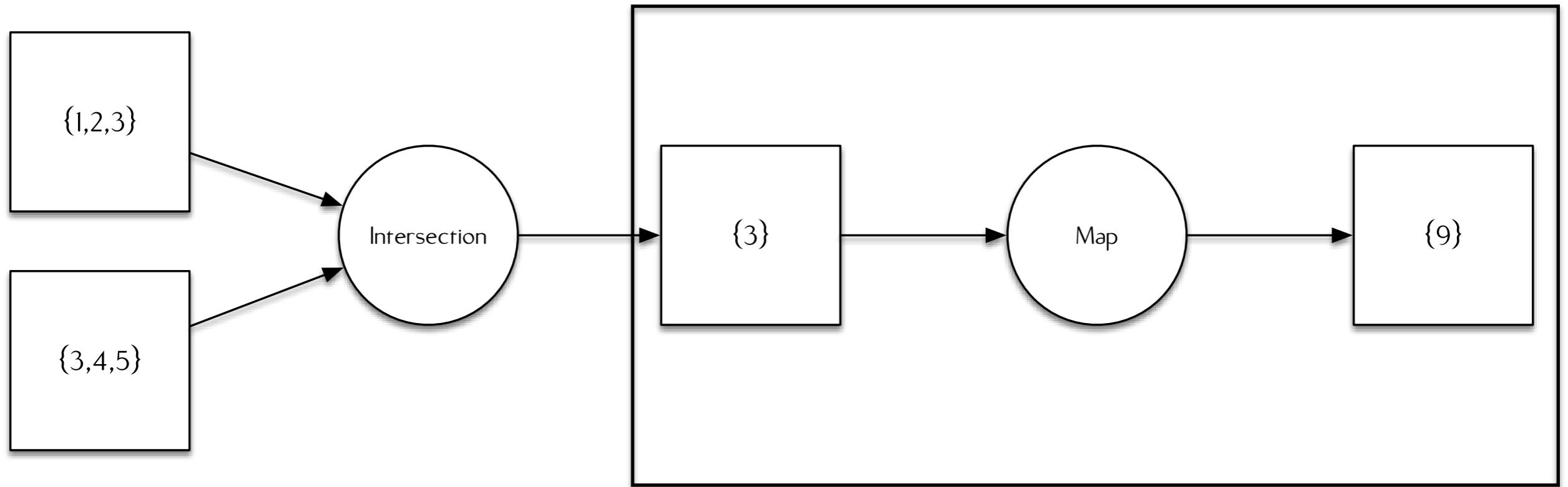
FRAGMENTS & PROGRAMS

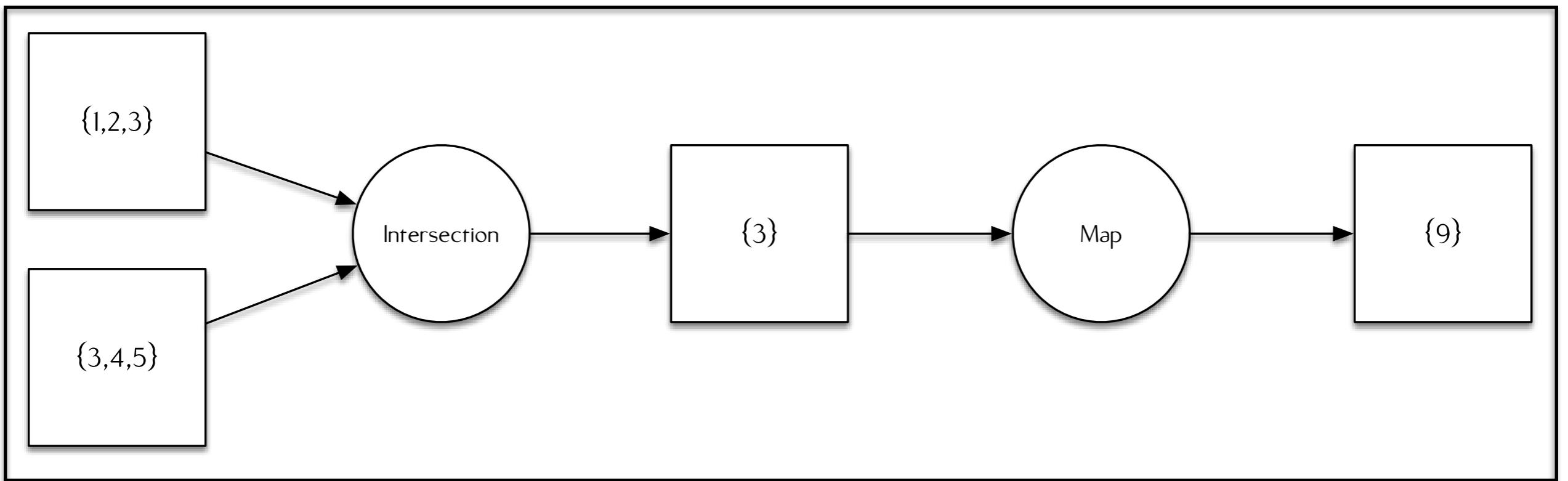
$$\{1,2,3\}$$
$$\{3,4,5\}$$

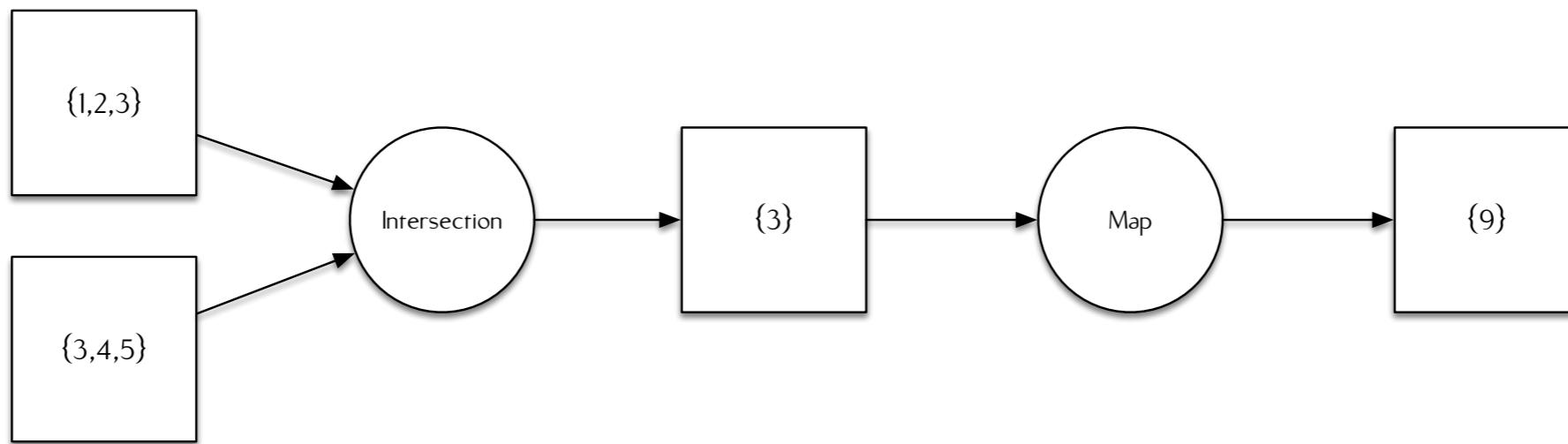
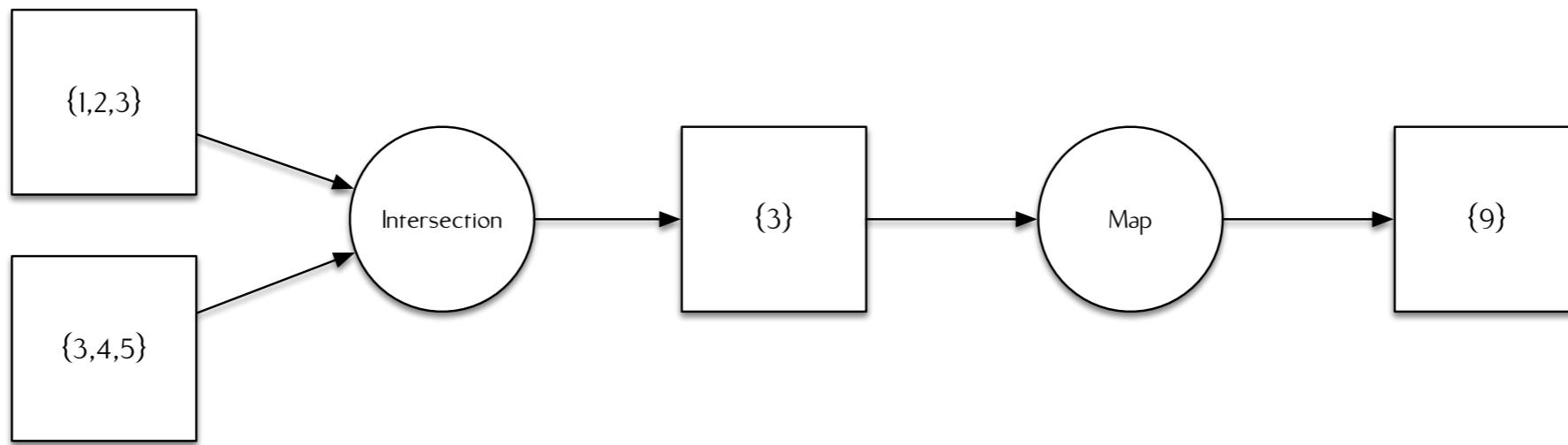


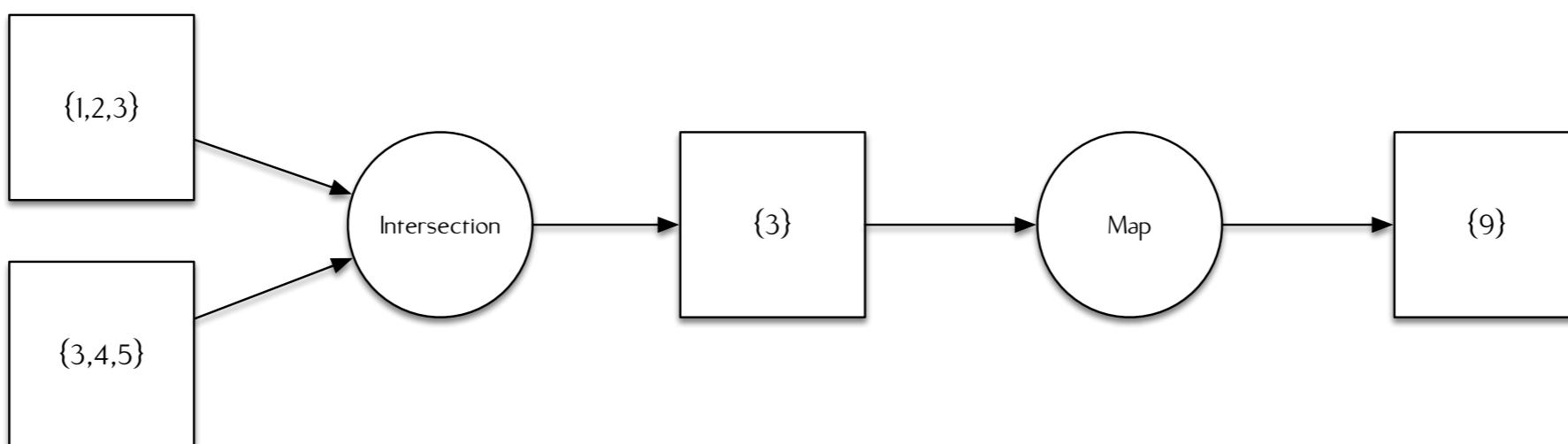
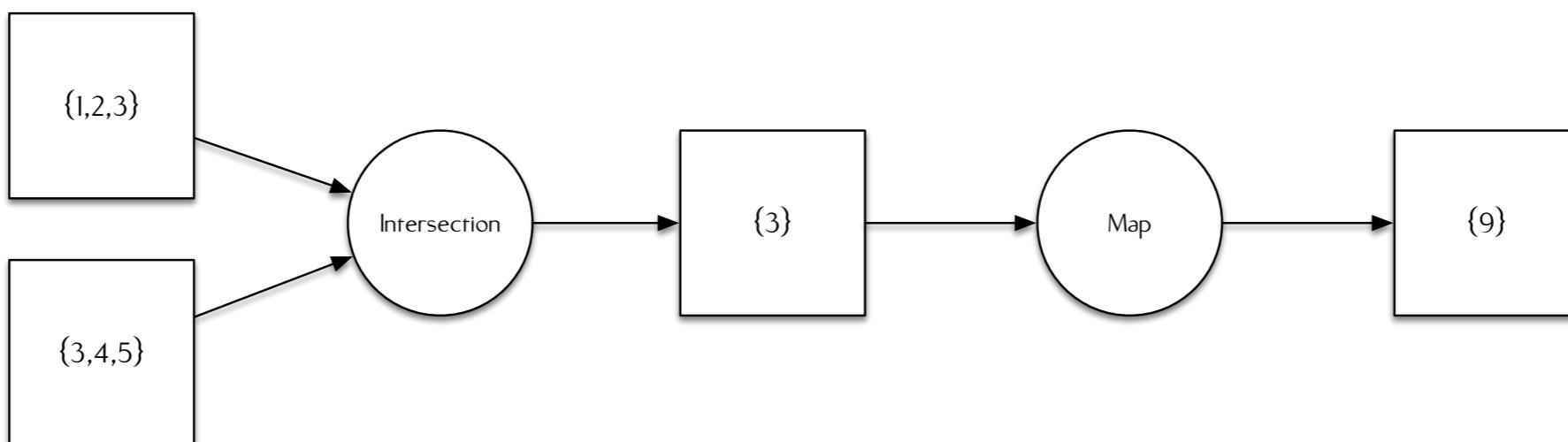


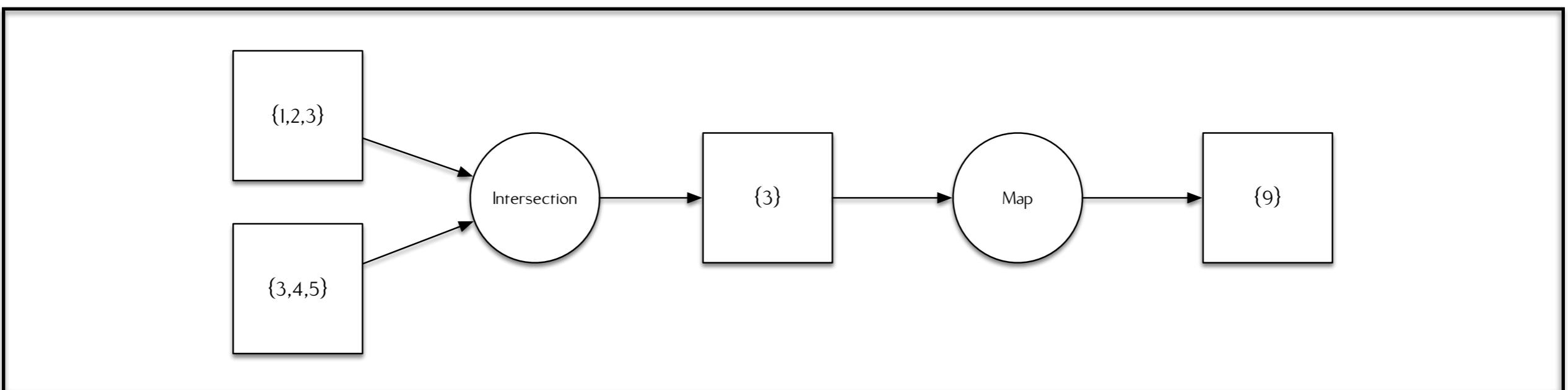
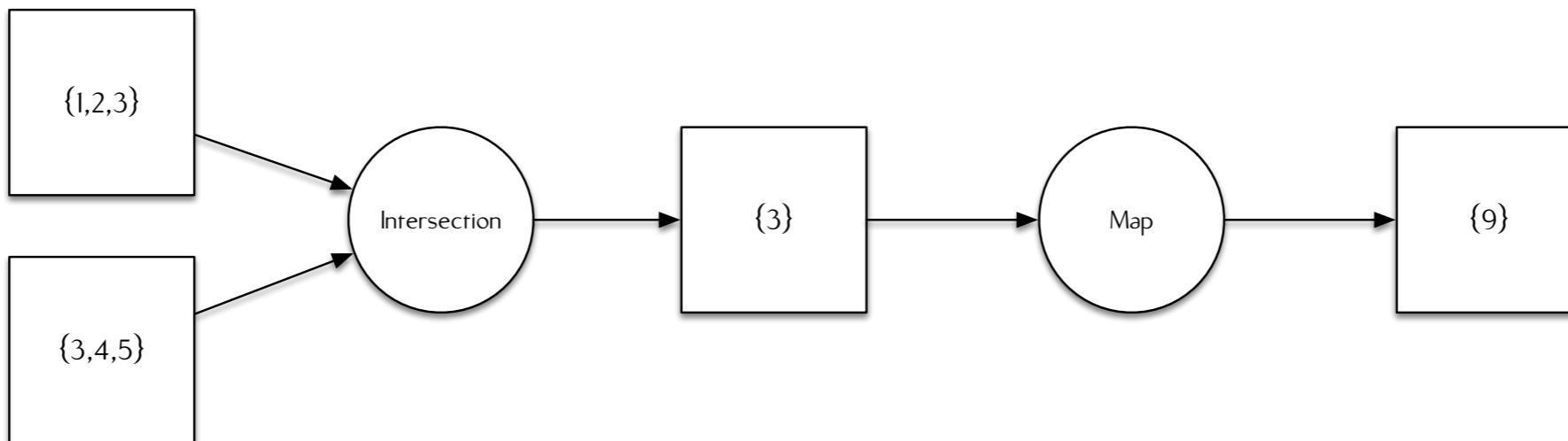




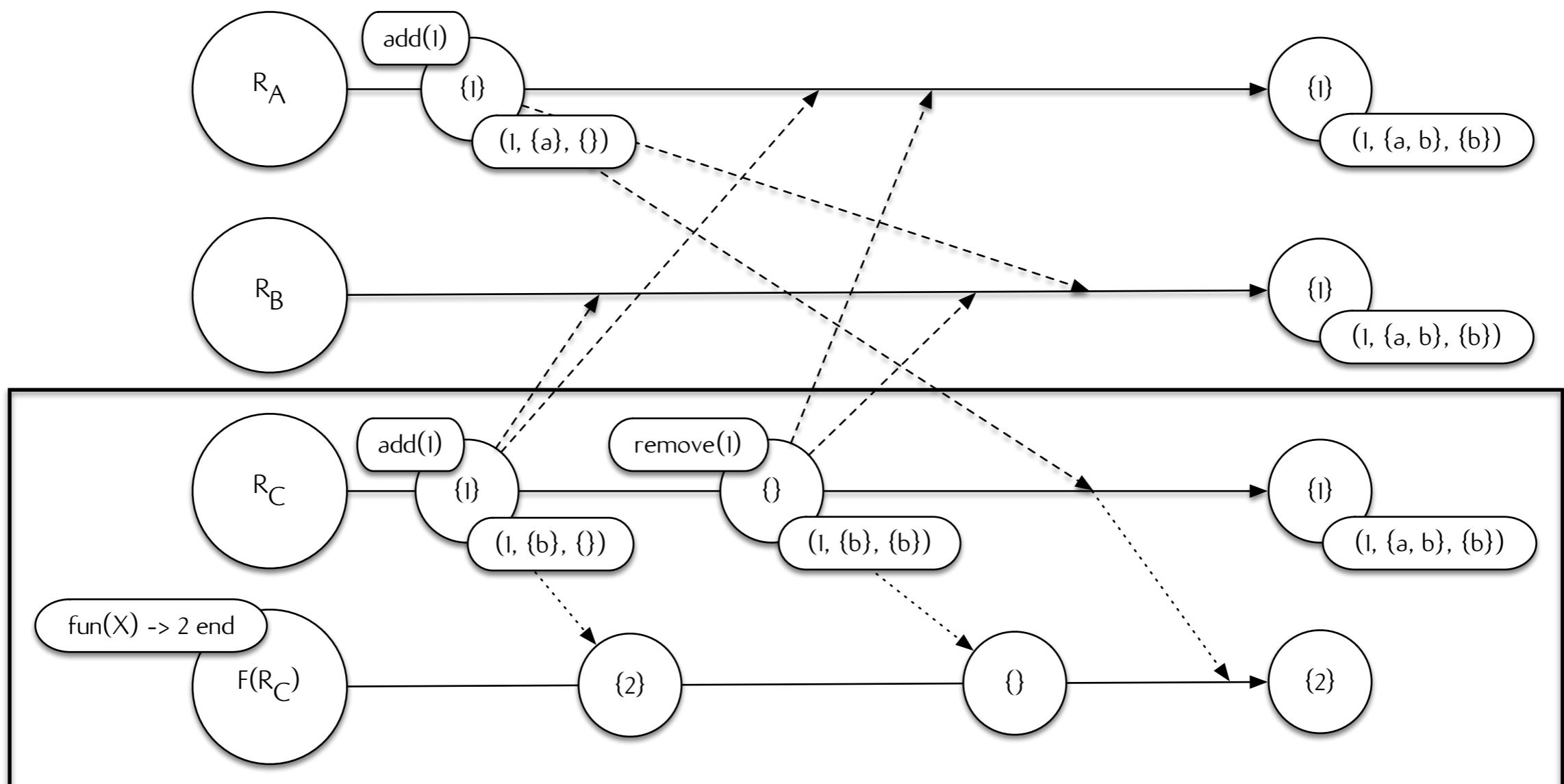


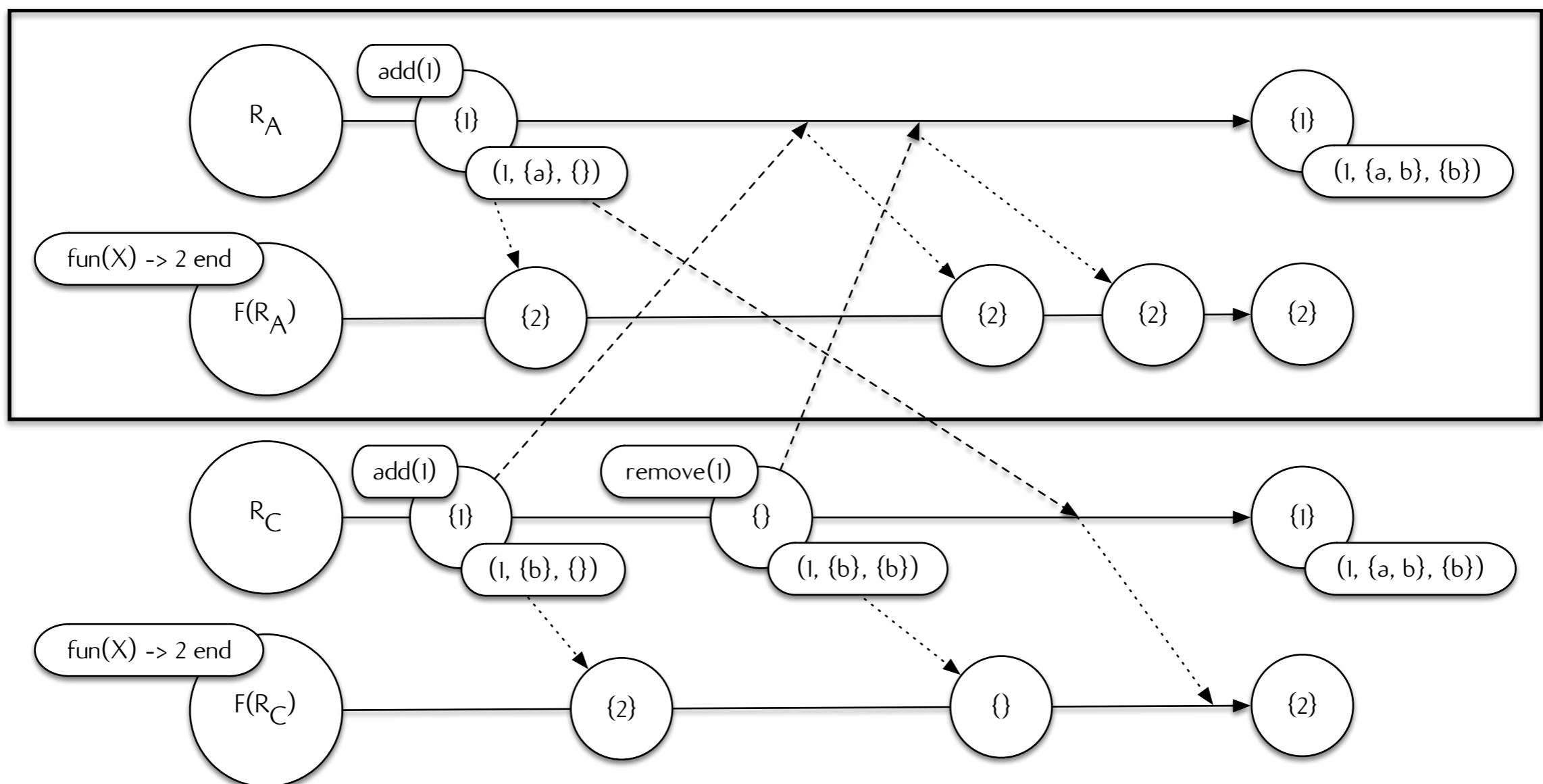


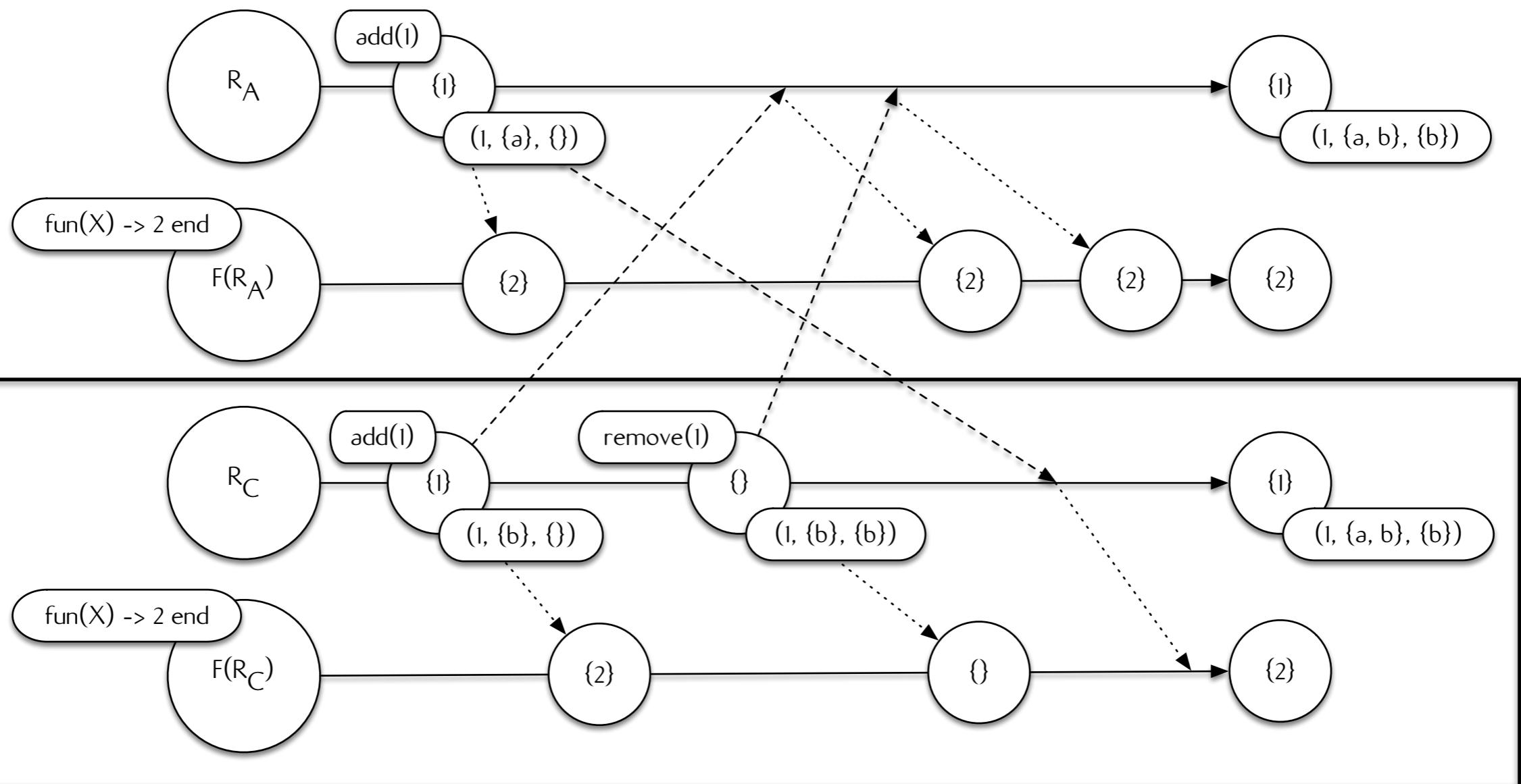


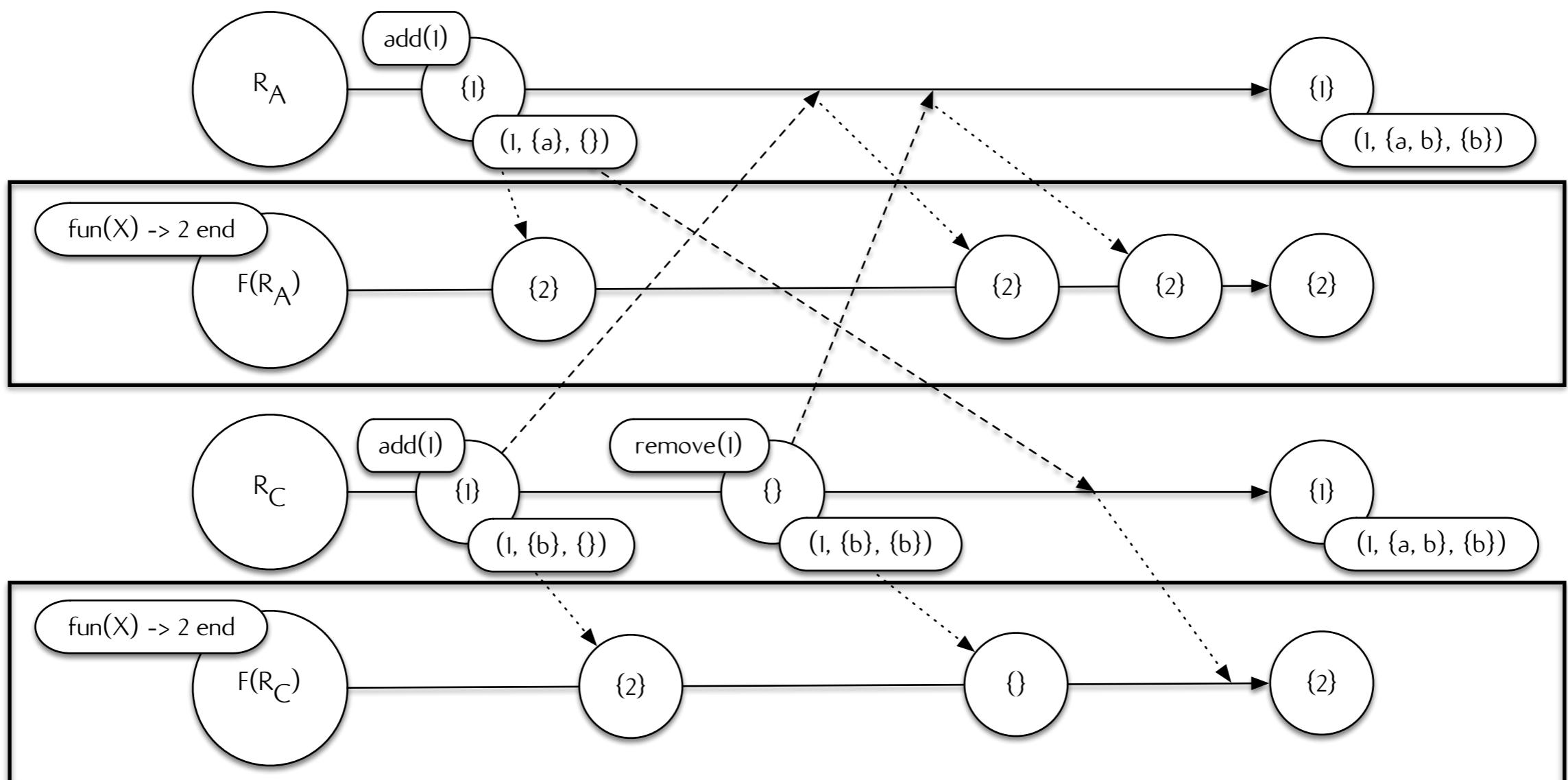


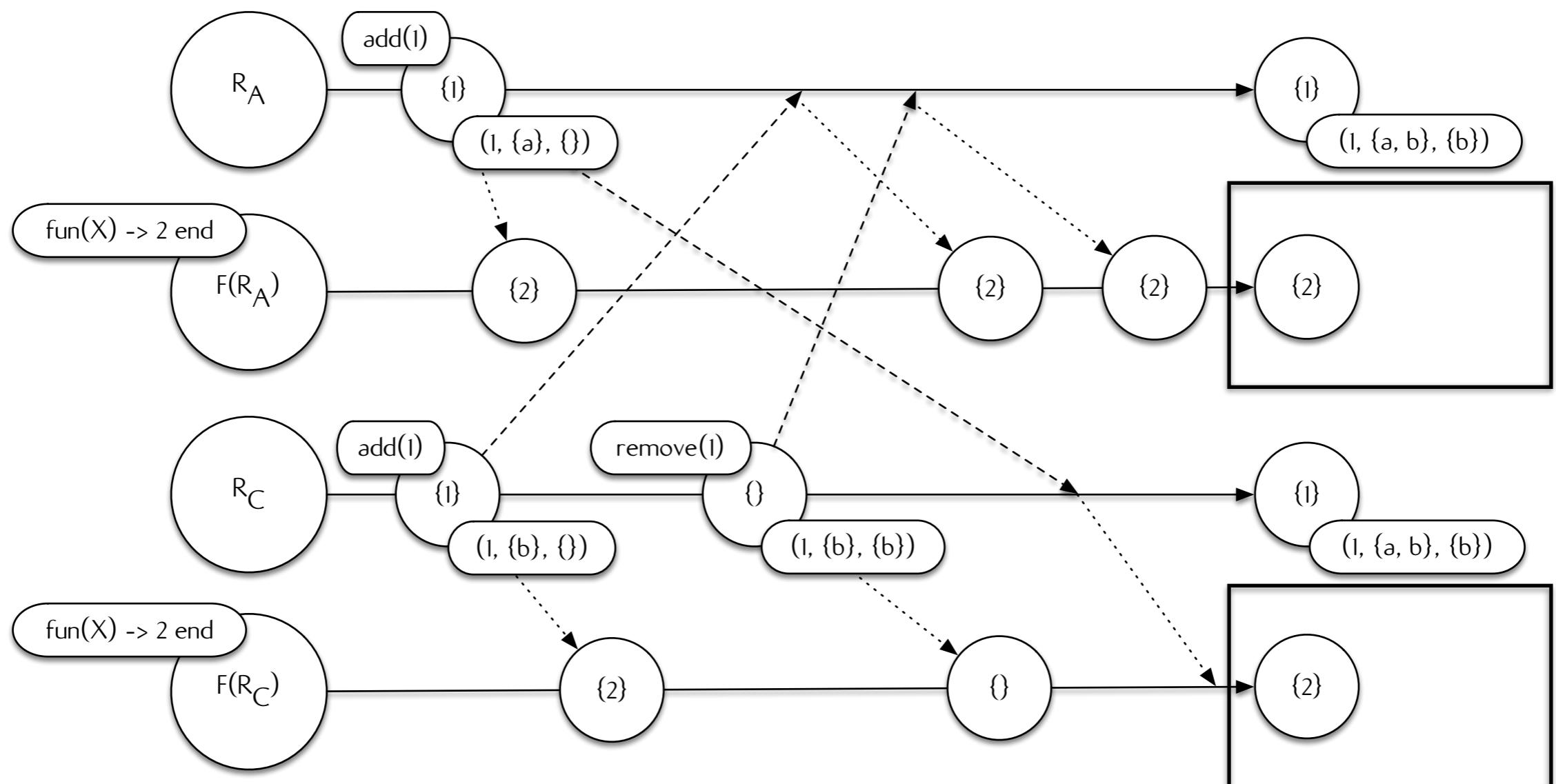
FUNCTION APPLICATION AND DATA COMPOSITION IS
NONTRIVIAL











COMPOSITION:

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

Brown, Russell, et al. "Riak dt map: A composable, convergent replicated dictionary." Proceedings of the First Workshop on Principles and Practice of Eventual Consistency. ACM, 2014.

Conway, Neil, et al. "Logic and lattices for distributed programming." Proceedings of the Third ACM Symposium on Cloud Computing. ACM, 2012.

Meiklejohn, Christopher. "On the composability of the Riak DT map: expanding from embedded to multi-key structures." Proceedings of the First Workshop on Principles and Practice of Eventual Consistency. ACM, 2014.

LASP LANGUAGE

LASP:

DISTRIBUTED RUNTIME
IMPLEMENTED AS ERLANG LIBRARY
USES RIAK-CORE

LASP:

CRDTS AS STREAMS OF STATE CHANGES
CRDTS CONNECTED BY MONOTONIC PROCESSES
MANY TO ONE MAPPING OF CRDTS

PRIMITIVE OPERATIONS:

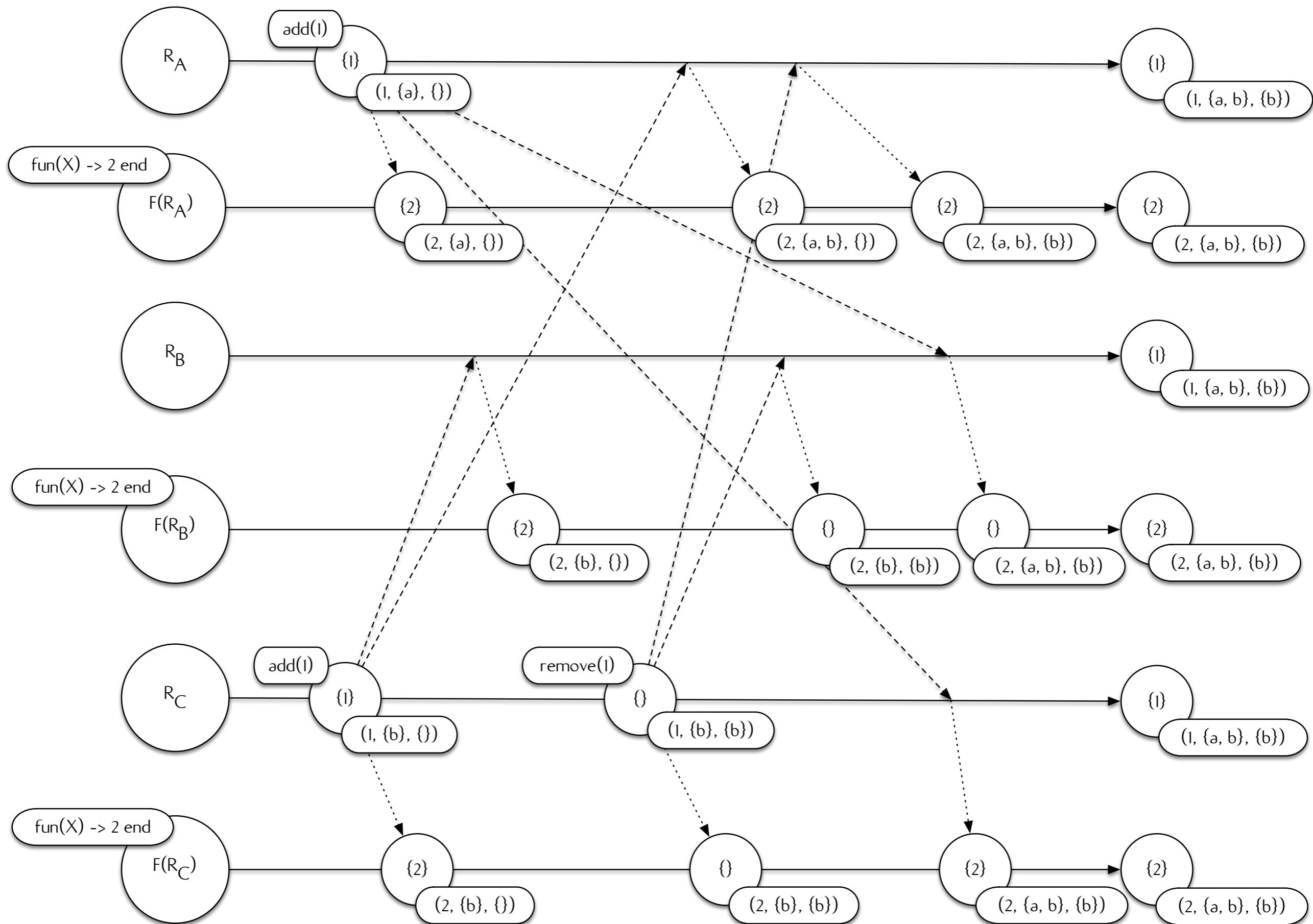
MONOTONIC READ, UPDATE

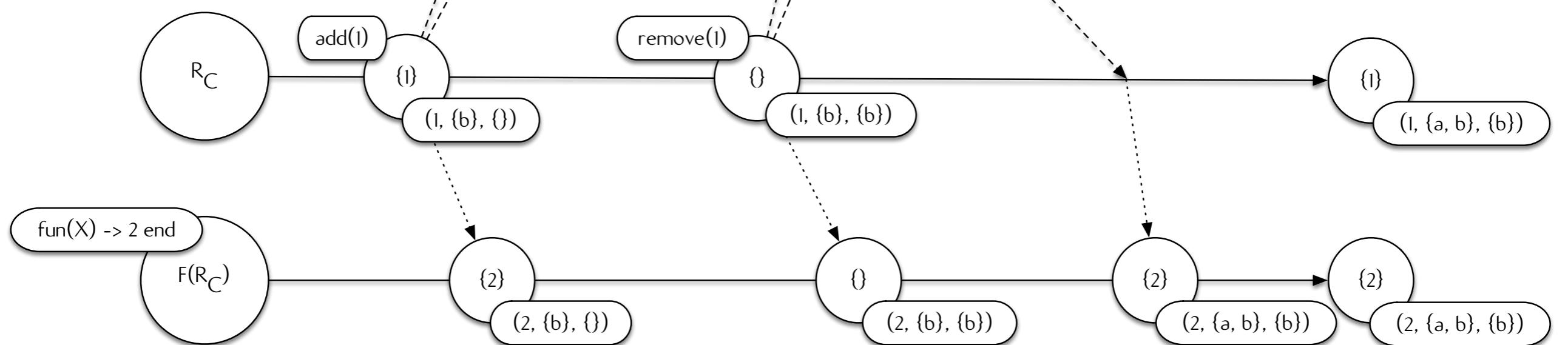
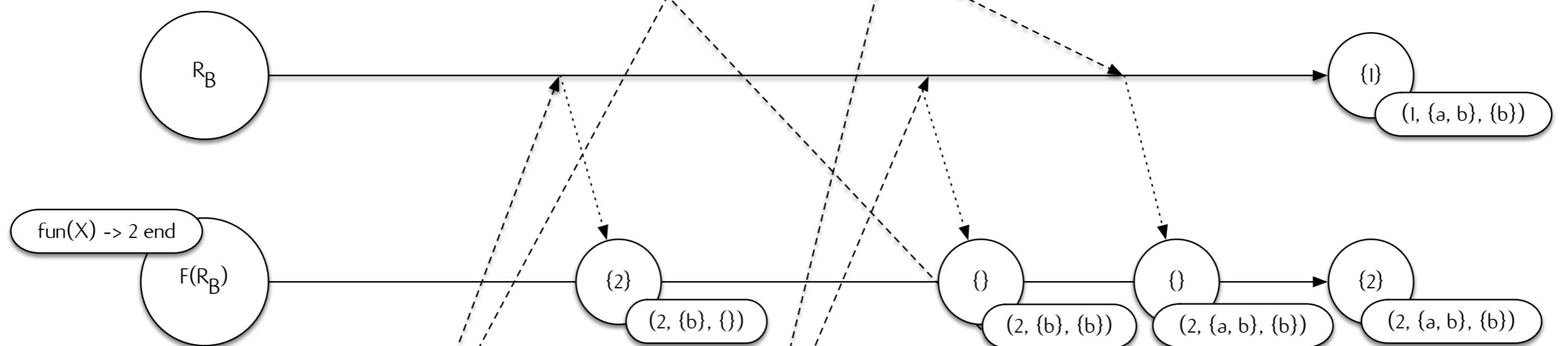
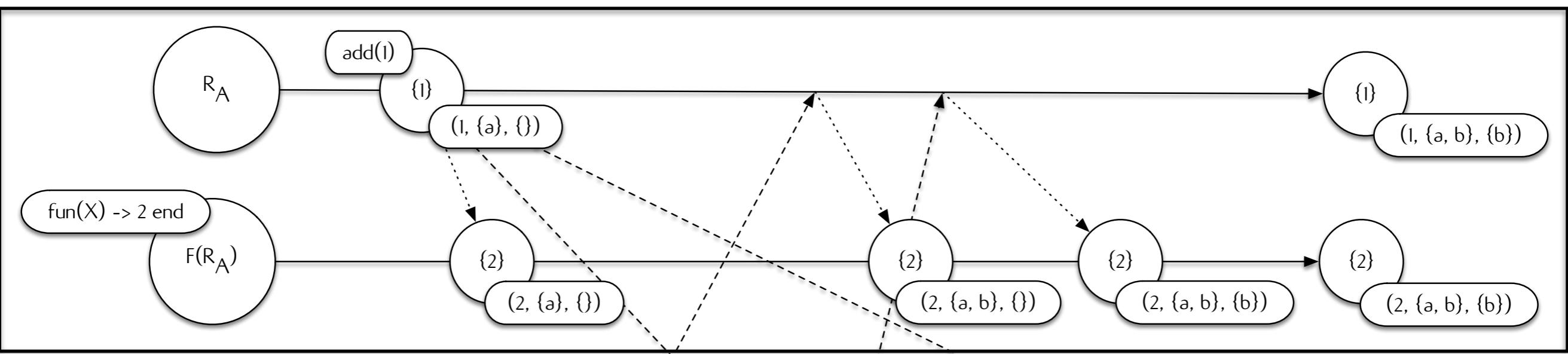
FUNCTIONAL: MAP, FILTER, FOLD

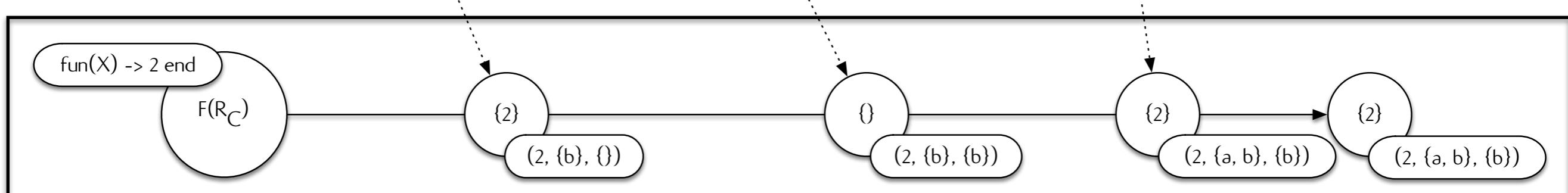
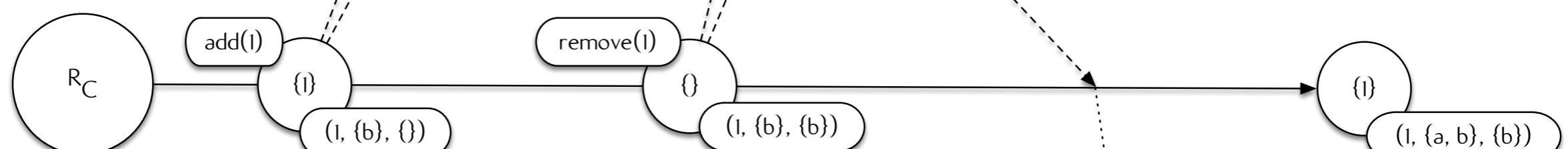
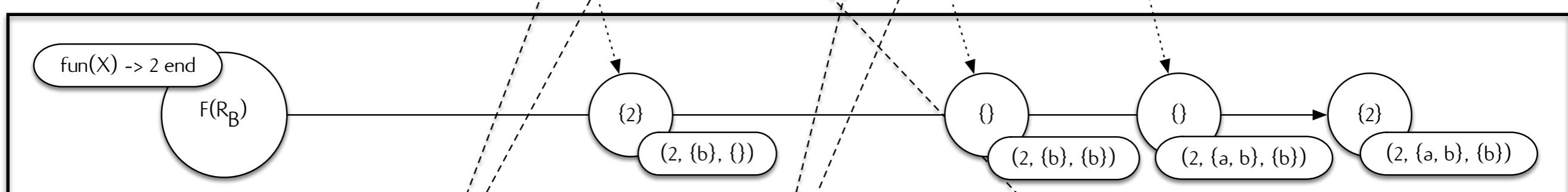
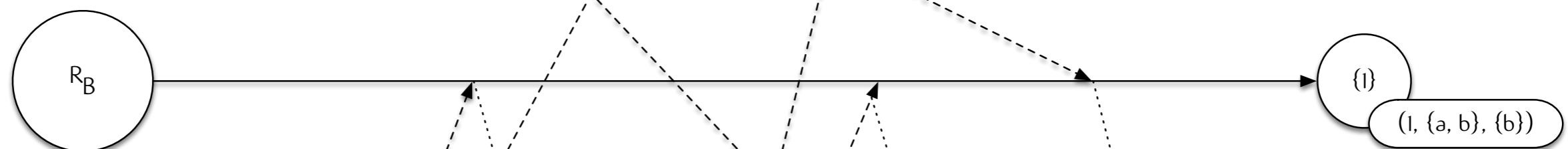
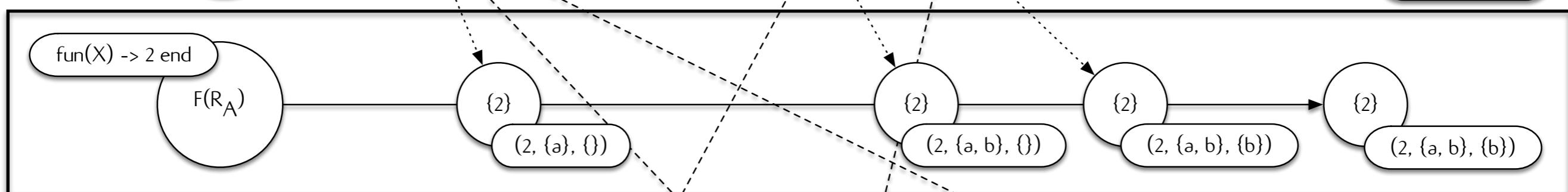
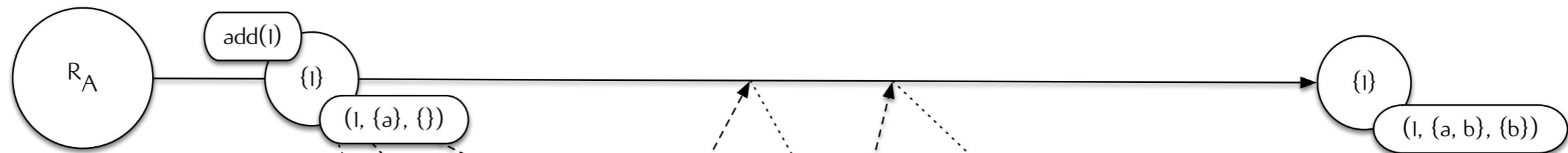
SET-THEORETIC: PRODUCT, UNION, INTERSECTION

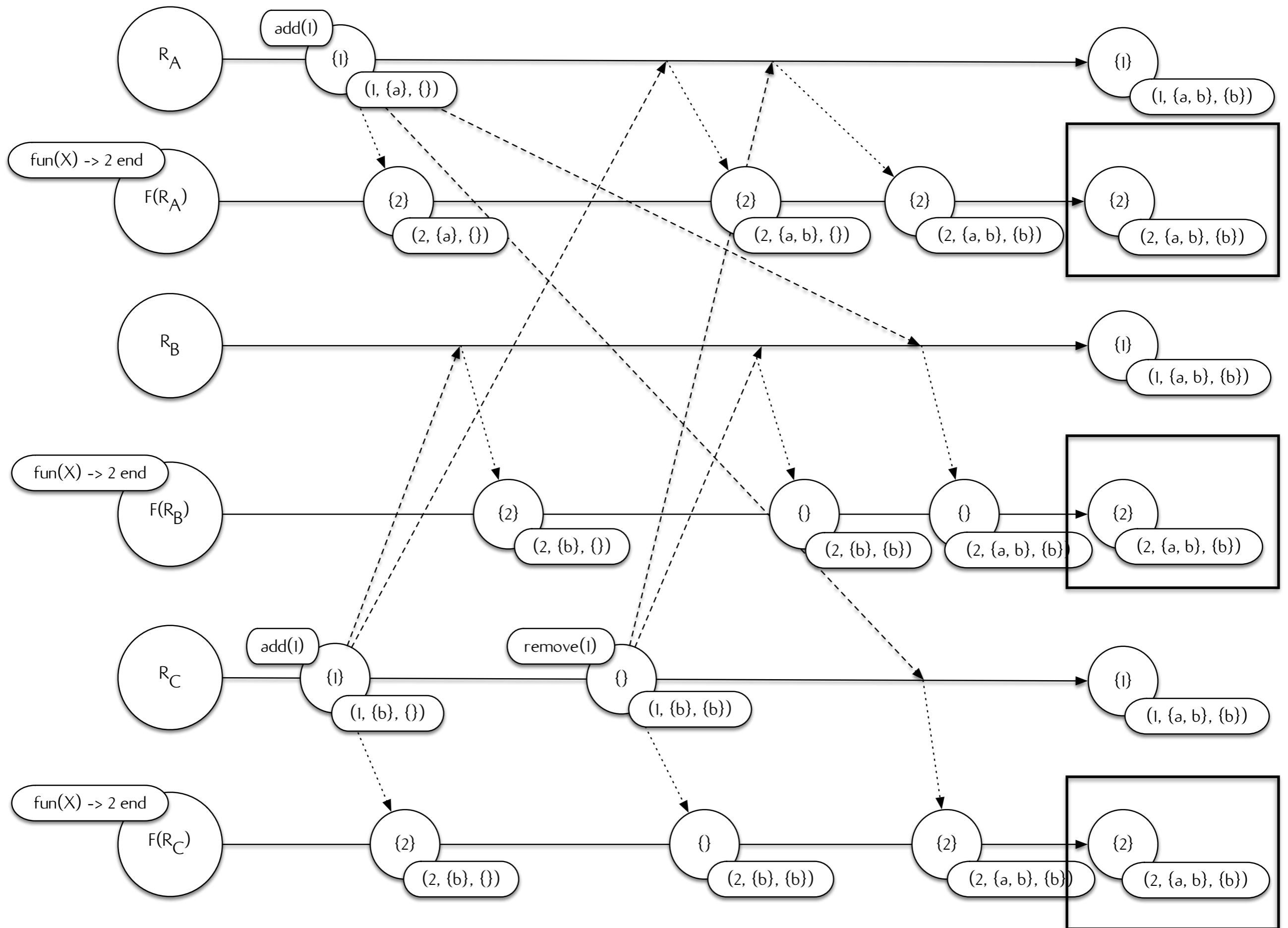
LIFTED TO OPERATE OVER METADATA

EXAMPLE MAP









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.

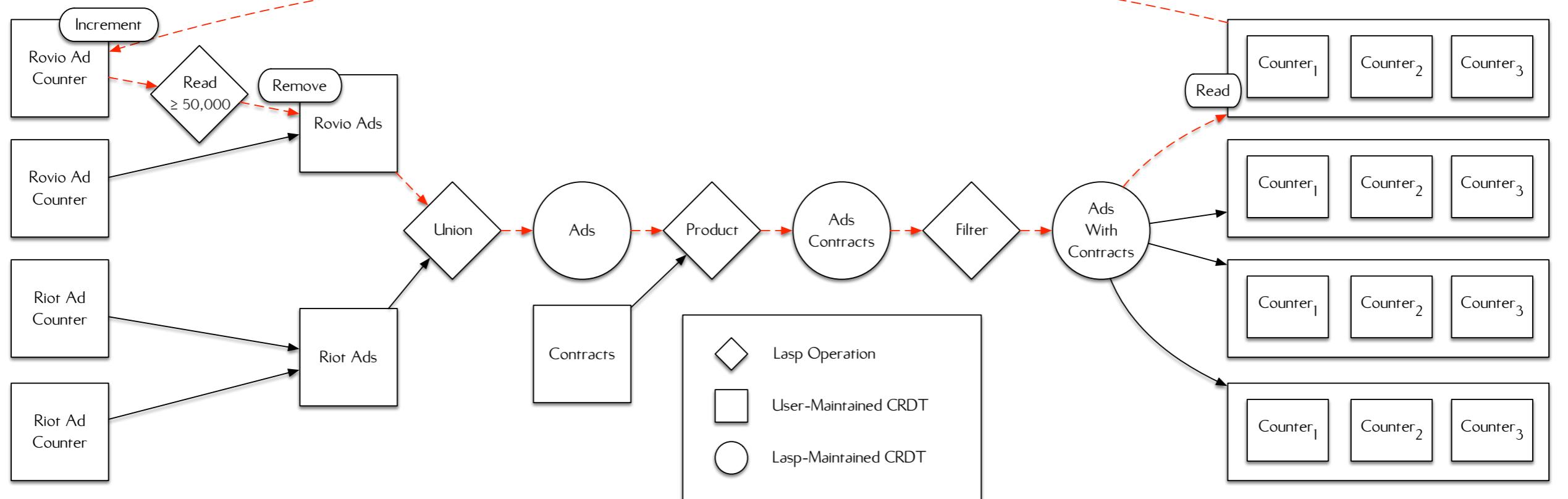
```
ok = lasp_core:map(S1, fun(X) -> X * 2 end, S2, Store),
```

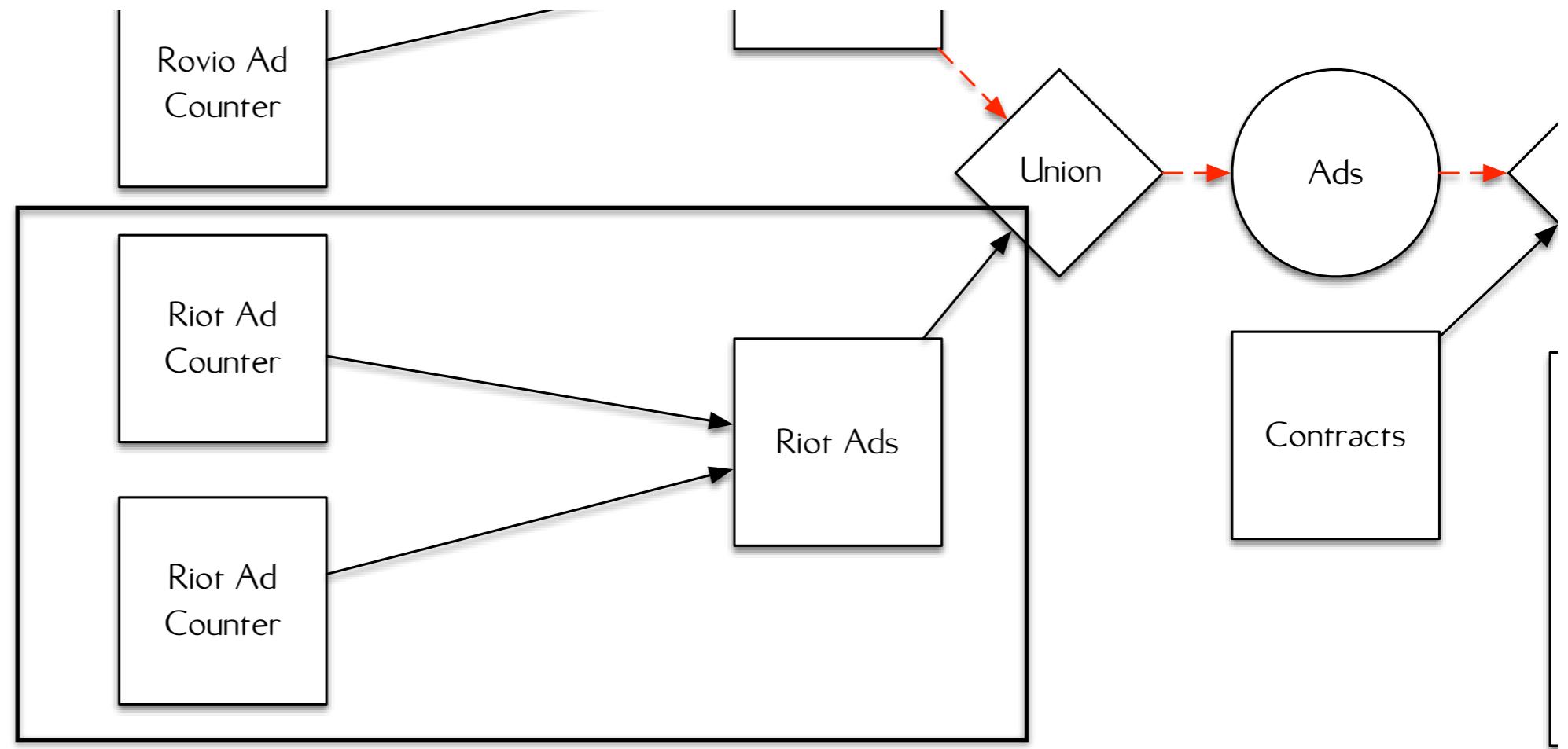
AD COUNTER

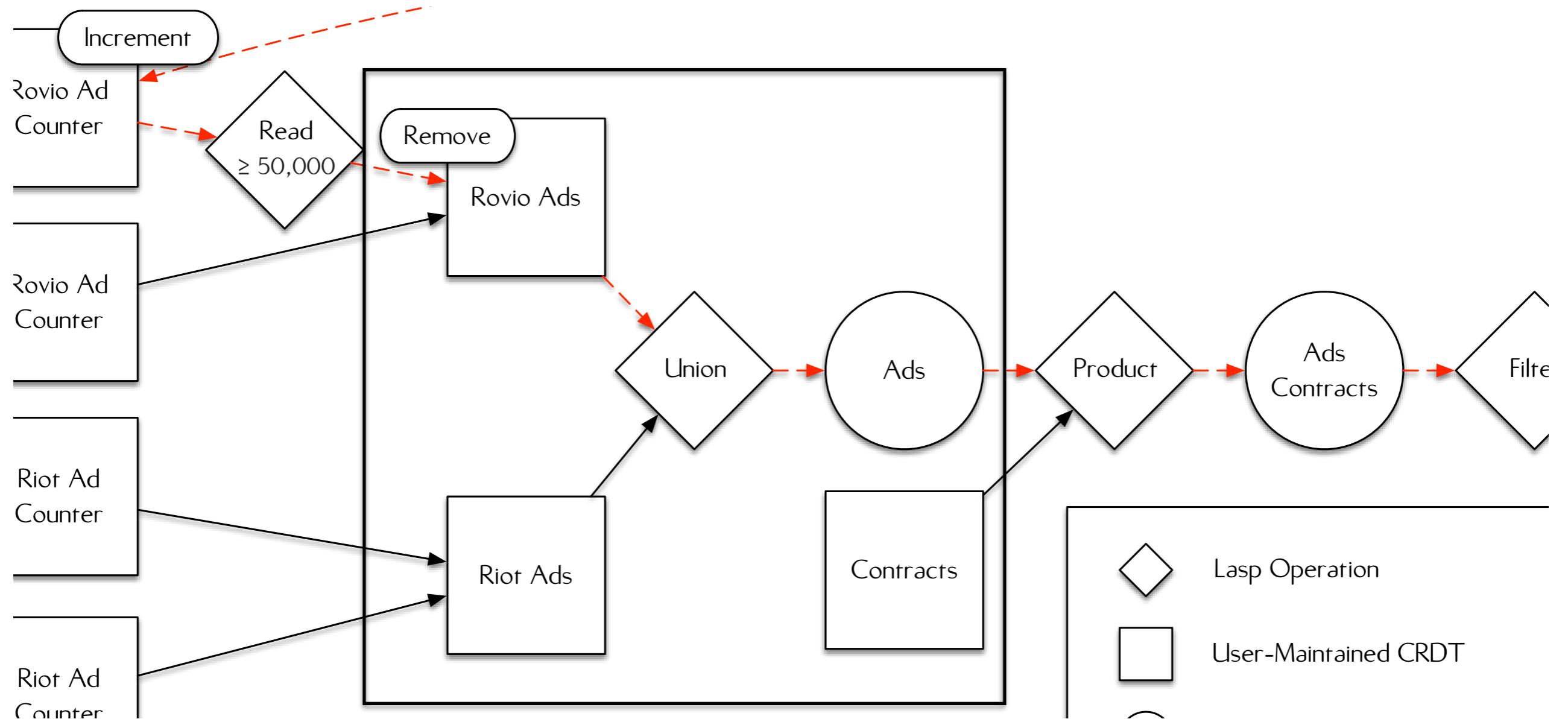
AD COUNTER:

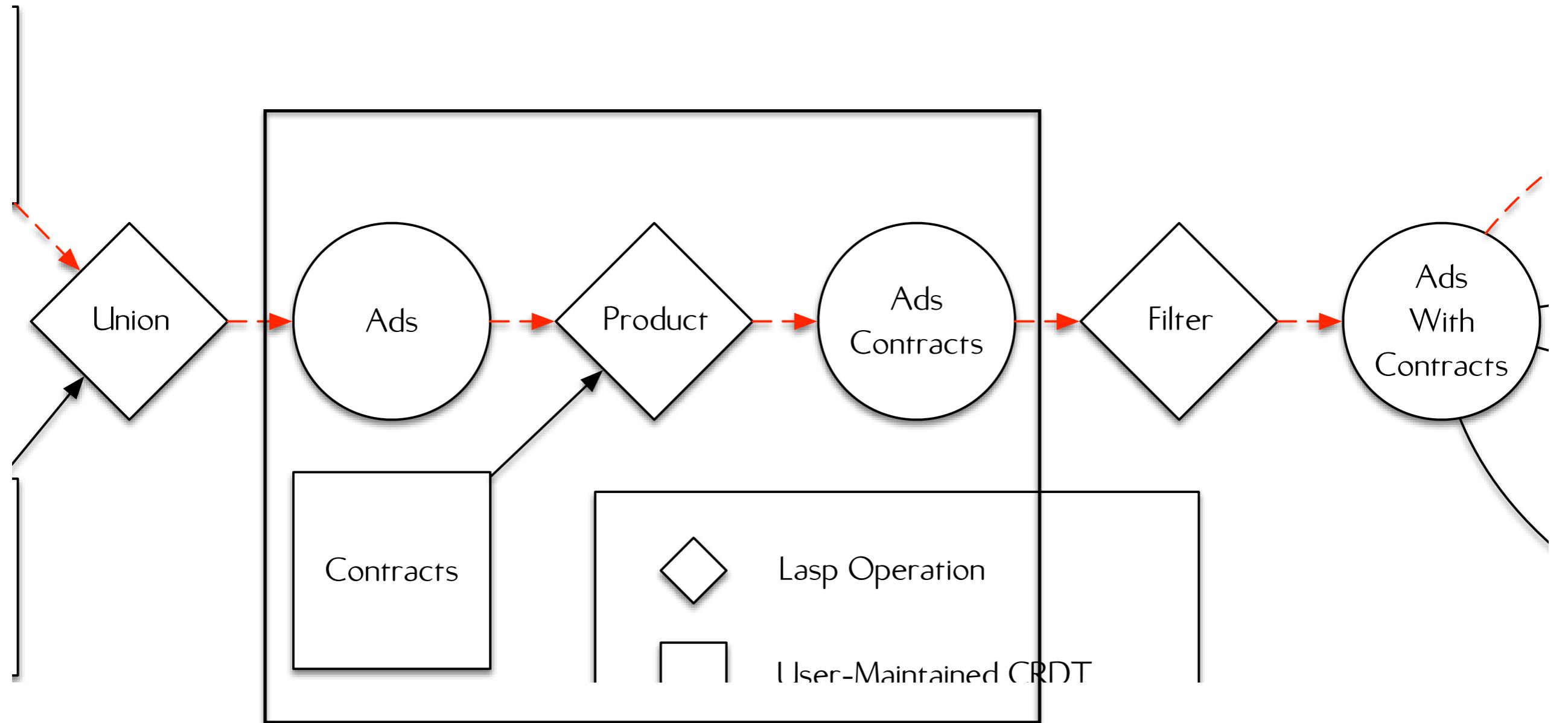
TRACKS AD IMPRESSIONS
PUSHES ADVERTISEMENTS TO THE CLIENT
DISABLES AD AT 50,000+ IMPRESSIONS
CLIENTS DISPLAY ADS WHEN OFFLINE

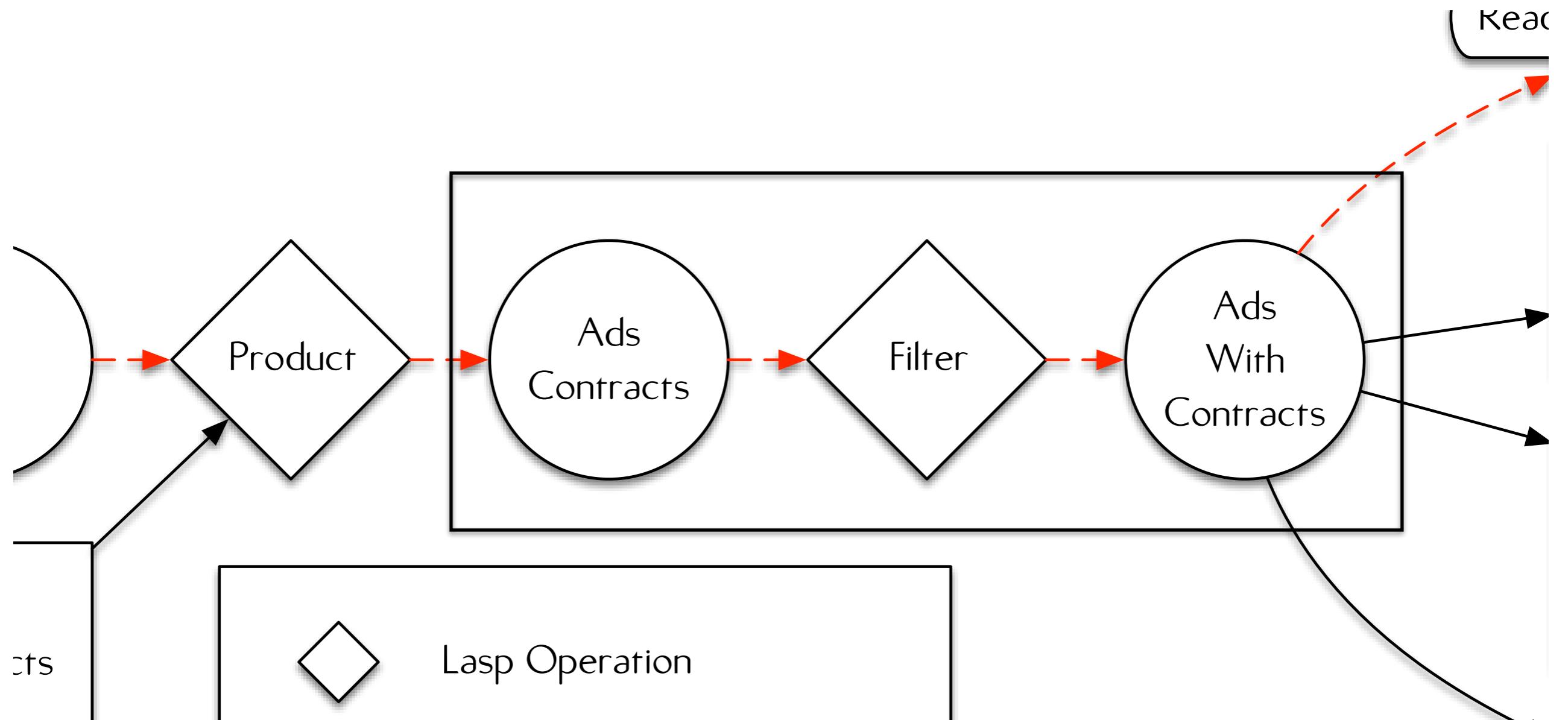
AD COUNTER INFORMATION FLOW

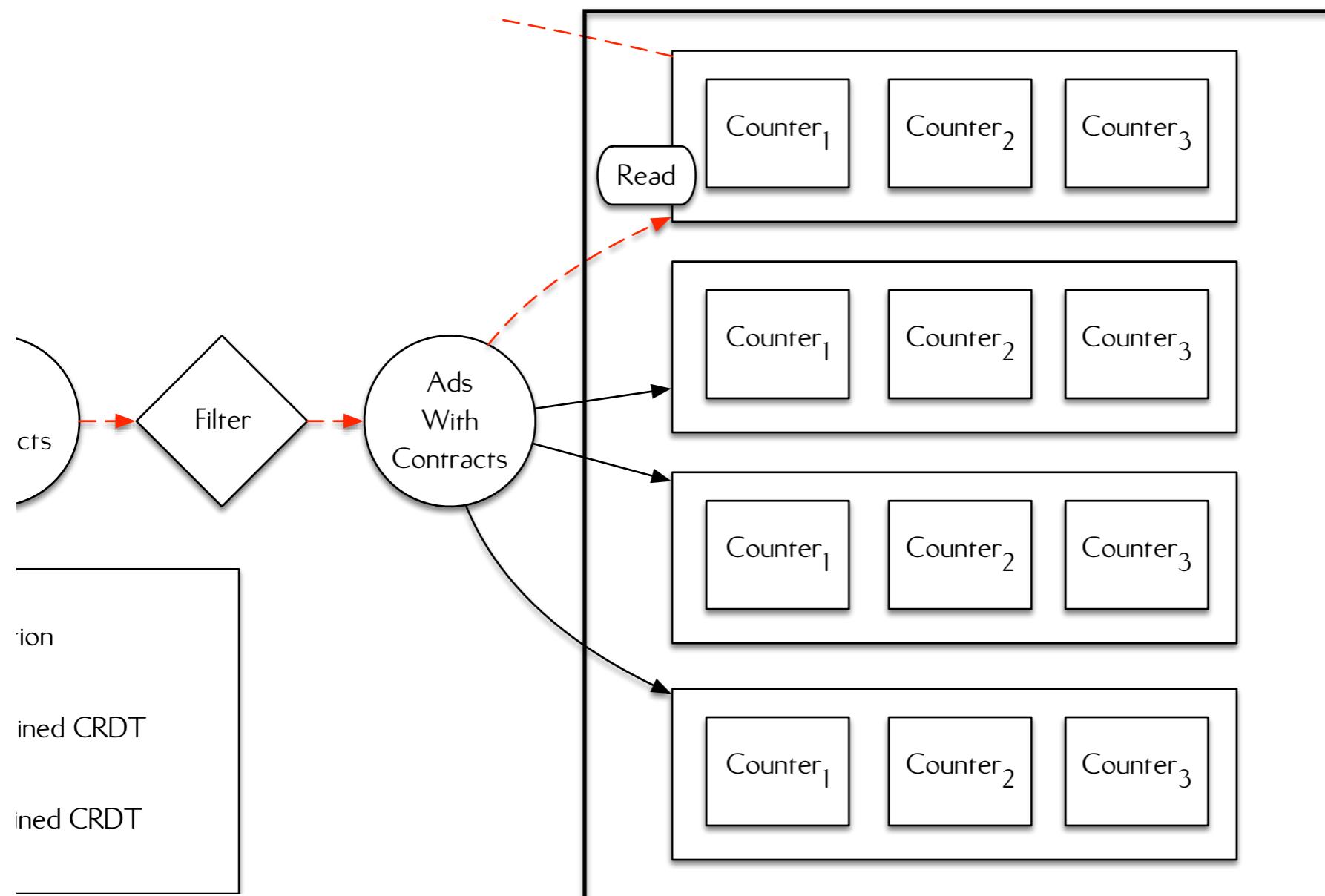


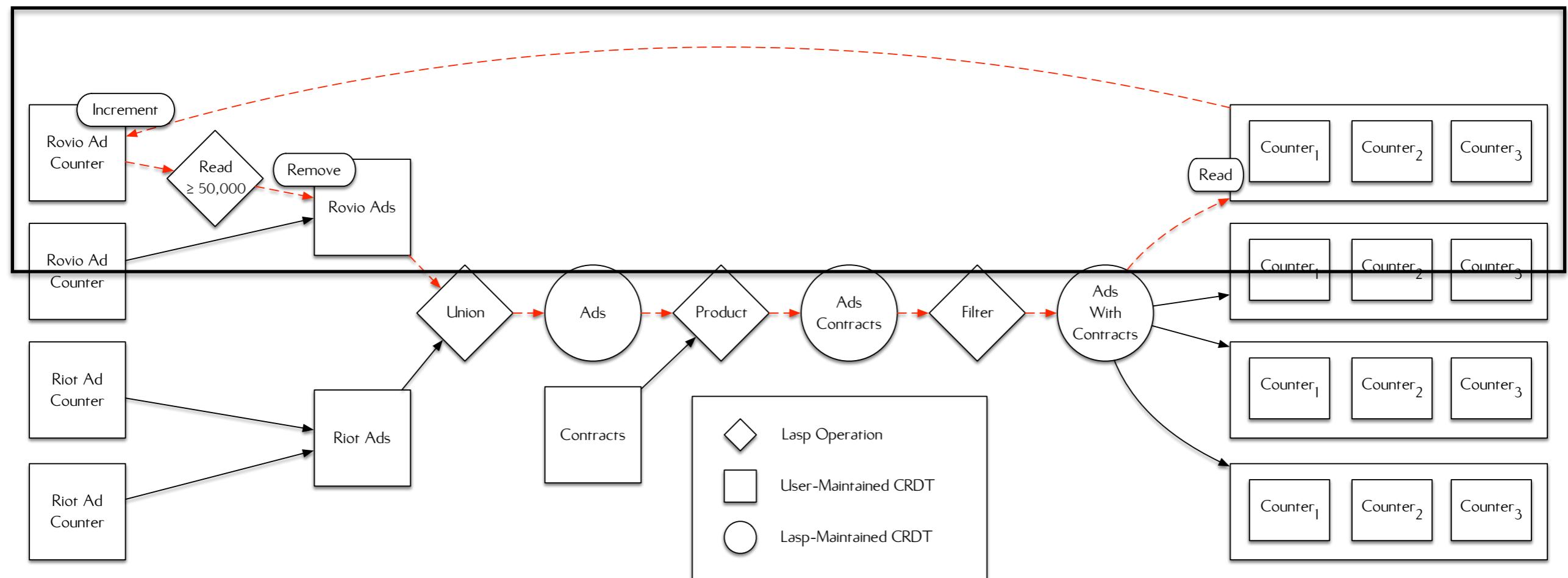


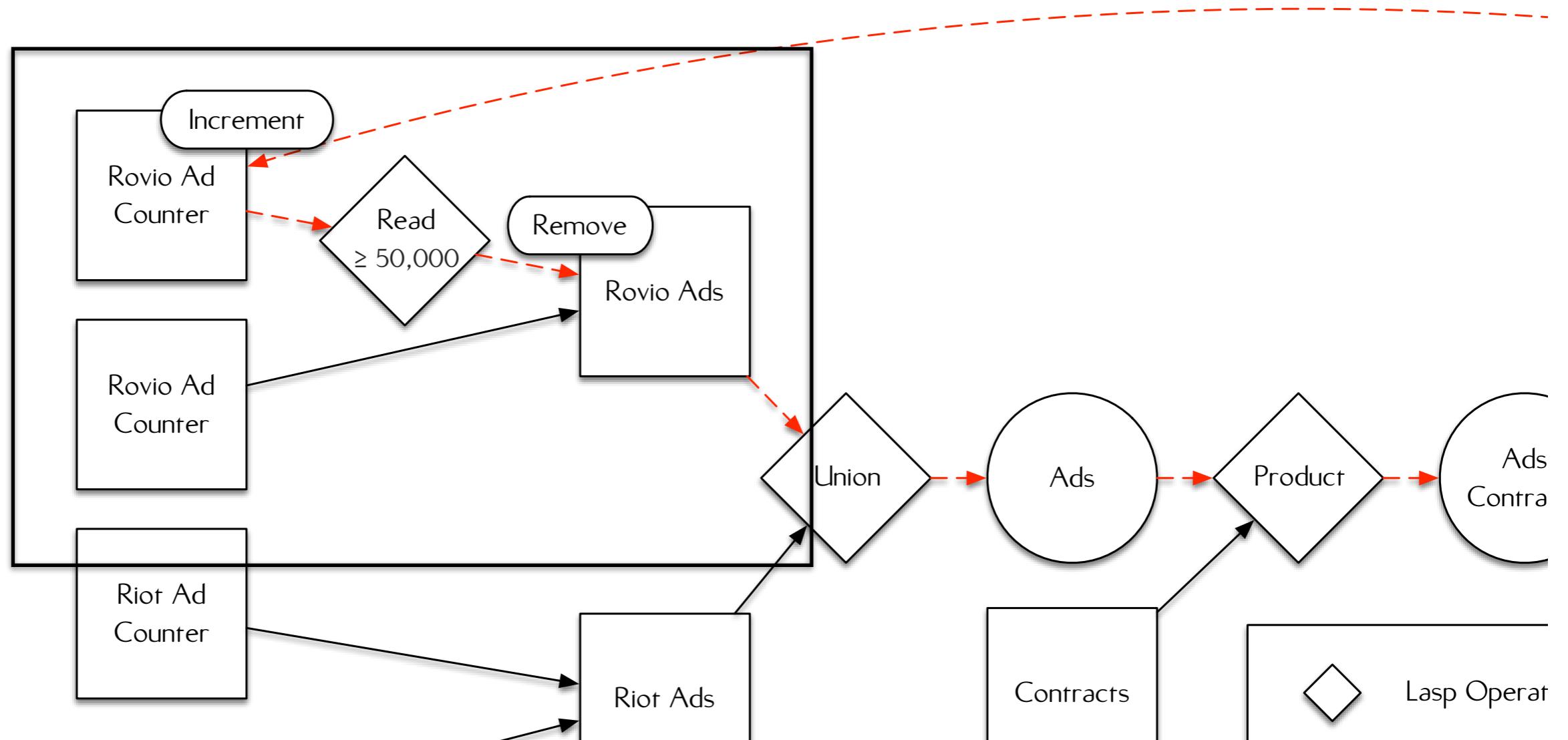


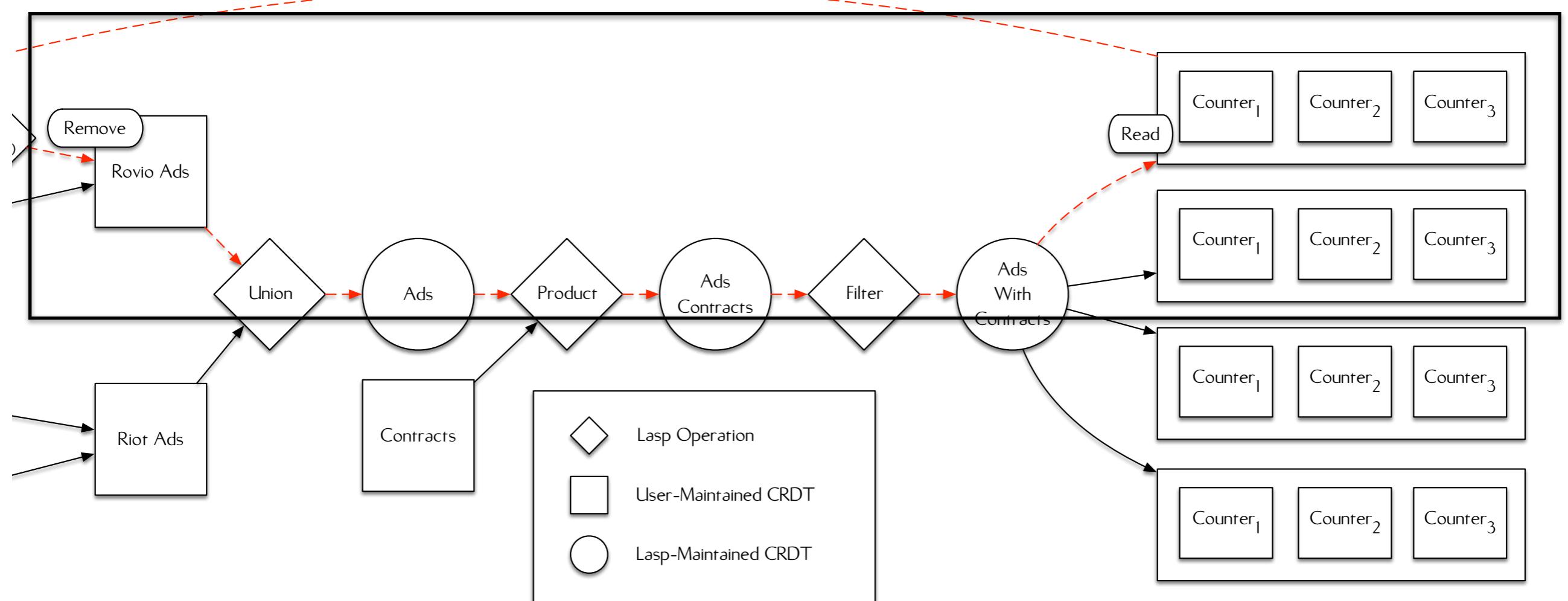












INFORMATION FLOW:

MONOTONIC

METADATA TO PREVENT DUPLICATE PROPAGATION

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.
```

```
%% @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:v4() end, lists:seq(1, 10)),
lager:info("Rovio Ad Identifiers are: ~p", [RovioAdIds]),

TriforkAdIds = lists:map(fun(_) -> druid:v4() 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} = lisp:declare(?SET),
lists:map(fun(Id) ->
    %% Generate a G-Counter.
    {ok, CounterId} = lisp:declare(?COUNTER),
    %% Add it to the advertisement set.
    {ok, _} = lisp:update(RovioAds,
        {add, #ad{id=Id, counter=CounterId}},
        undefined)
    %% Union ads.
    {ok, Ads} = lisp:declare(?SET),
    ok = lisp:union(RovioAds, TriforkAds, Ads),
    %% For each identifier, generate a contract.
    {ok, Contracts} = lisp:declare(?SET),
    lists:map(fun(Id) ->
        {ok, _} = lisp:update(Contracts,
            {add, #contract{id=Id}}),
        undefined)
    %% Compute the Cartesian product of both ads and contracts.
    {ok, AdsContracts} = lisp:declare(?SET),
    ok = lisp:product(Ads, Contracts, AdsContracts),
    %% Filter items by join on item id.
    {ok, AdsWithContracts} = lisp:declare(?SET),
    FilterFun = fun({#ad{id=Id1}, #contract{id=Id2}}) ->
        Id1 =:= Id2
    end,
    ok = lisp: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} = lisp: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, _} = lisp: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} = lisp:declare(?SET),
    %% Get the current advertisement list.
    {ok, {_, _, AdList0}} = lisp: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, Ads]),
        {ok, _} = lisp:update(Servers,
            {add, ServerPid}),
        undefined)
    end, AdList),

```

```

%% Generate a series of unique identifiers.
RovioAdIds = lists:map(fun(_) -> druid:v4() end, lists:seq(1, 10)),
lager:info("Rovio Ad Identifiers are: ~p", [RovioAdIds]),

TriforkAdIds = lists:map(fun(_) -> druid:v4() 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} = lisp:declare(?SET),
lists:map(fun(Id) ->
    %% Generate a G-Counter.
    {ok, CounterId} = lisp:declare(?COUNTER),
    %% Add it to the advertisement set.
    {ok, _} = lisp:update(RovioAds,
        {add, #ad{id=Id, counter=CounterId}},
        undefined)
    end, RovioAdIds),

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

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

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

```

```

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

%% Filter items by join on item id.
{ok, AdsWithContracts} = lisp:declare(?SET),
FilterFun = fun({#ad{id=Id1}, #contract{id=Id2}}) ->
    Id1 =:= Id2
end,
ok = lisp: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} = lisp:declare(?SET),
lists:map(fun(Id) ->
    ClientPid = spawn_link(?MODULE, client,
        [Id, AdsWithContracts, undefined]),
    {ok, _} = lisp: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} = lisp:declare(?SET),
lists:map(fun(Ad) ->
    ServerPid = spawn_link(?MODULE, server, [Ad, Ads]),
    {ok, _} = lisp:update(Servers,
        {add, ServerPid}),
    undefined)
end, AdList),

```

```

%% Generate a series of unique identifiers.
RovioAdIds = lists:map(fun(_) -> druid:v4() end, lists:seq(1, 10)),
lager:info("Rovio Ad Identifiers are: ~p", [RovioAdIds]),

TriforkAdIds = lists:map(fun(_) -> druid:v4() 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} = lisp:declare(?SET),
lists:map(fun(Id) ->
    %% Generate a G-Counter.
    {ok, CounterId} = lisp:declare(?COUNTER),
    %% Add it to the advertisement set.
    {ok, _} = lisp:update(RovioAds,
        {add, #ad{id=Id, counter=CounterId}},
        undefined)
    %% Rovio Advertisements.
    end, RovioAdIds),

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

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

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

```

```

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

%% Filter items by join on item id.
{ok, AdsWithContracts} = lisp:declare(?SET),
FilterFun = fun({#ad{id=Id1}, #contract{id=Id2}}) ->
    Id1 =:= Id2
end,
ok = lisp: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} = lisp:declare(?SET),
lists:map(fun(Id) ->
    ClientPid = spawn_link(?MODULE, client,
        [Id, AdsWithContracts, undefined]),
    {ok, _} = lisp:update(Clients,
        {add, ClientPid}),
    %% Clients.
    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} = lisp:declare(?SET),

```

```

%% Get the current advertisement list.
{ok, {_, _, AdList0}} = lisp: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, Ads]),
    {ok, _} = lisp:update(Servers,
        {add, ServerPid}),
    %% Servers.
    end, AdList),

```

```

%% Generate a series of unique identifiers.
RovioAdIds = lists:map(fun(_) -> druid:v4() end, lists:seq(1, 10)),
lager:info("Rovio Ad Identifiers are: ~p", [RovioAdIds]),

TriforkAdIds = lists:map(fun(_) -> druid:v4() 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} = lisp:declare(?SET),
lists:map(fun(Id) ->
    %% Generate a G-Counter.
    {ok, CounterId} = lisp:declare(?COUNTER),
    %% Add it to the advertisement set.
    {ok, _) = lisp:update(RovioAds,
        {add, #ad{id=Id, counter=CounterId}},
        undefined)
    end, RovioAdIds),

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

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

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

```

```

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

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

%% Launch a series of client processes, each of which is responsible
%% for displaying a particular advertisement.
{ok, Clients} = lisp:declare(?SET),
lists:map(fun(Id) ->
    ClientPid = spawn_link(?MODULE, client,
        [Id, AdsWithContracts, undefined]),
    {ok, _) = lisp: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} = lisp:declare(?SET),
lists:map(fun(Ad) ->
    ServerPid = spawn_link(?MODULE, server, [Ad, Ads]),
    {ok, _) = lisp:update(Servers,
        {add, ServerPid}),
    undefined)
end, AdList),

```

```

%% Generate a series of unique identifiers.
RovioAdIds = lists:map(fun(_) -> druid:v4() end, lists:seq(1, 10)),
lager:info("Rovio Ad Identifiers are: ~p", [RovioAdIds]),

TriforkAdIds = lists:map(fun(_) -> druid:v4() 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} = lisp:declare(?SET),
lists:map(fun(Id) ->
    %% Generate a G-Counter.
    {ok, CounterId} = lisp:declare(?COUNTER),
    %% Add it to the advertisement set.
    {ok, _} = lisp:update(RovioAds,
        {add, #ad{id=Id, counter=CounterId}},
        undefined)
    end, RovioAdIds),

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

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

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


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

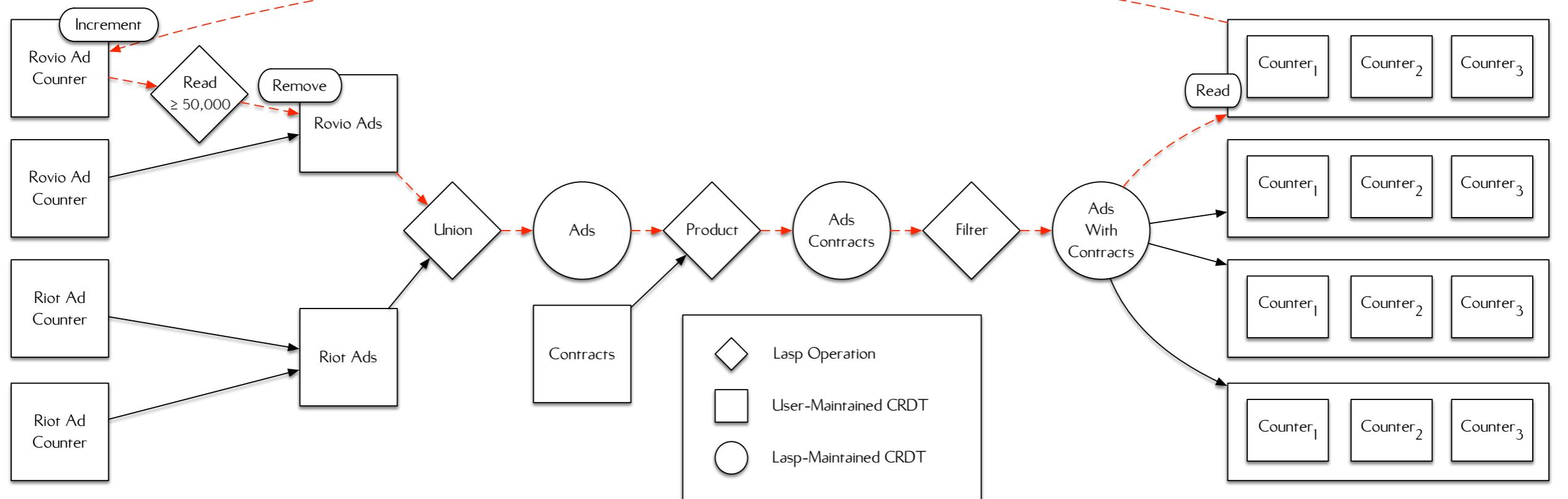

%% Filter items by join on item id.
{ok, AdsWithContracts} = lisp:declare(?SET),
FilterFun = fun({#ad{id=Id1}, #contract{id=Id2}}) ->
    Id1 =:= Id2
end,
ok = lisp:filter(AdsContracts, FilterFun, AdsWithContracts),

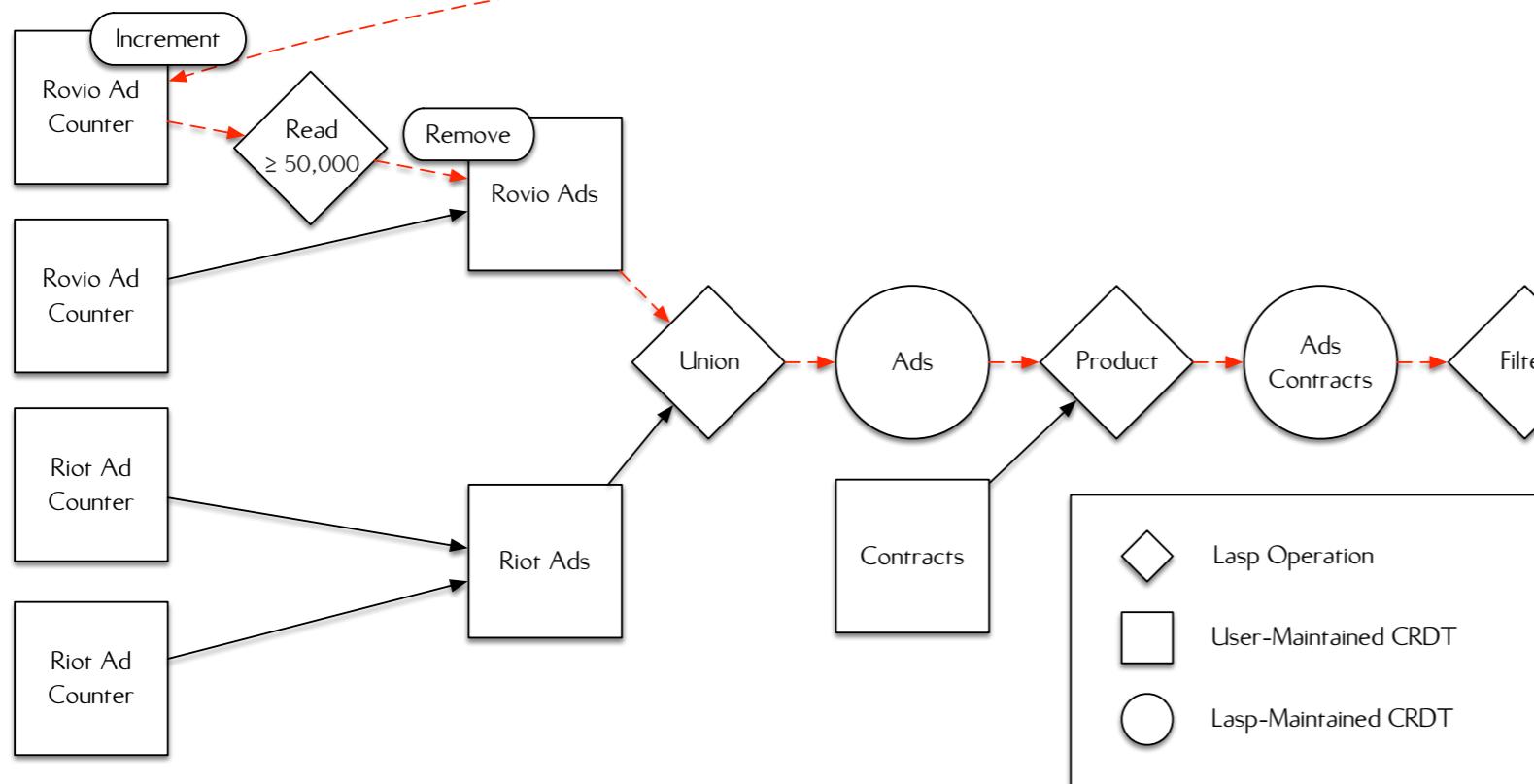

%% Launch a series of client processes, each of which is responsible
%% for displaying a particular advertisement.
{ok, Clients} = lisp:declare(?SET),
lists:map(fun(Id) ->
    %% Each client takes the full list of ads when it starts, and reads
    %% from the variable store.
    ClientPid = spawn_link(?MODULE, client,
        [Id, AdsWithContracts, undefined]),
    {ok, _} = lisp:update(Clients,
        {add, ClientPid}),
    undefined)
end, lists:seq(1,5)),

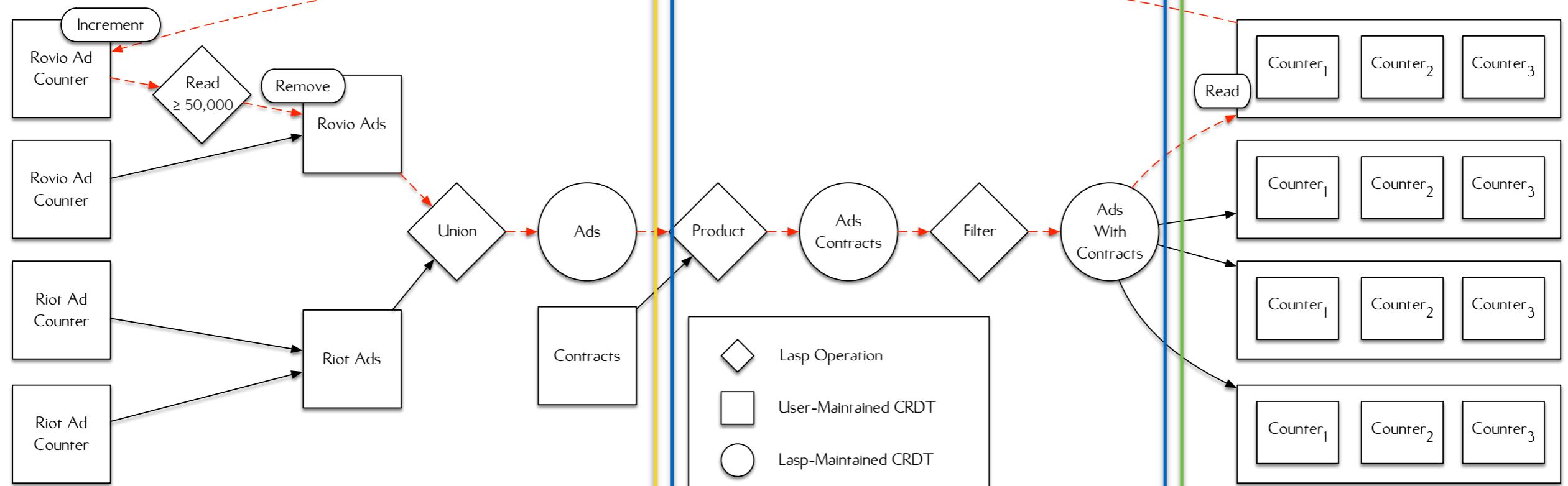

%% Launch a server process for each advertisement, which will block
%% until the advertisement should be disabled.
{ok, Servers} = lisp:declare(?SET),
lists:map(fun(Ad) ->
    %% Create a OR-set for the server list.
    %% Get the current advertisement list.
    {ok, {_, _, AdList0}} = lisp:read(AdsWithContracts),
    AdList = riak_dt_orset:value(AdList0),
    %% For each advertisement, launch one server for tracking it's
    %% impressions and wait to disable.
    ServerPid = spawn_link(?MODULE, server, [Ad, Ads]),
    {ok, _} = lisp:update(Servers,
        {add, ServerPid}),
    undefined)
end, AdList),

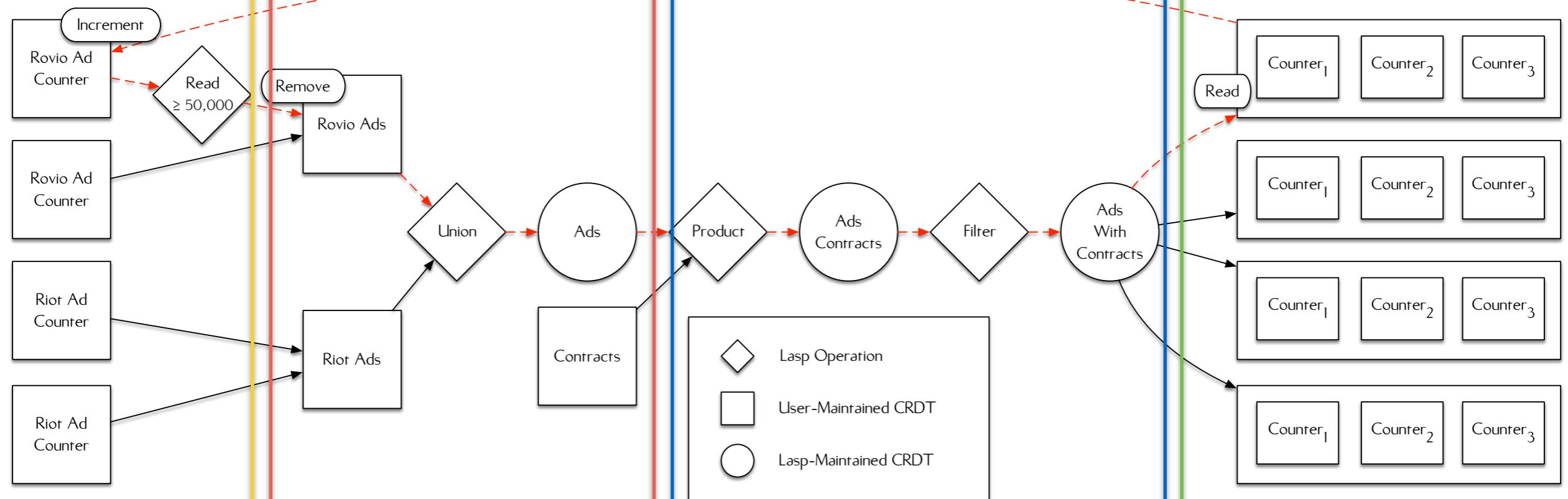
```

AD COUNTER DISTRIBUTION









DISTRIBUTION BOUNDARIES:

ARBITRARY

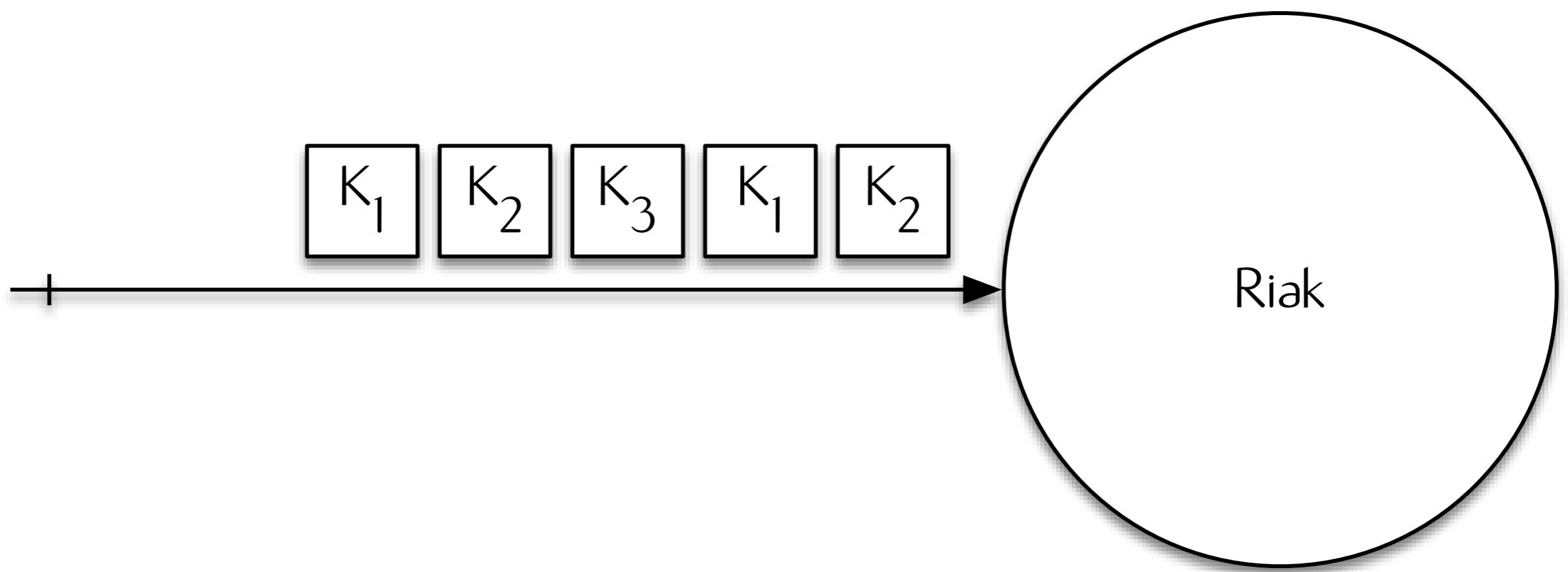
ALLOWS COMPOSITION OF ENTIRE SYSTEM

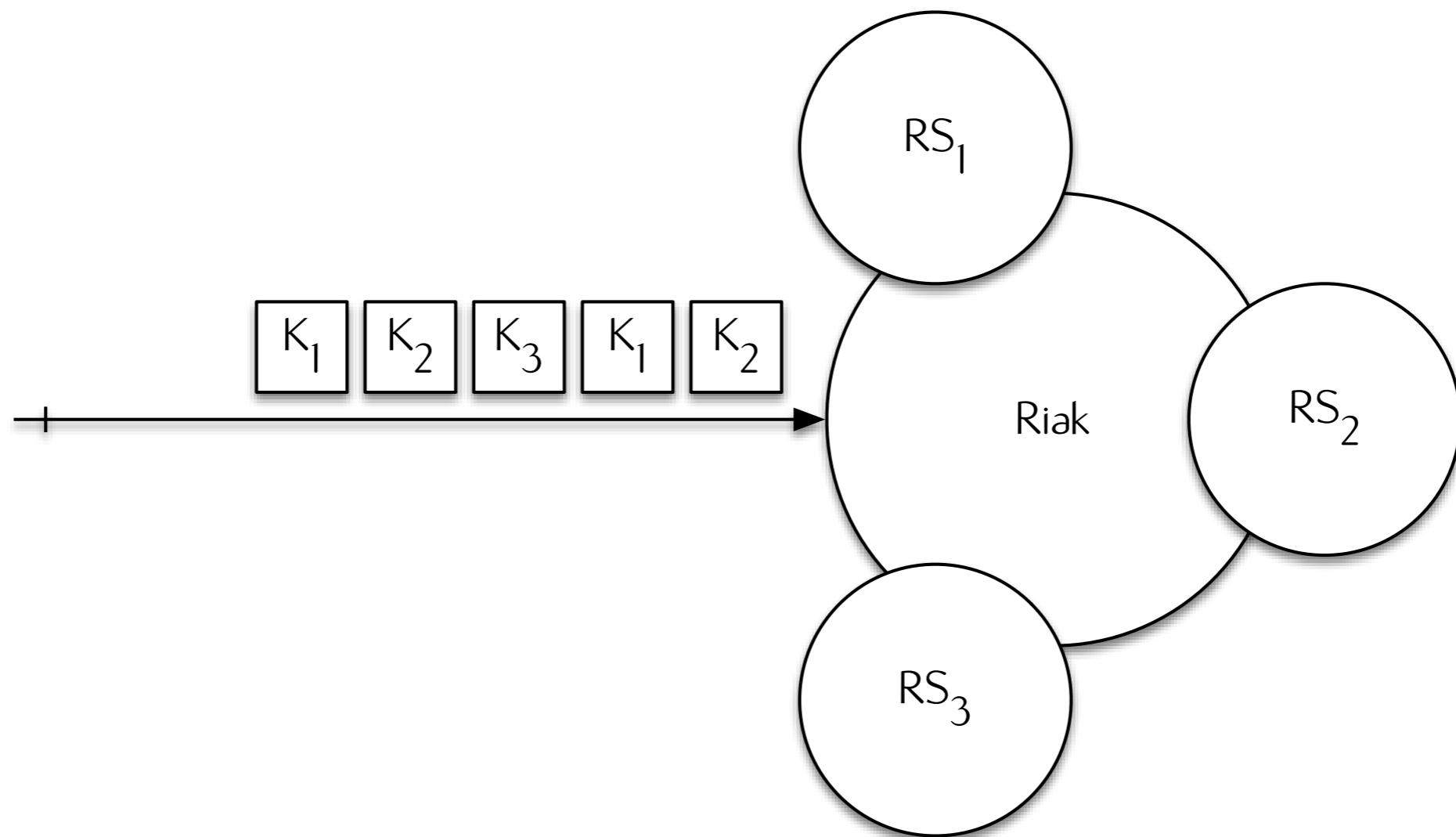
MATERIALIZED VIEWS

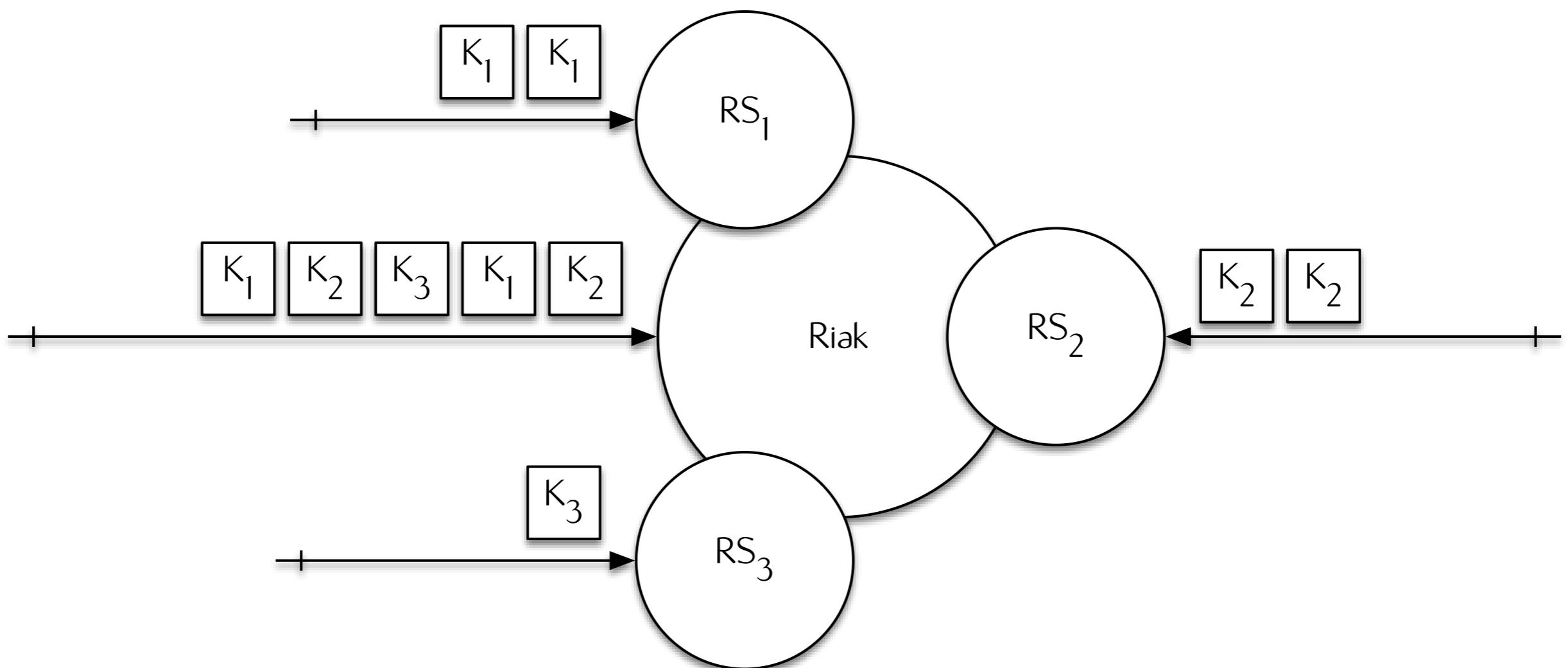
MATERIALIZED VIEWS:

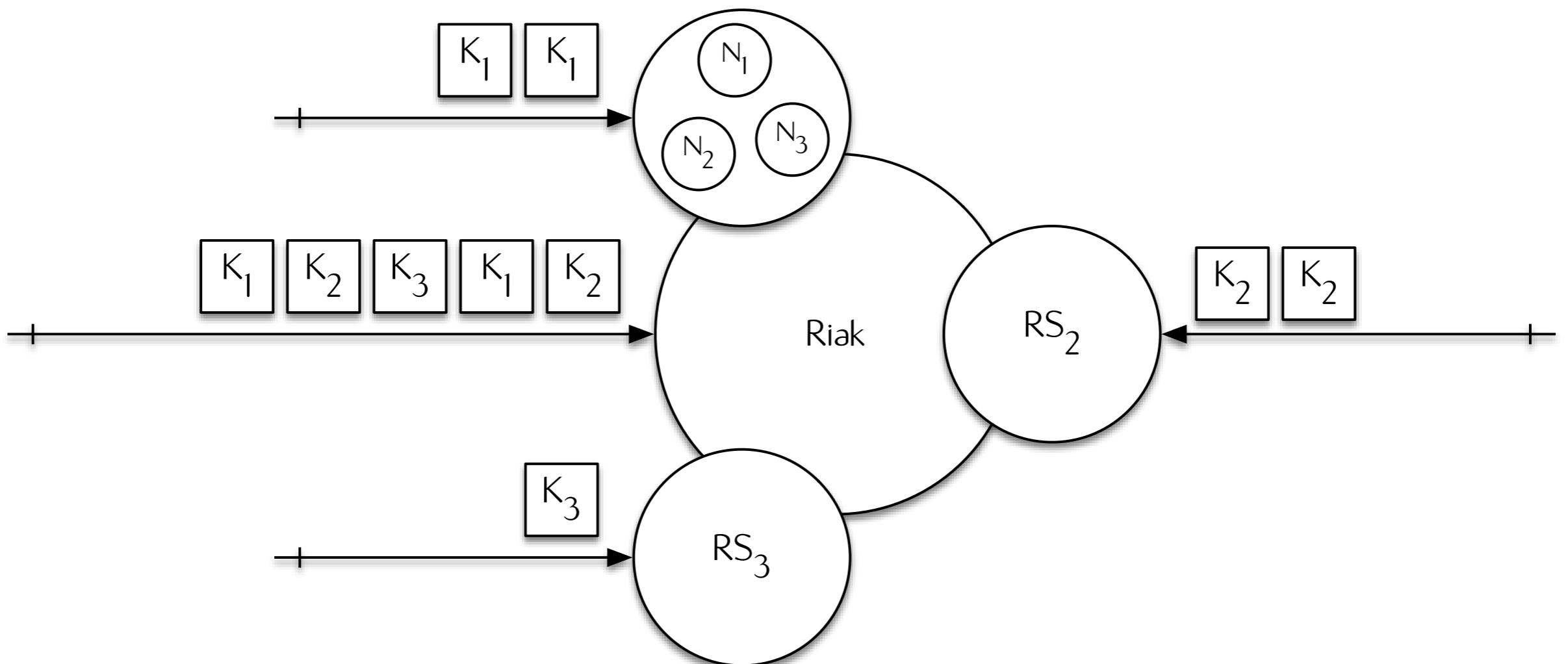
INCREMENTALLY UPDATING, DELTA PROPAGATION
SCHEMA BASED OR DYNAMIC; 2I IN RIAK

DATABASE AS STREAM OF UPDATES

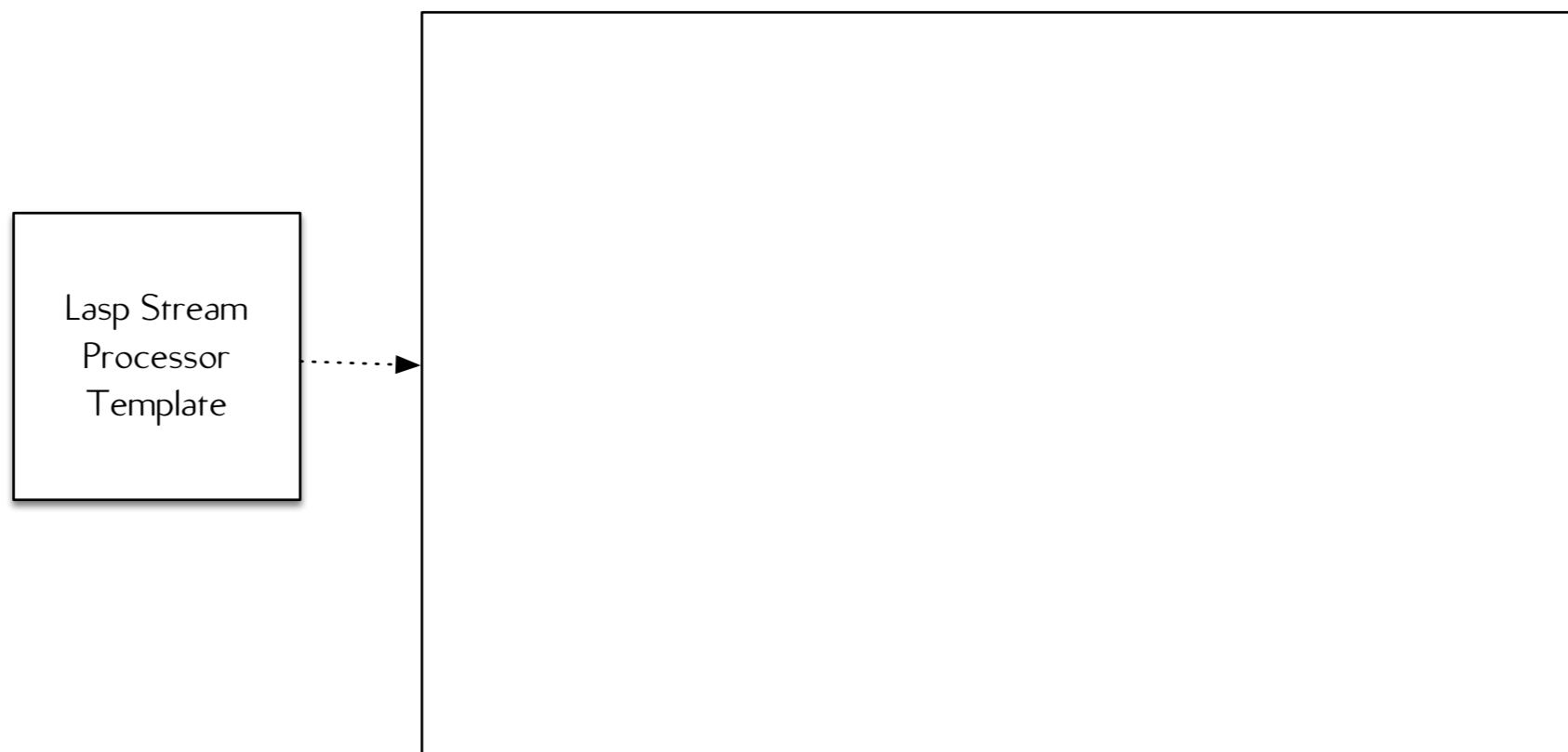
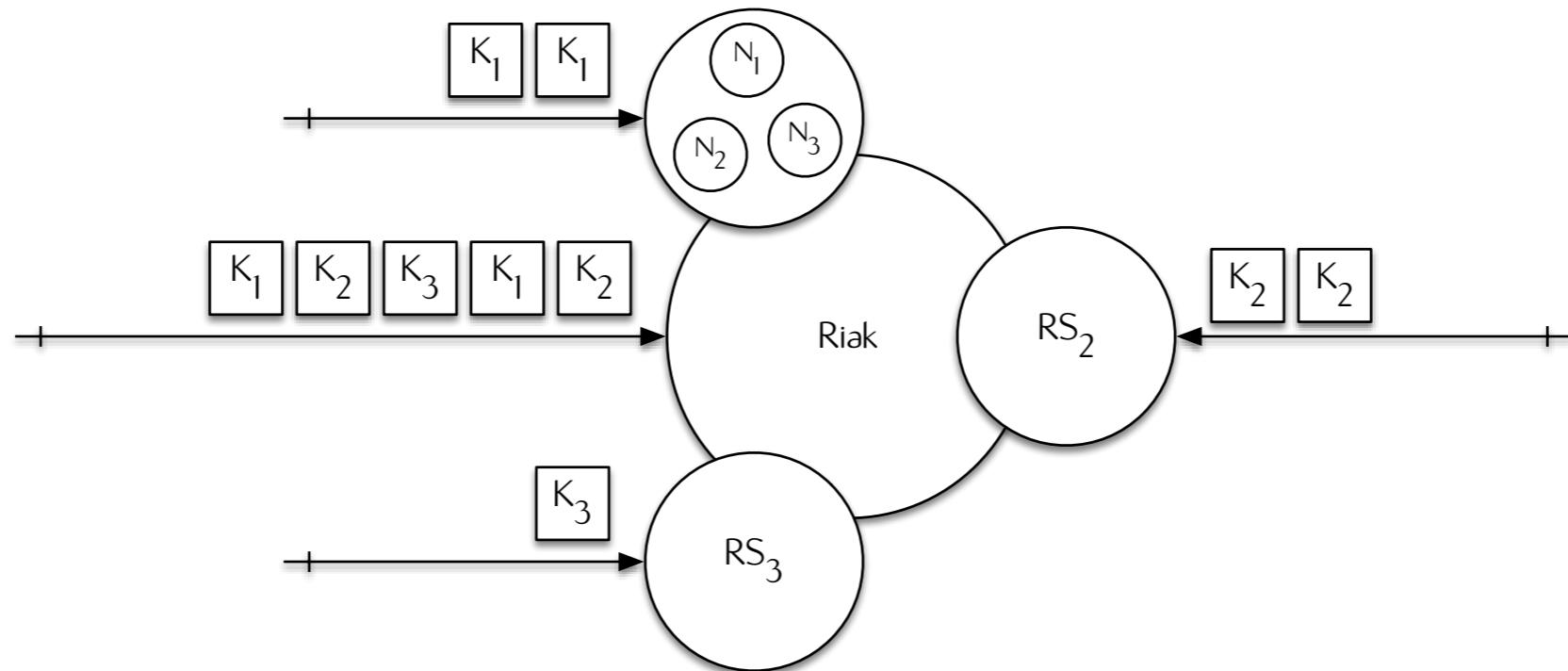


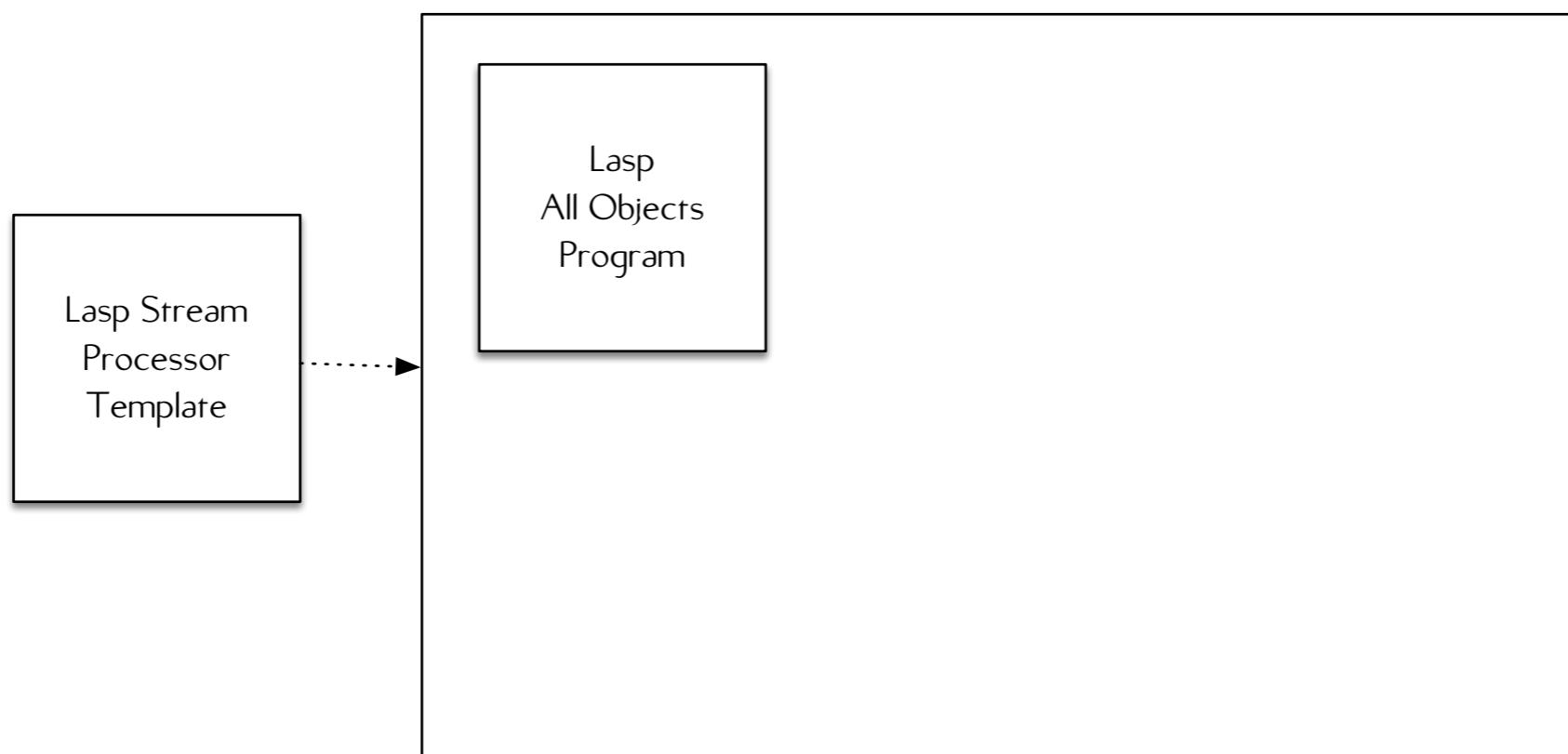
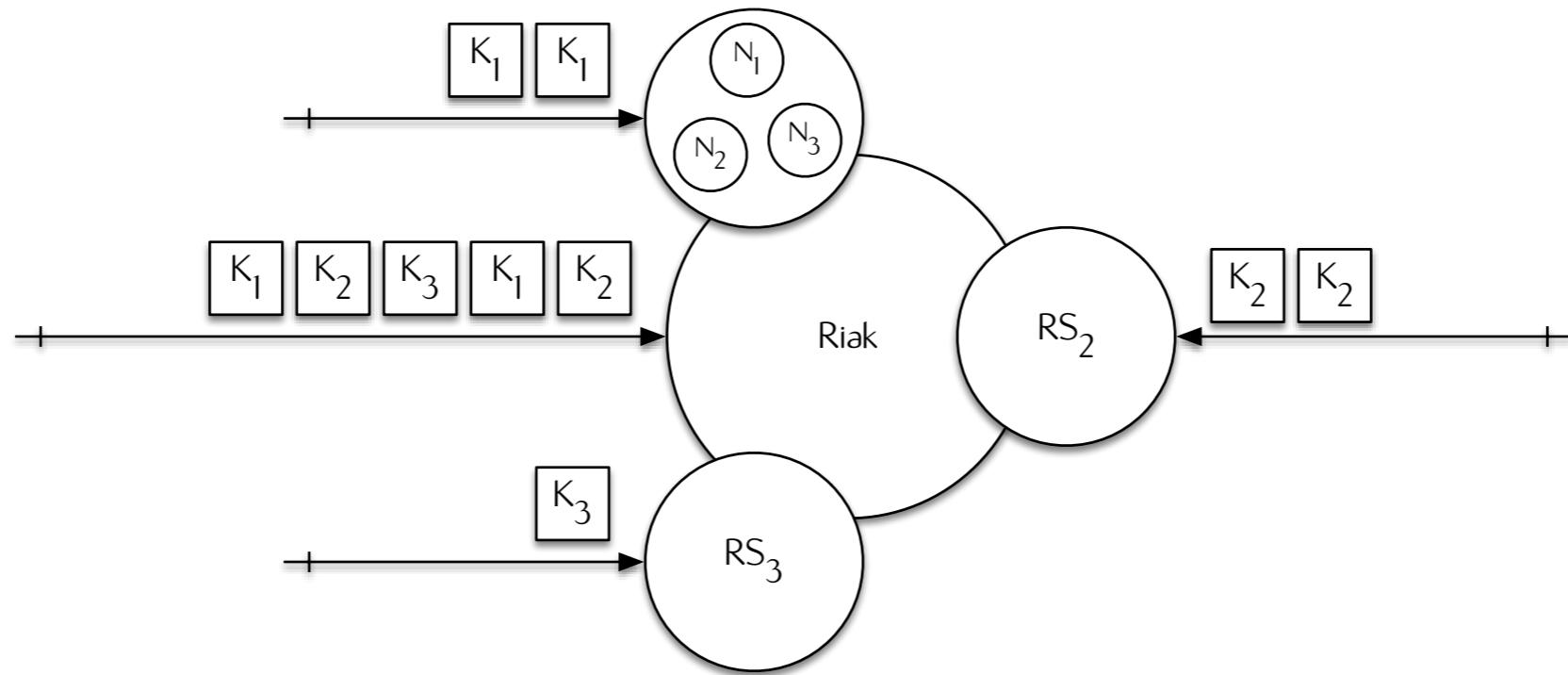


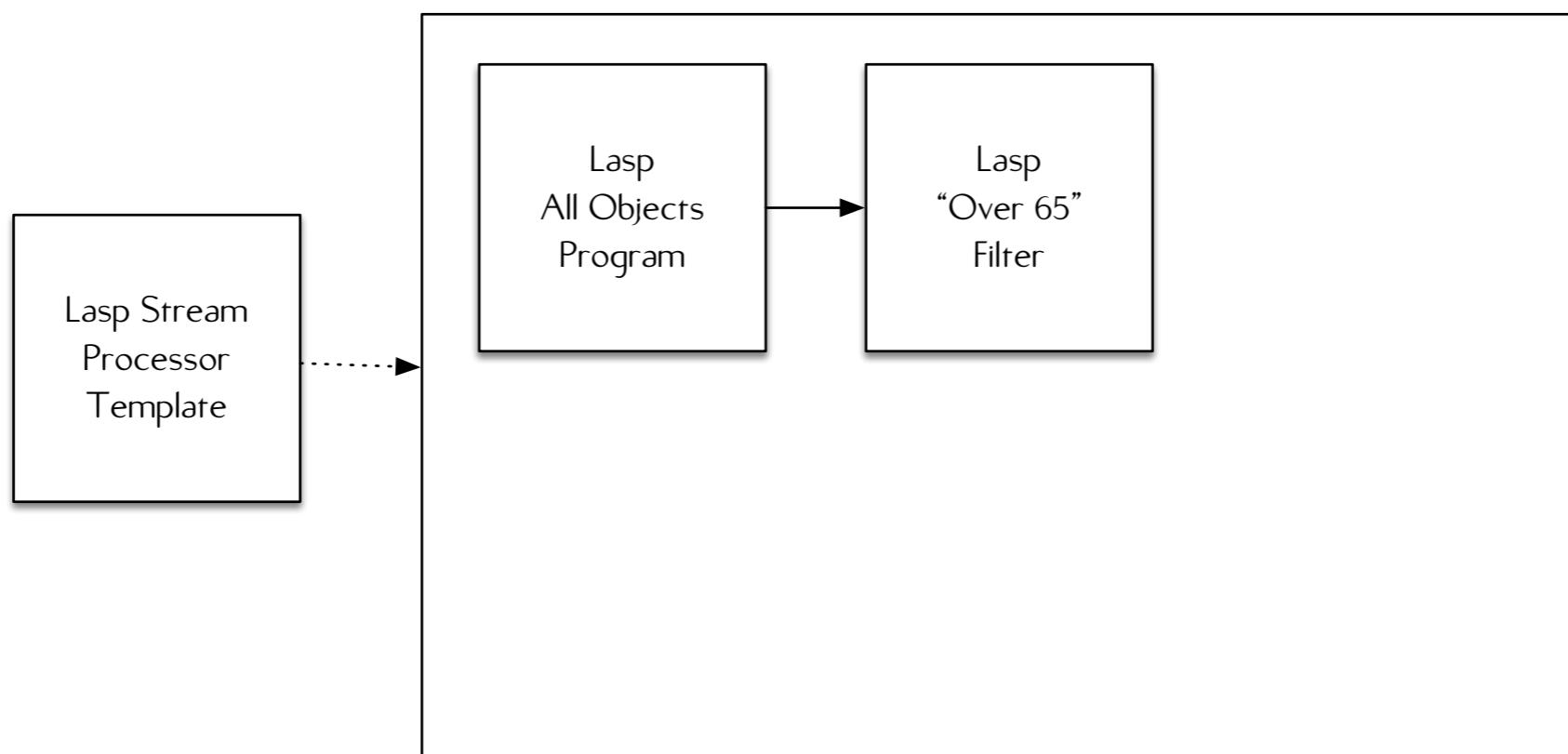
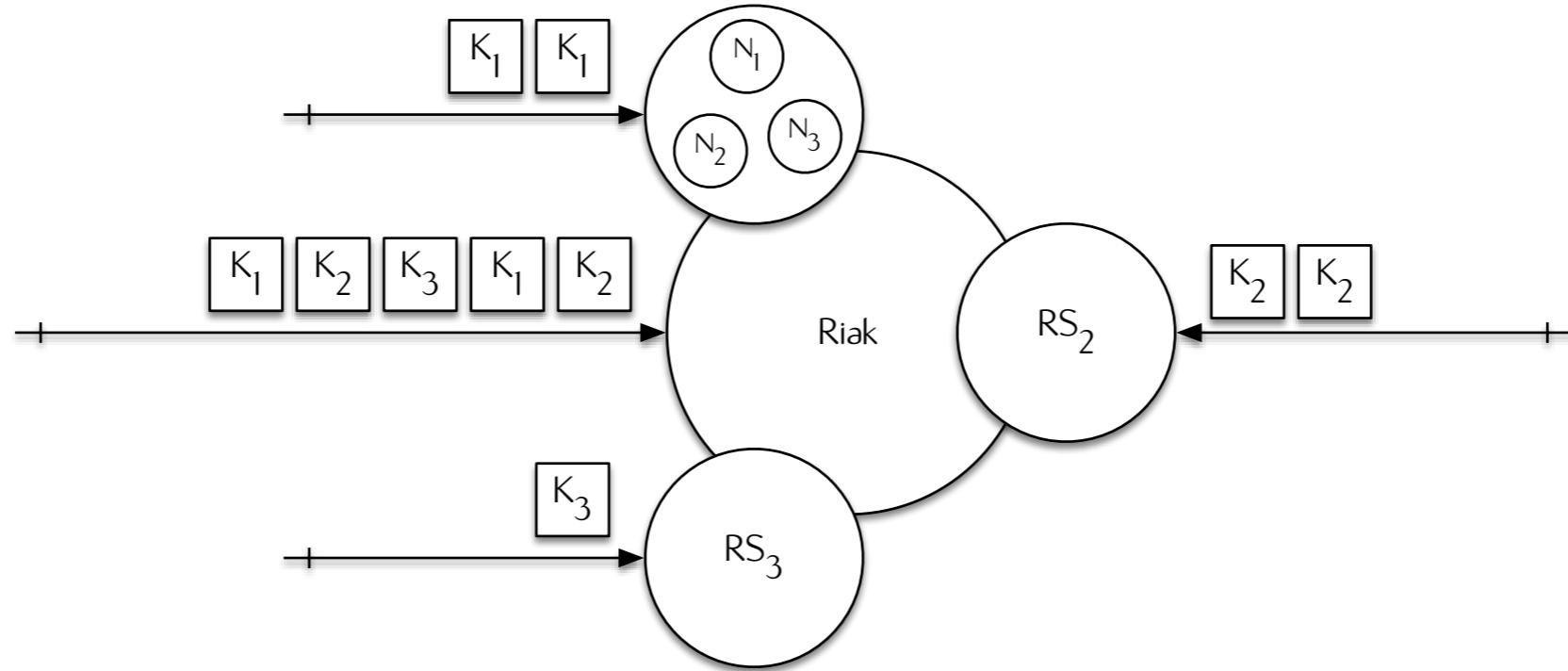


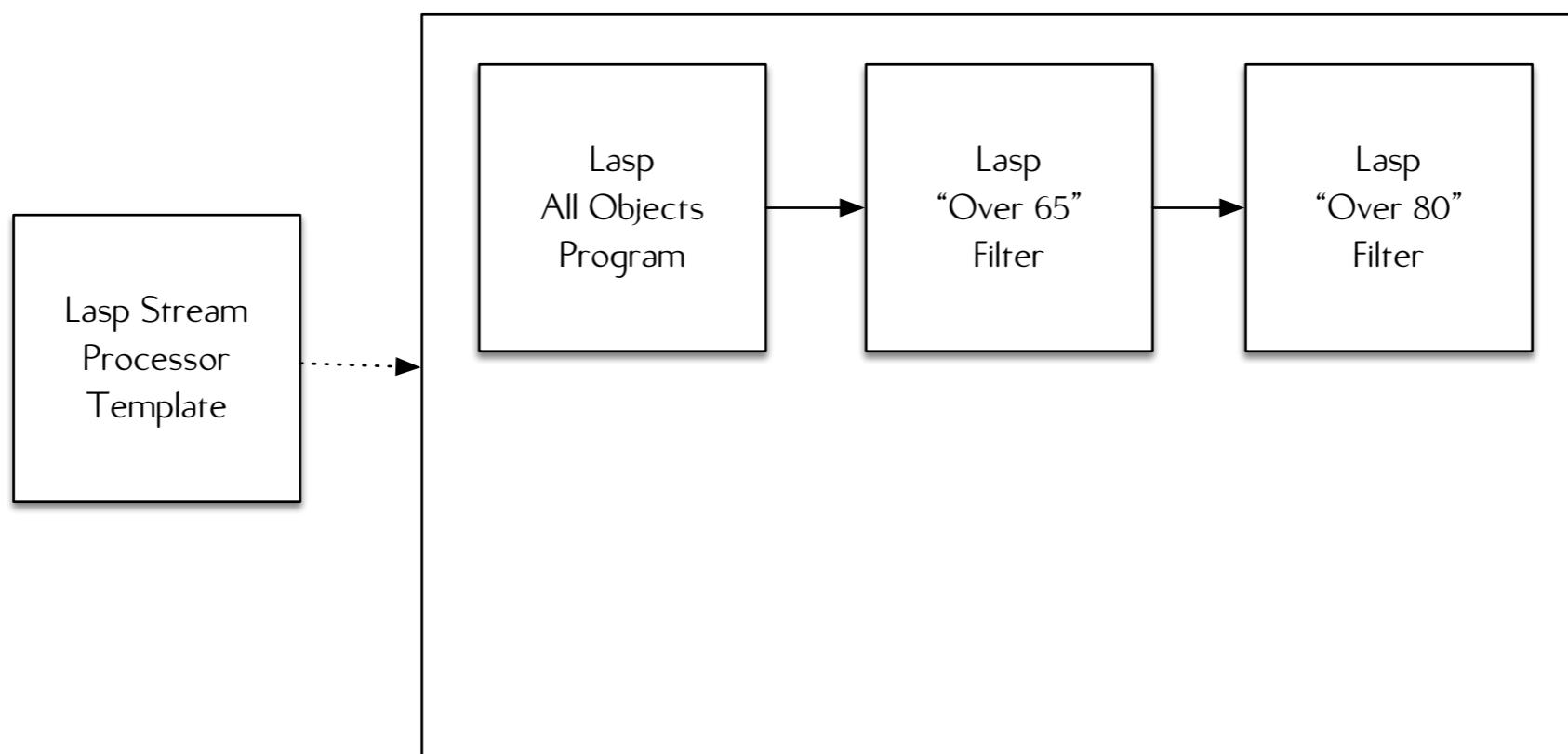
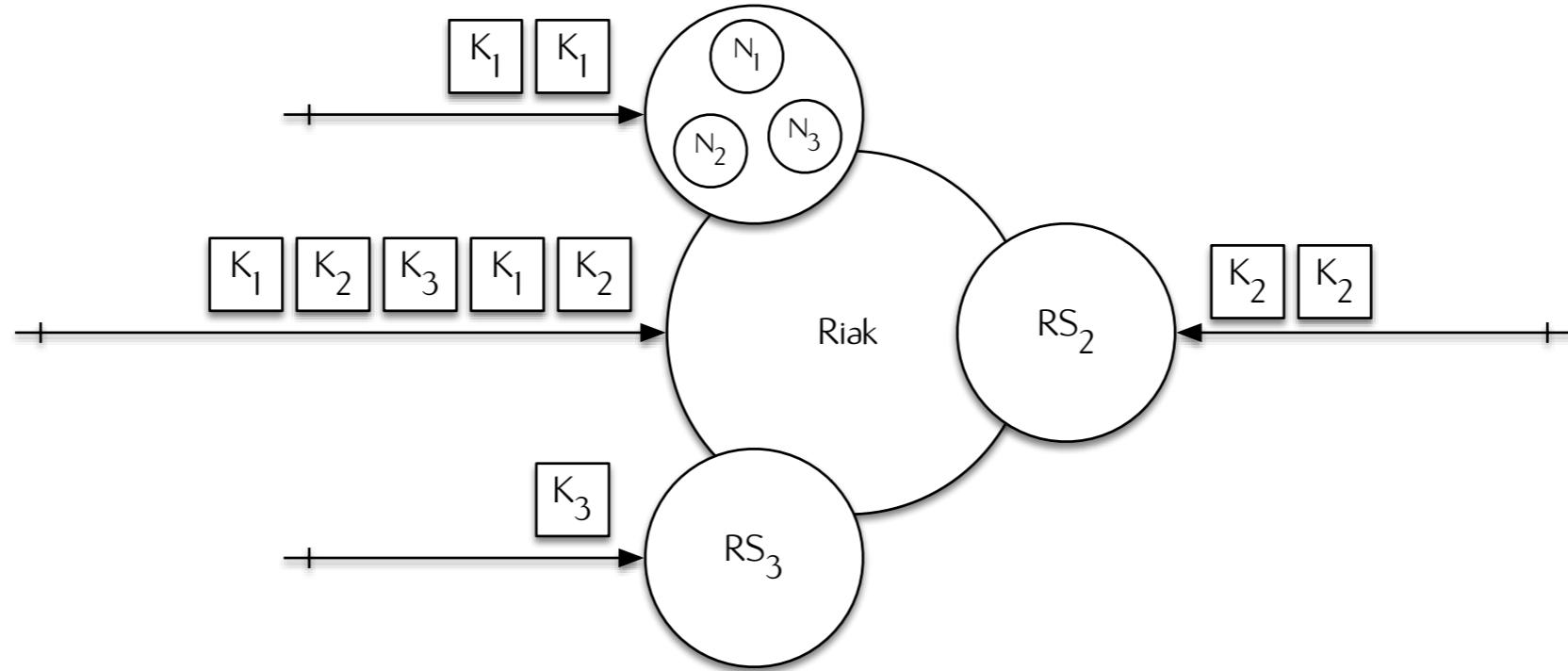


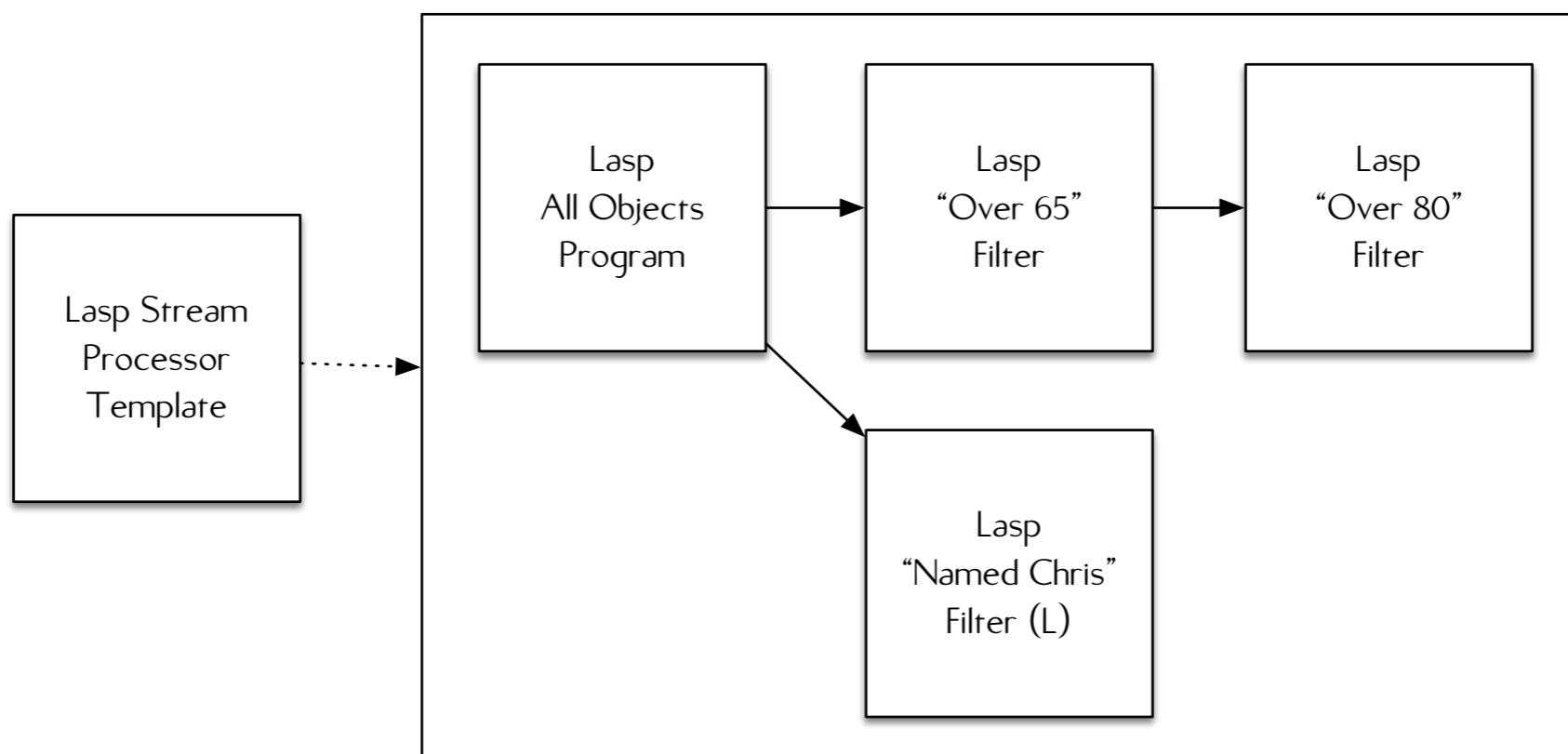
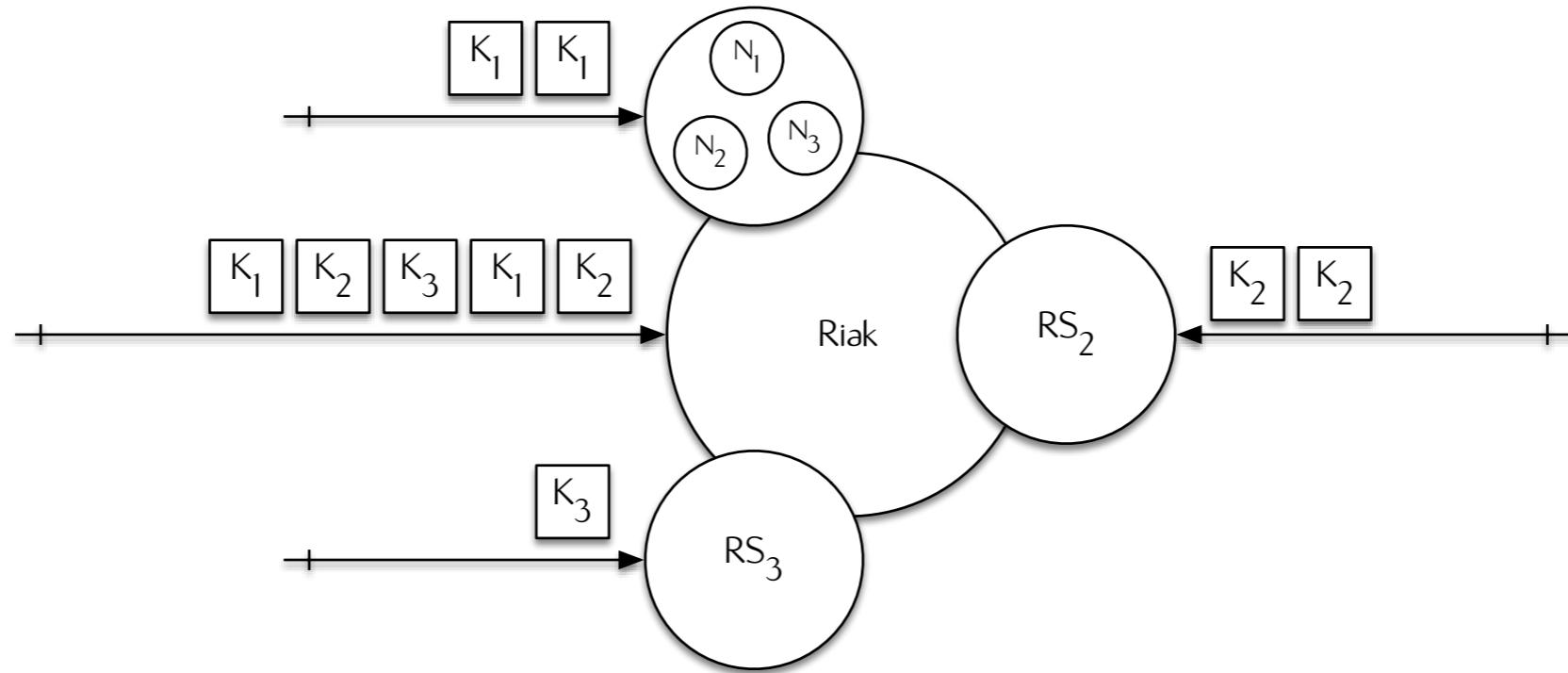
COMPOSE PROGRAMS



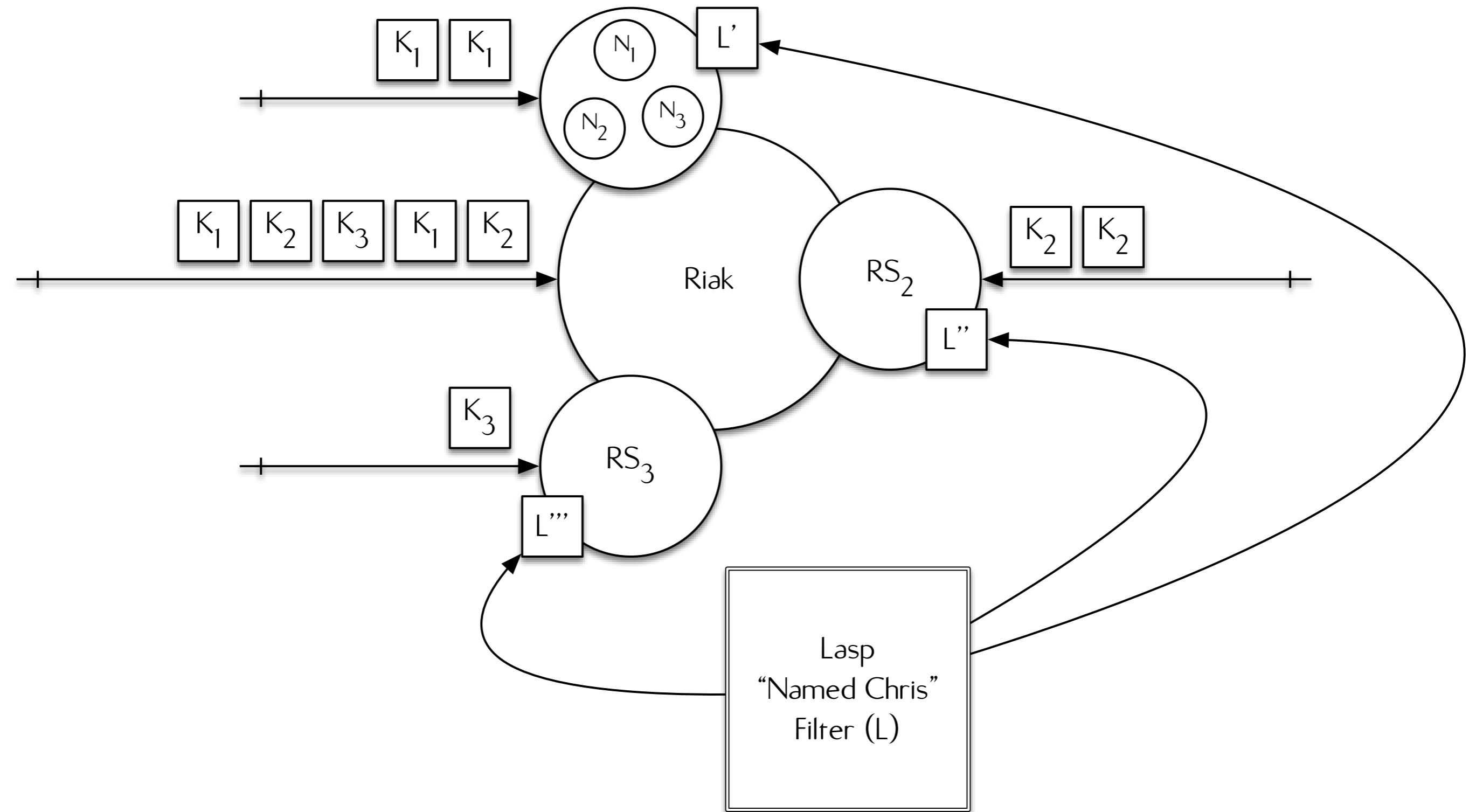


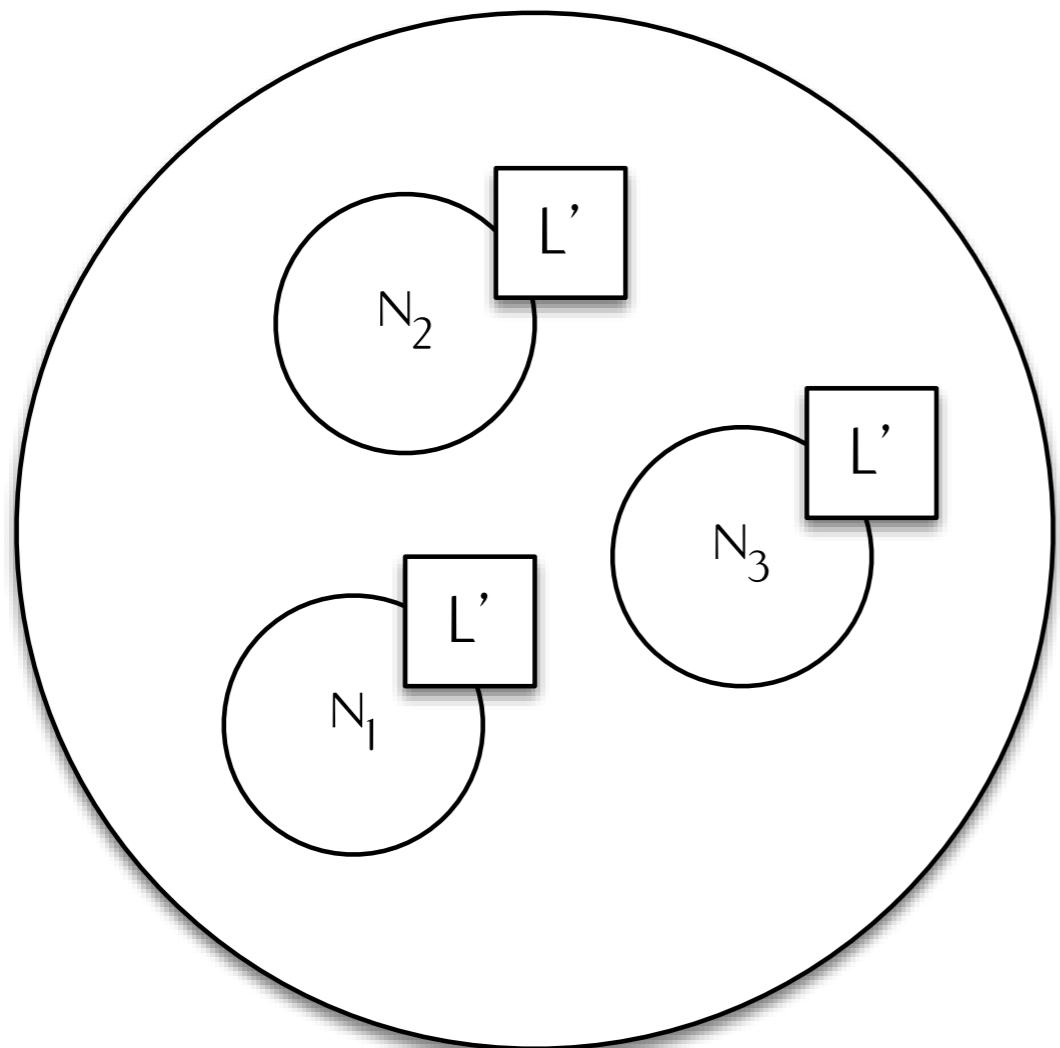




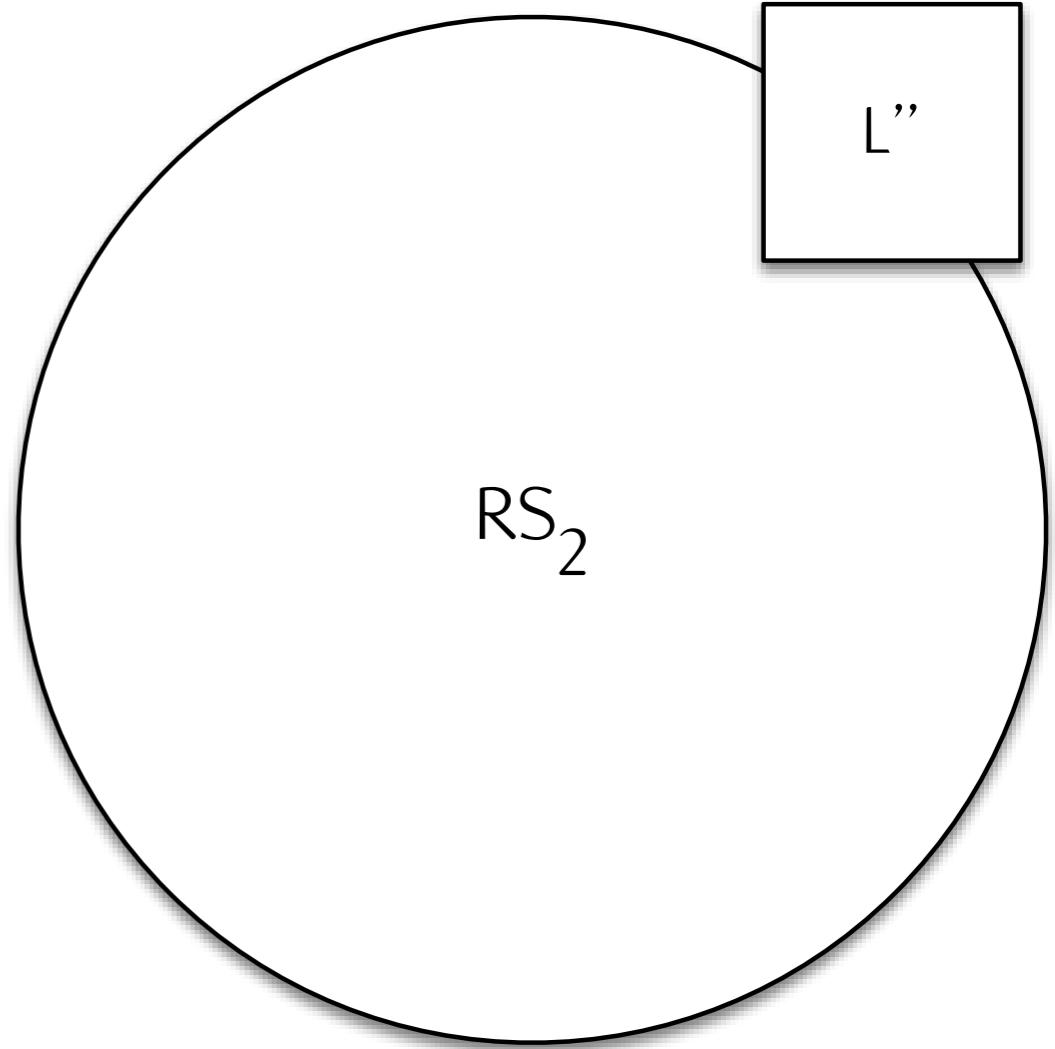
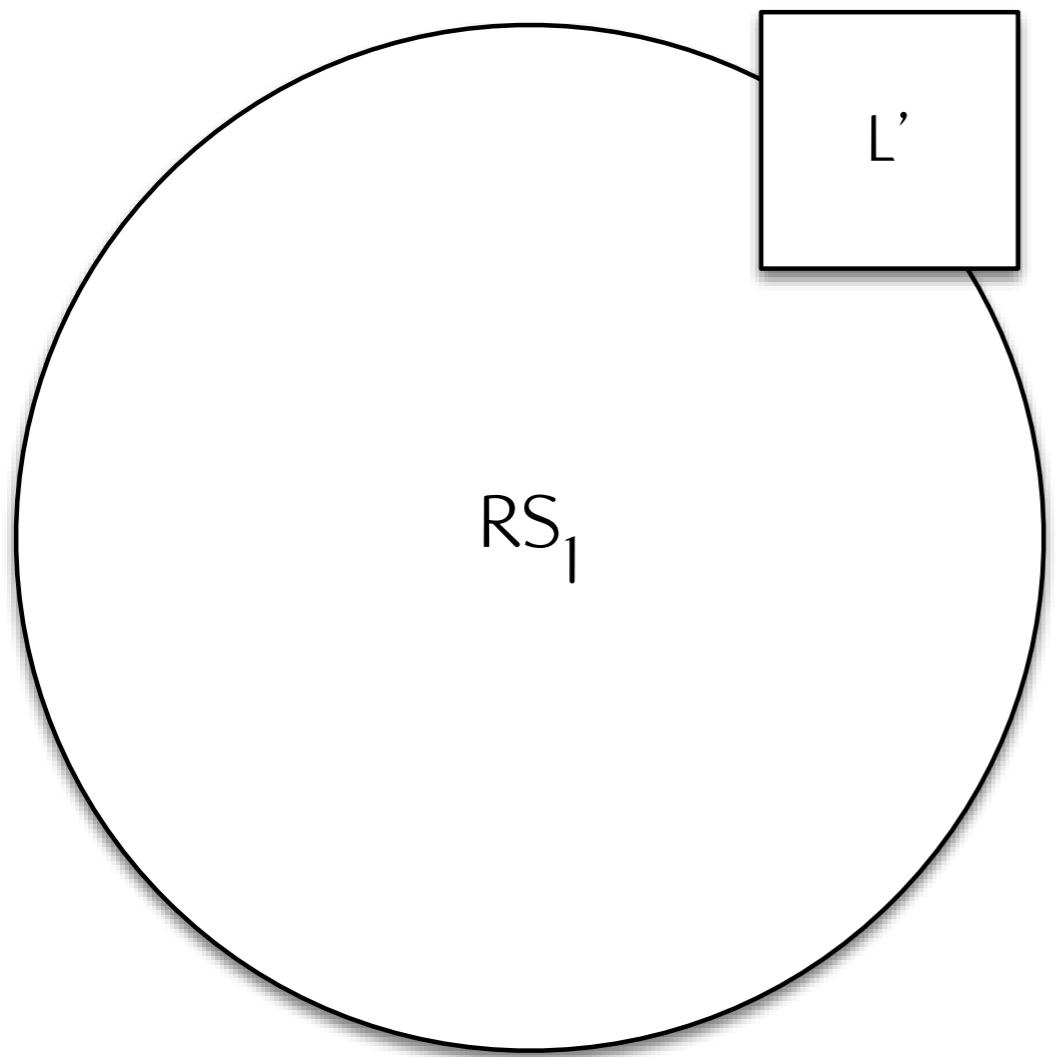


DISTRIBUTE APPLICATIONS

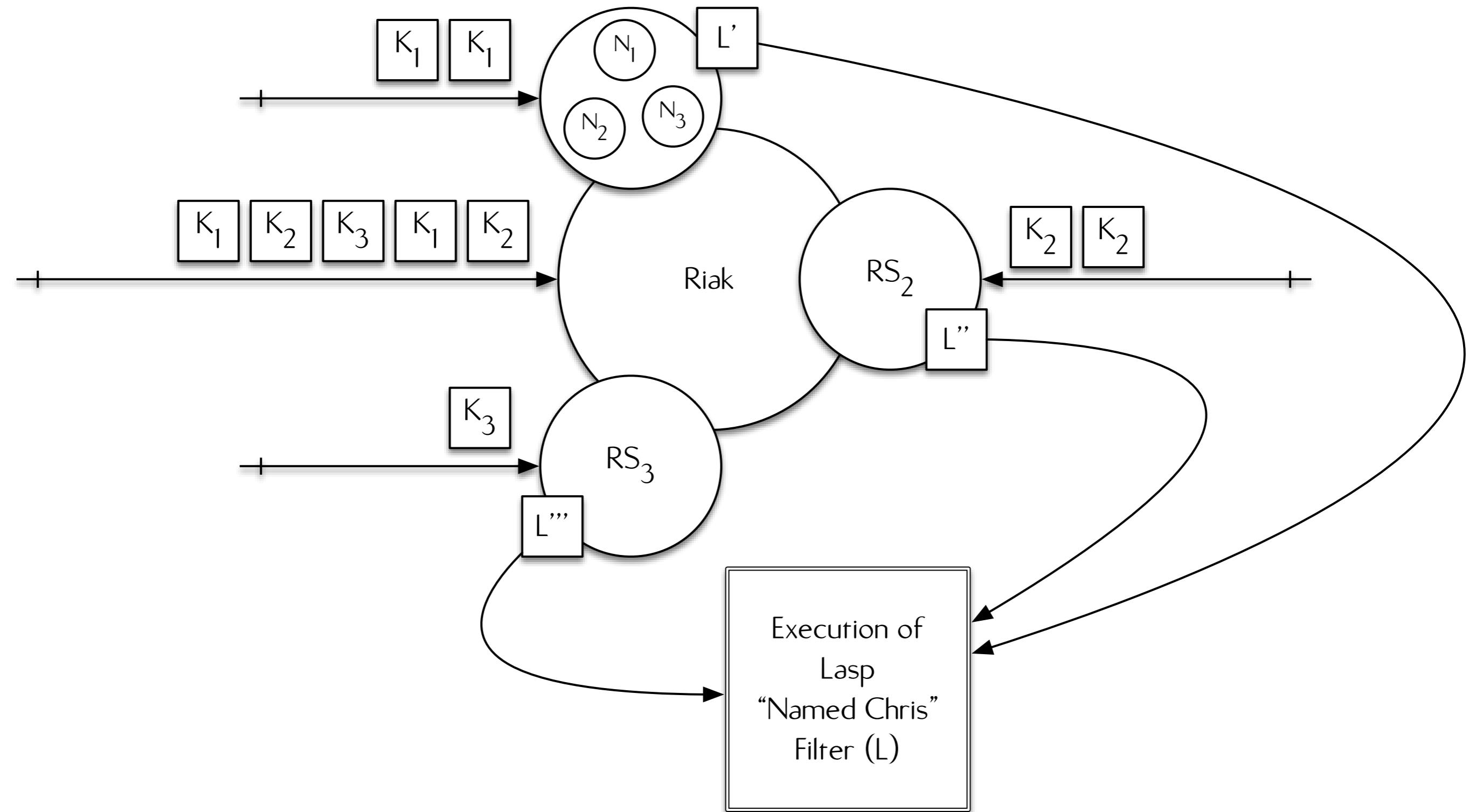




MERGE
RESULTS
(SEC)



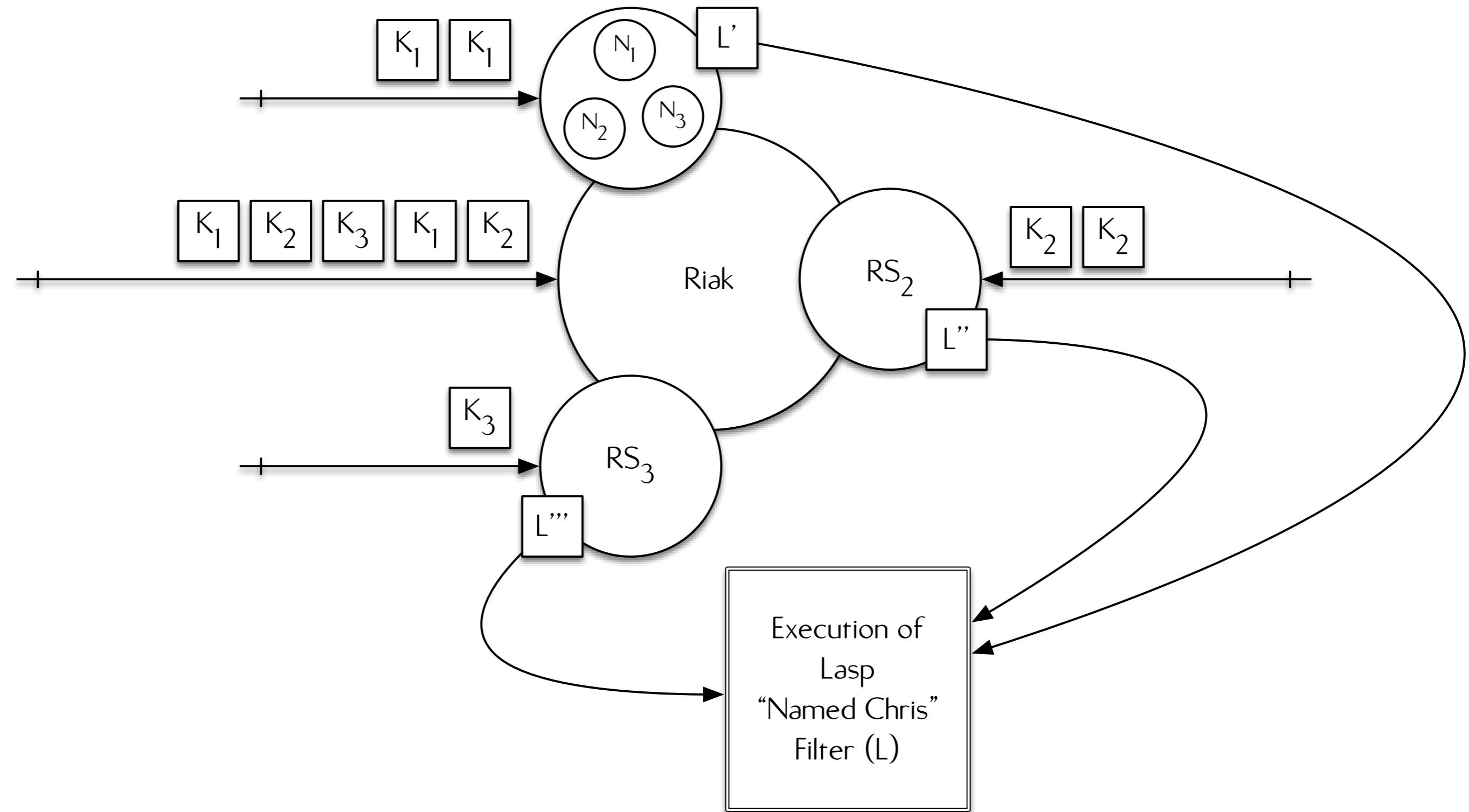
SUM RESULTS

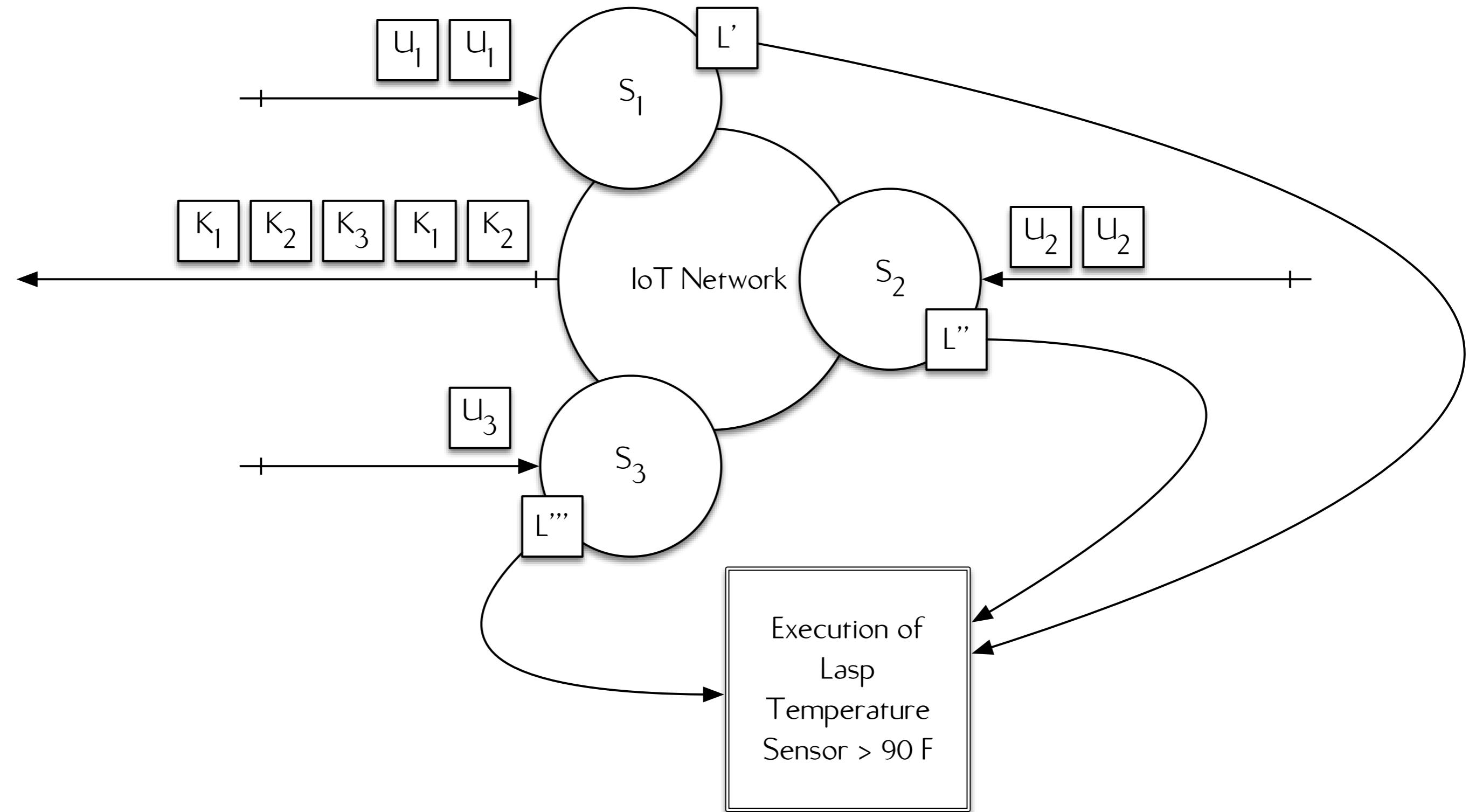


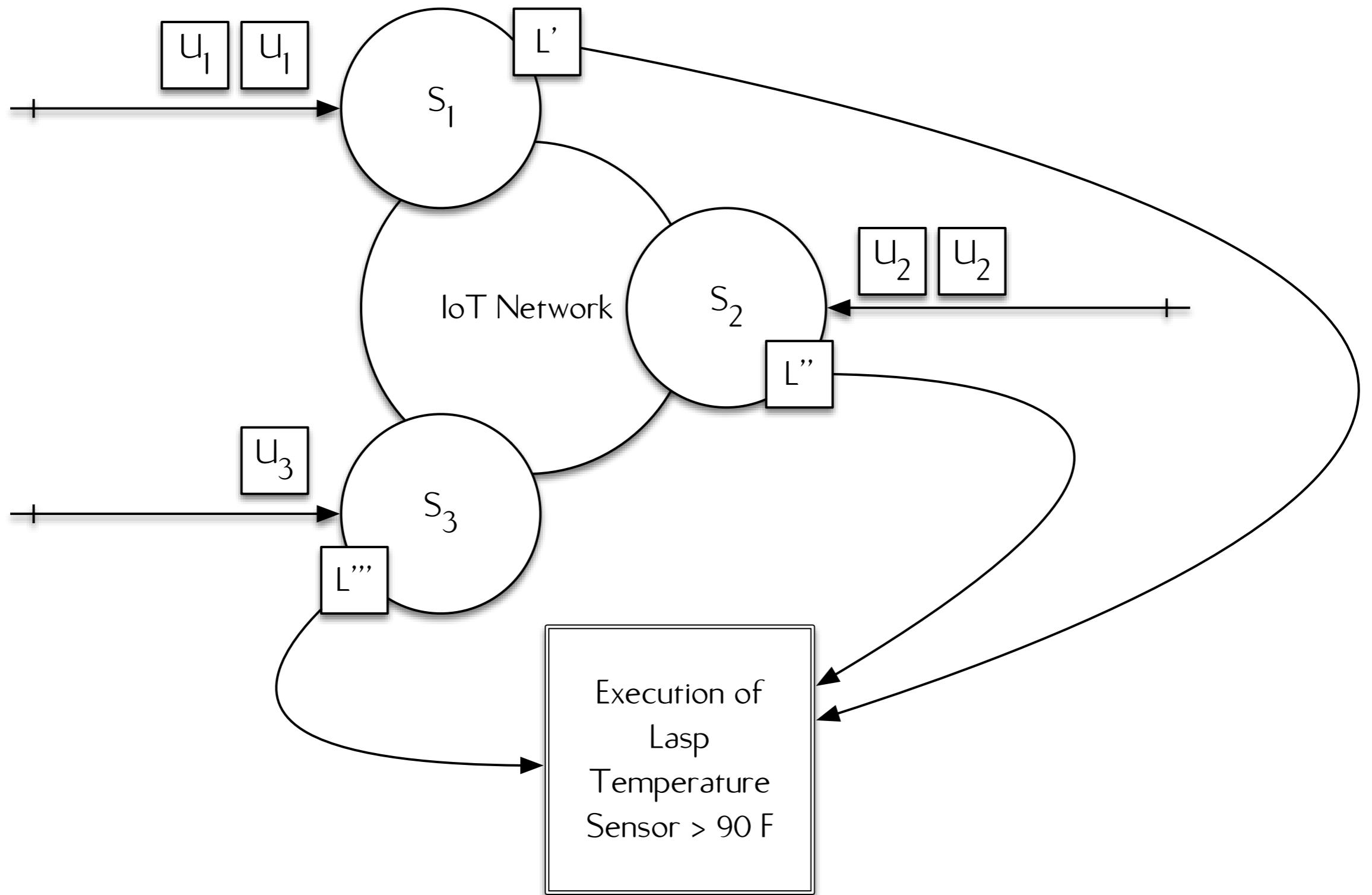
CACHE:

CACHE RESULTS OF MERGE
SPECULATIVELY EXECUTE BASED ON DIVERGENCE

“INTERNET OF THINGS”







IOT:

EXECUTE AT THE EDGE
WRITE CODE THINKING ABOUT “ALL” DATA

RELATED WORK

RELATED WORK

DISTRIBUTED OZ

RELATED WORK

DERFLOW_L

RELATED WORK

BLOOM^L

RELATED WORK

LVARS

RELATED WORK

D-STREAMS

RELATED WORK
SUMMINGBIRD

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 / EUROSYS 2015

LASP

A LANGUAGE FOR DISTRIBUTED, EVENTUALLY
CONSISTENT COMPUTATIONS WITH CRDTs



RICON 2015
NOVEMBER 4-6, 2015
SAN FRANCISCO, CA



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.

Questions?

*Please remember to evaluate via the GOTO
Guide App*