

BEEHIVE

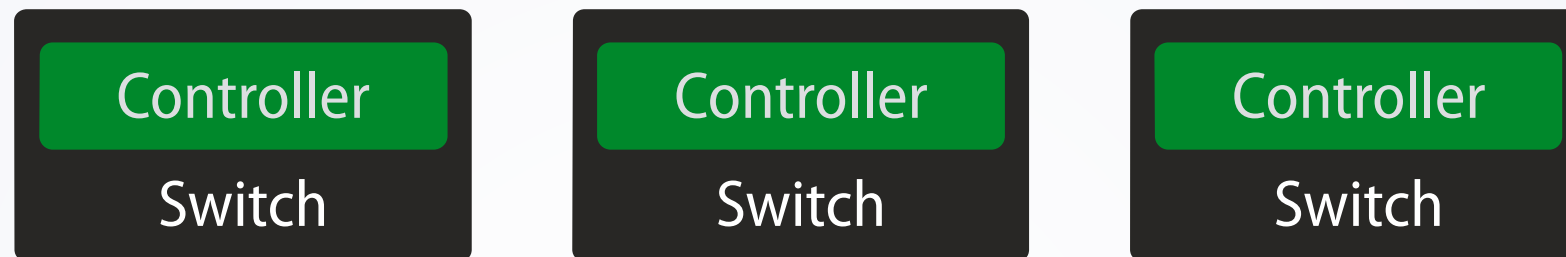
Towards a simple abstraction for scalable
software-defined networking

SOHEIL HASSAS YEGANEH YASHAR GANJALI

University of Toronto

TRADITIONAL NETWORKS

Hard to Program Distributed Systems



SOFTWARE DEFINED NETWORKING

Easy

~~Hard to Program Distributed Systems~~

Centralized

Application

Controller

Switch

Switch

Switch

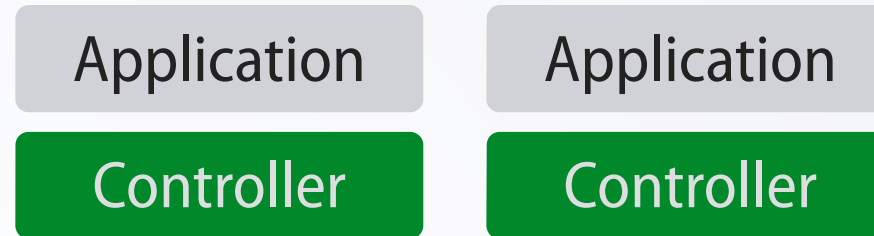
SOFTWARE DEFINED NETWORKING

Easy

~~Hard~~ to Program Distributed Systems

Existing Distributed Controllers

- Excellent in performance & scalability
- Perfect fit for some specific scenarios



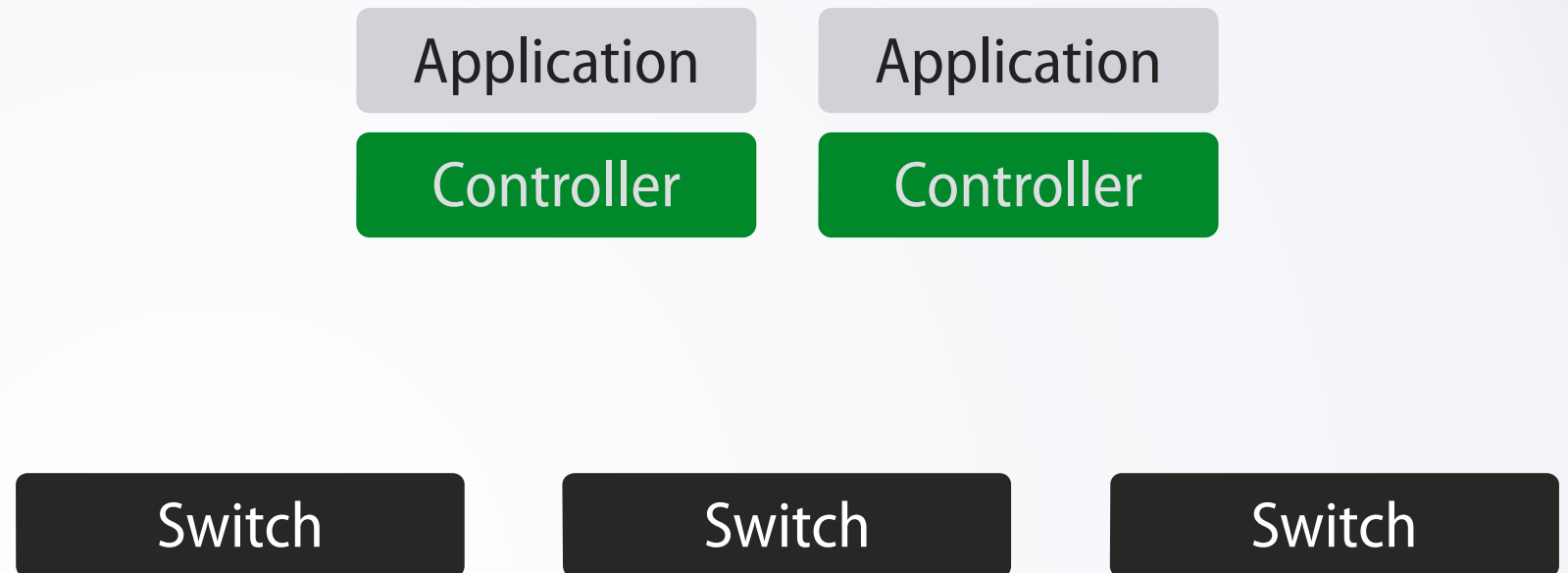
SOFTWARE DEFINED NETWORKING

Much better than traditional networks

still Hard to Program Distributed Systems

Existing Distributed Controllers

- Don't hide the boilerplates of distributed programming
- Require significant efforts to instrument and optimize apps



OUR GOAL

Easy

~~Hard to Program Distributed Systems~~

*similar to
centralized
controllers*

Application

Application

Controller

Controller

*+ optimized
placements*

*+ application
analytics*

Switch

Switch

Switch

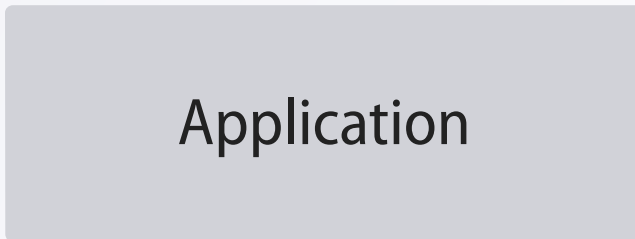
OUR GOAL

centralized

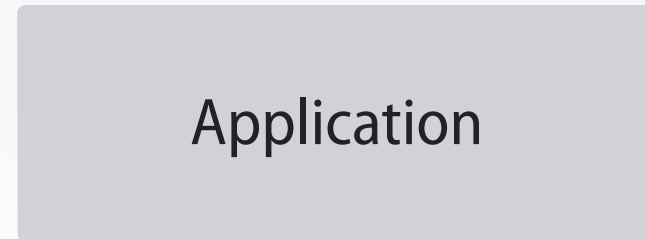
Application

OUR GOAL

centralized

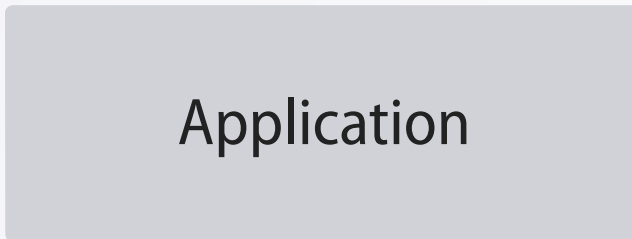


can be automatically
transformed into



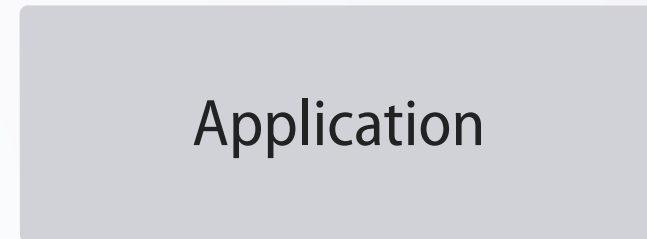
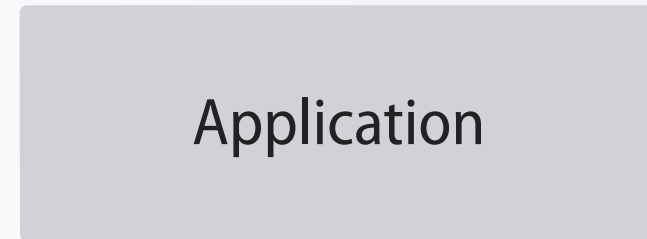
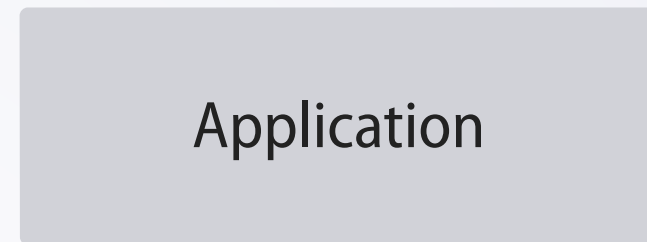
OUR GOAL

centralized



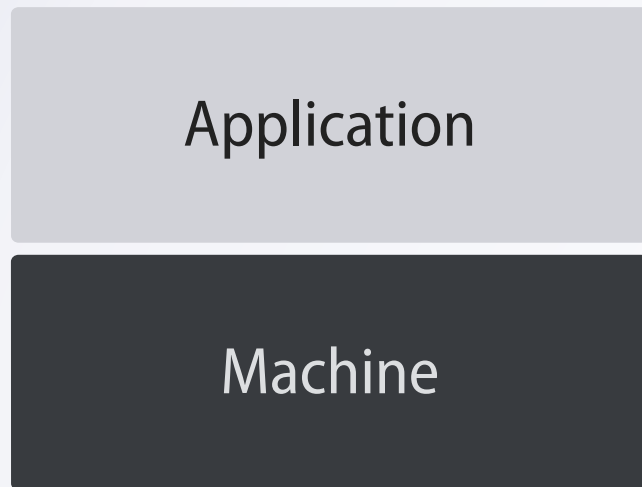
can be automatically transformed into

distributed



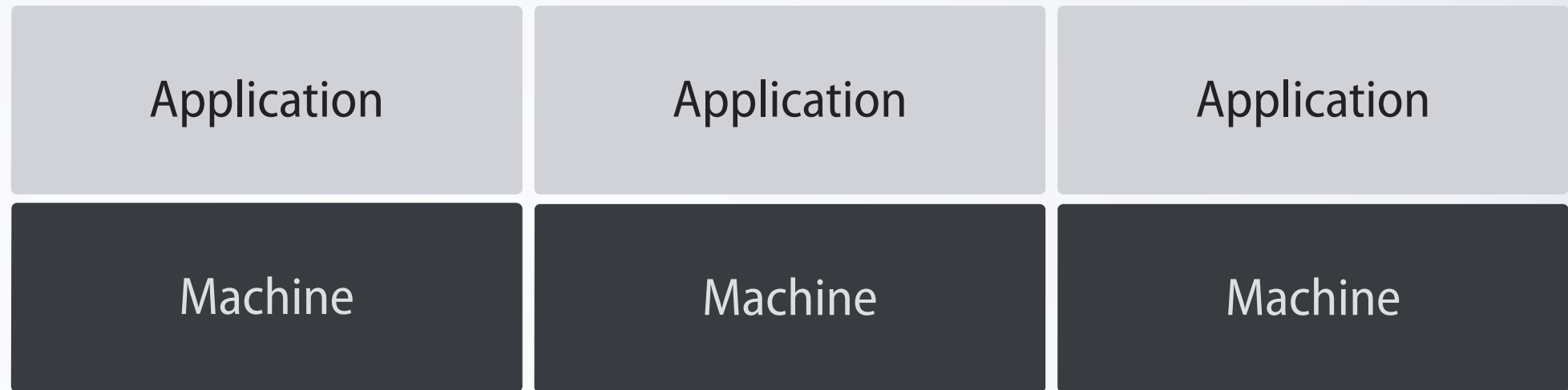
OUR GOAL

centralized



=

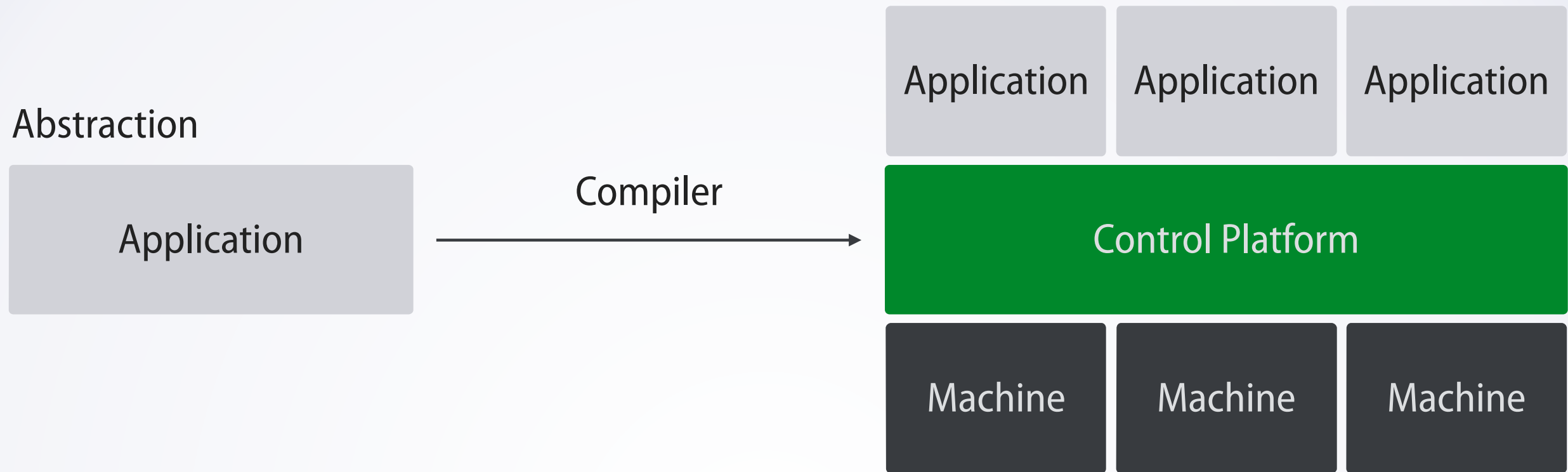
distributed



deployed on multiple physical machines.

Very challenging for generic control applications.

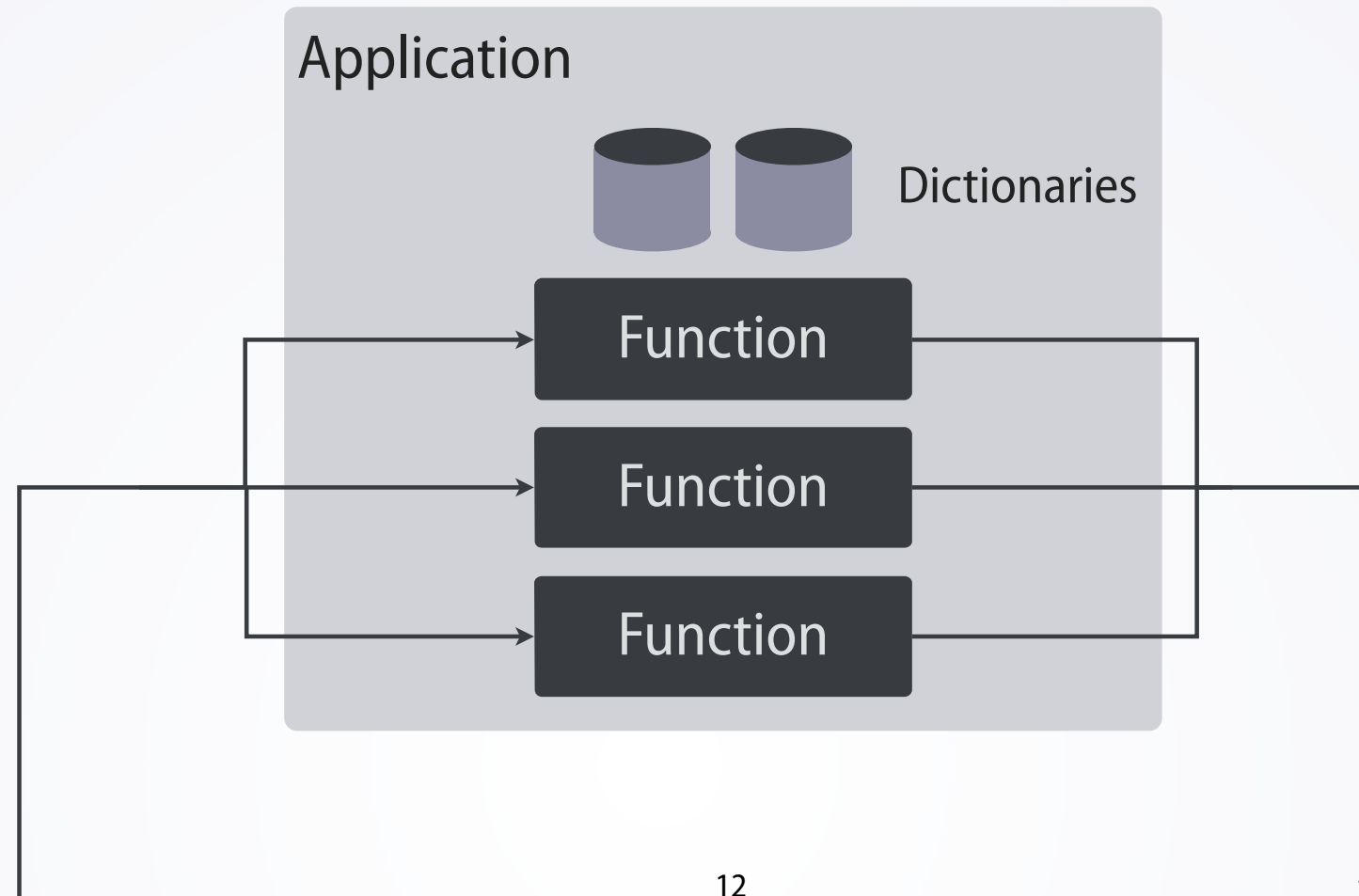
OVERVIEW



ABSTRACTION

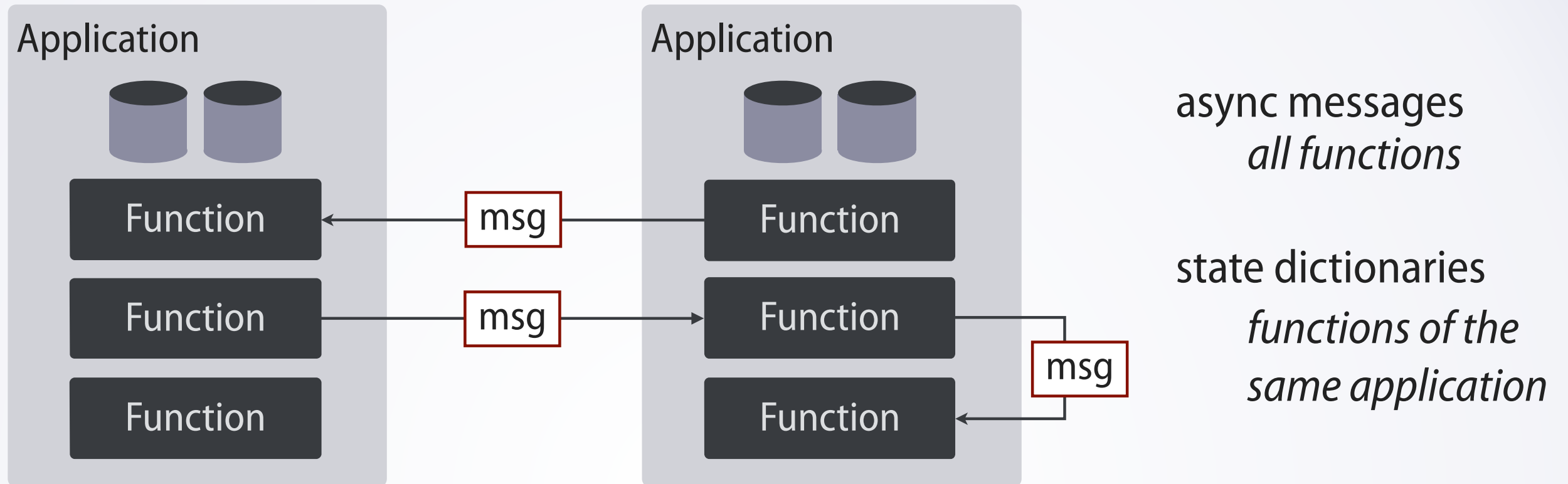
what is a control application?

Process async messages in application functions using state dictionaries

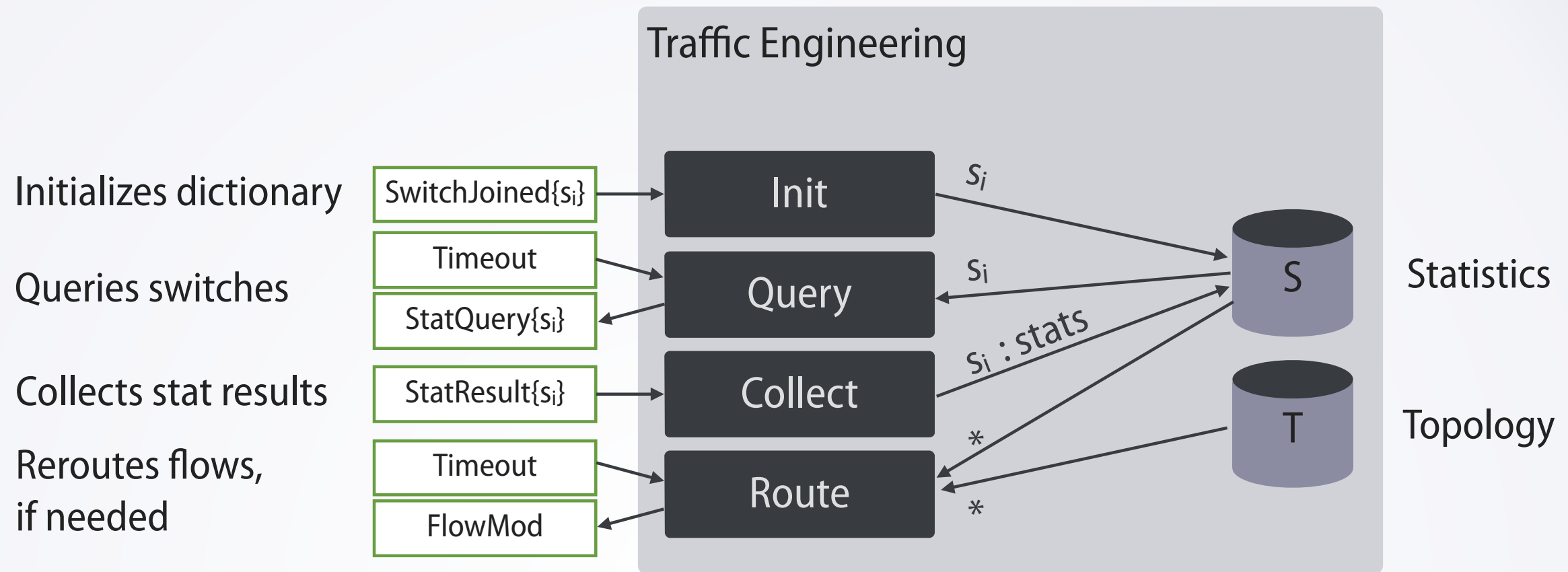


ABSTRACTION

how do applications communicate?

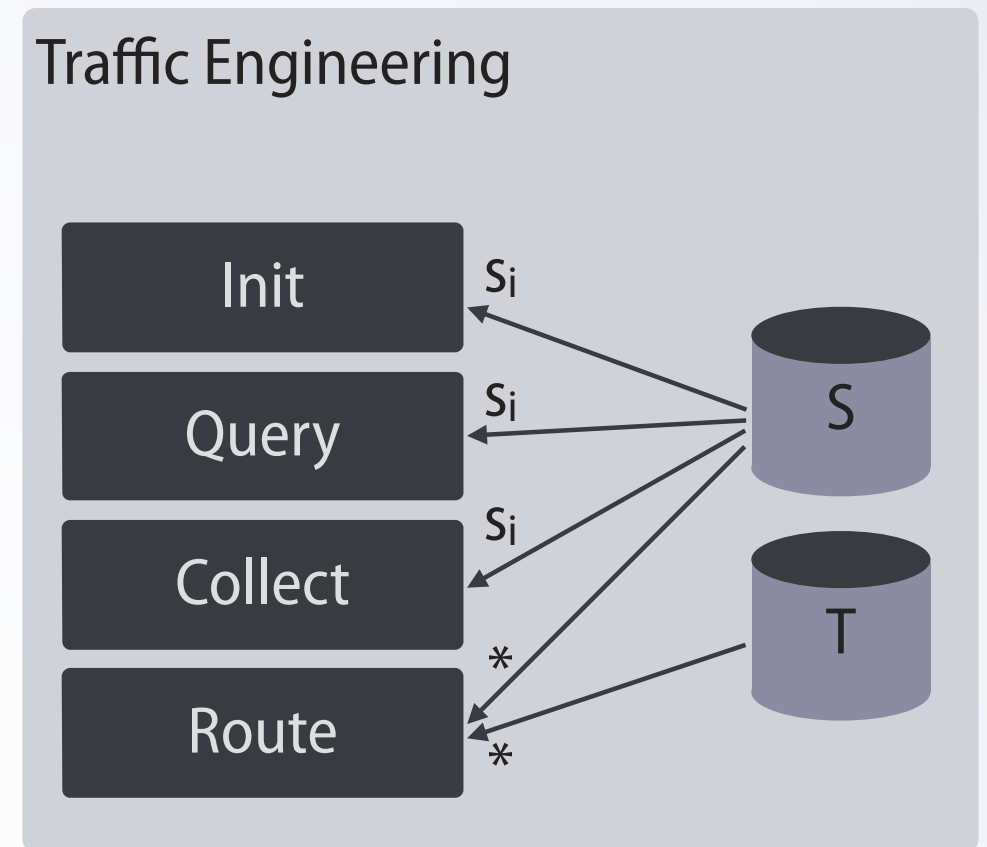


EXAMPLE



EXAMPLE

How to transform TE into a distributed application while preserving state consistency?



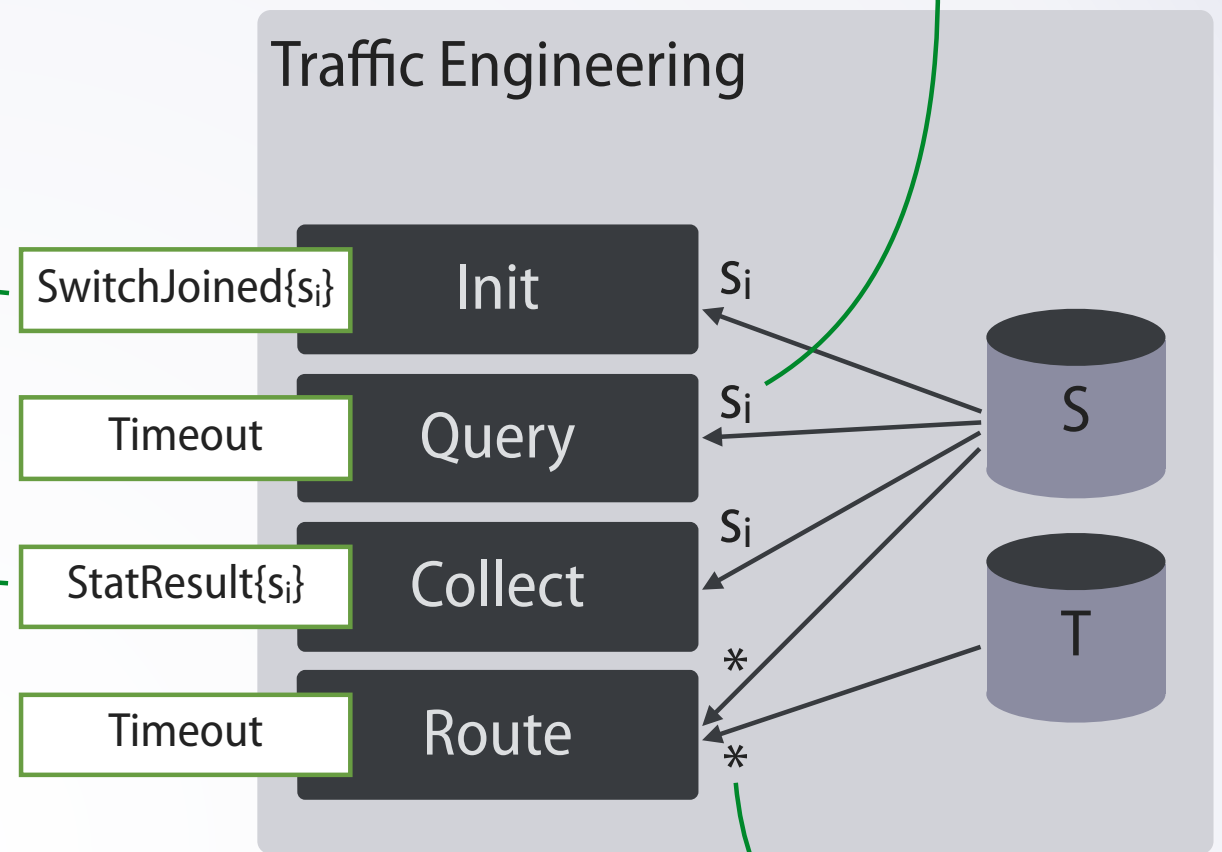
EXAMPLE

Functions create an implicit mapping between messages and dictionary entries:

The entries a function needs to process a message.

The dictionary key is in the message

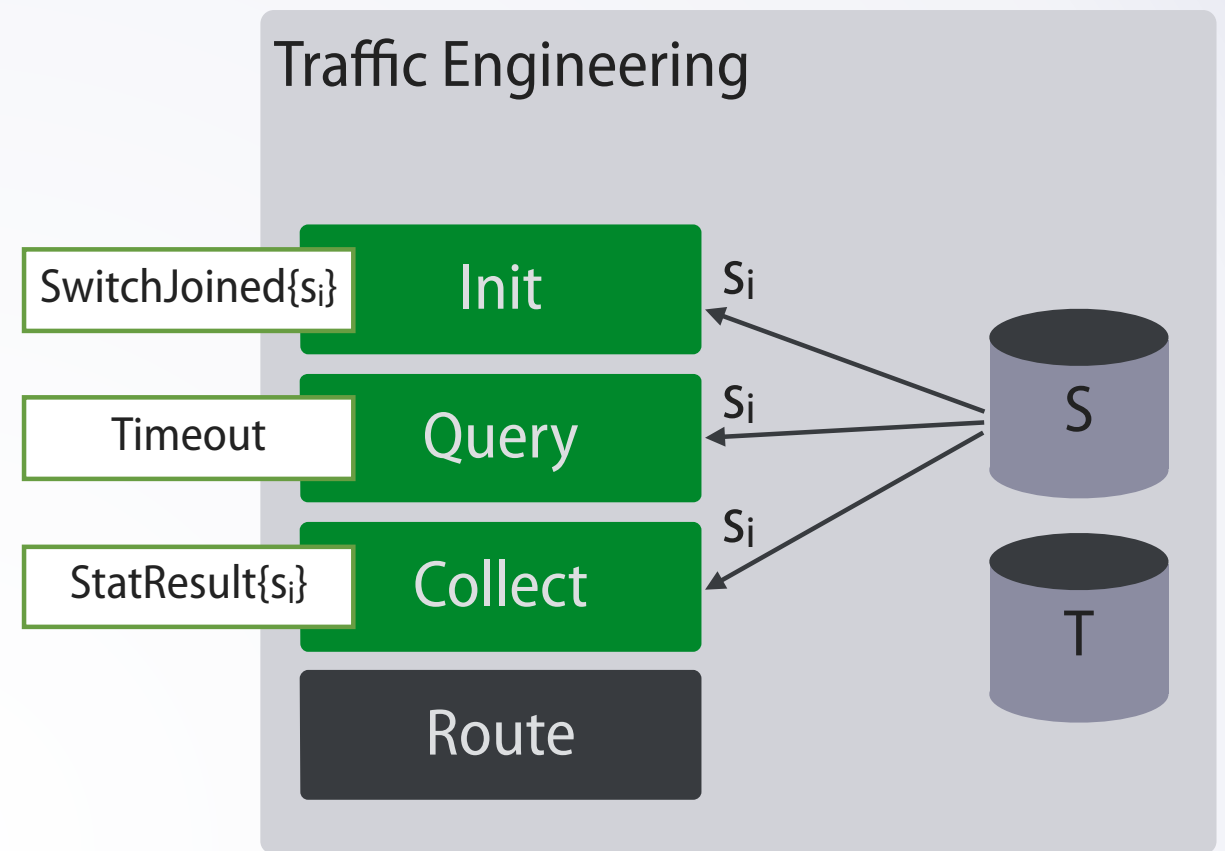
For each entry



All entries

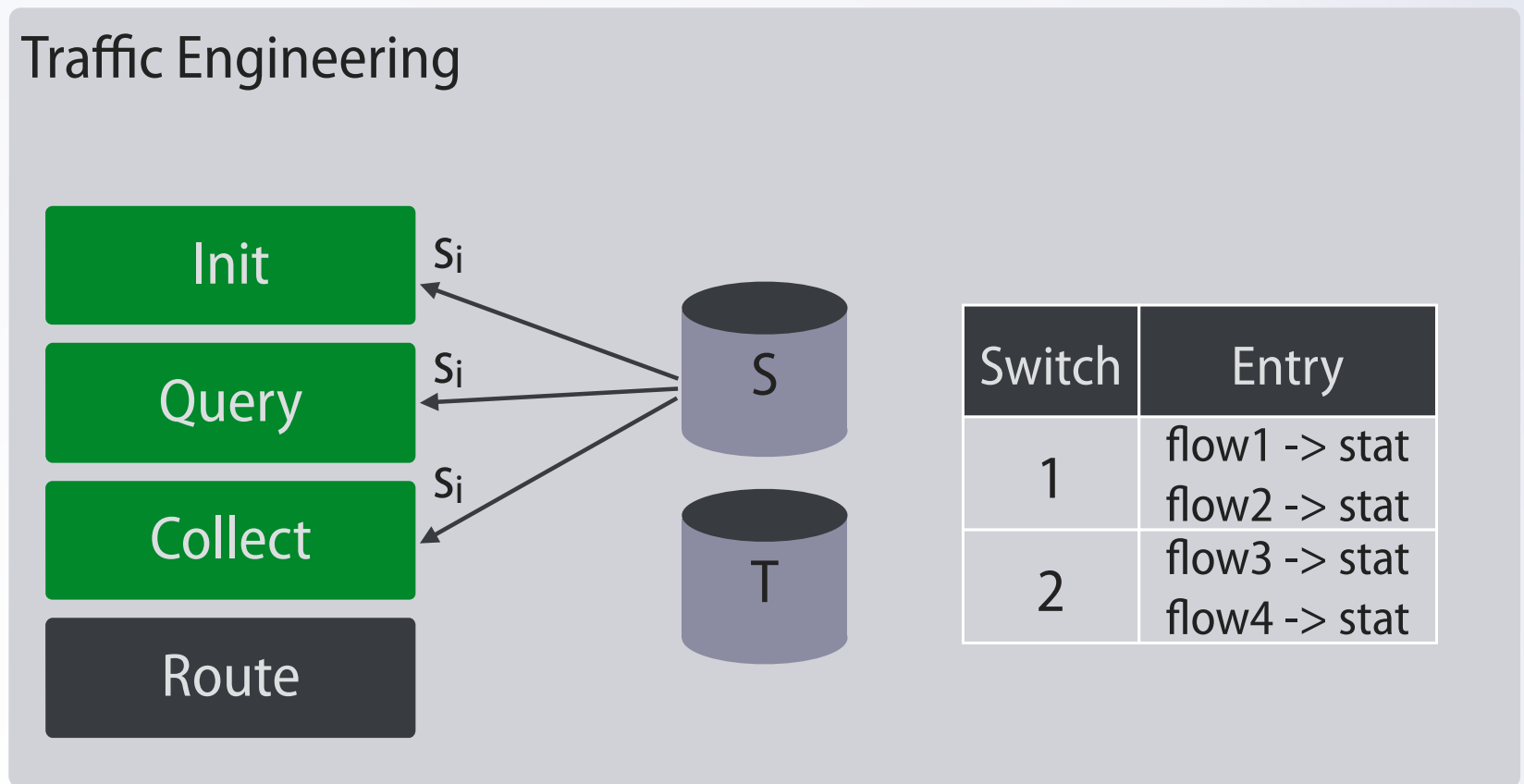
EXAMPLE

Init(), **Query()** and **Collect()**
access **S** on a per switch basis.

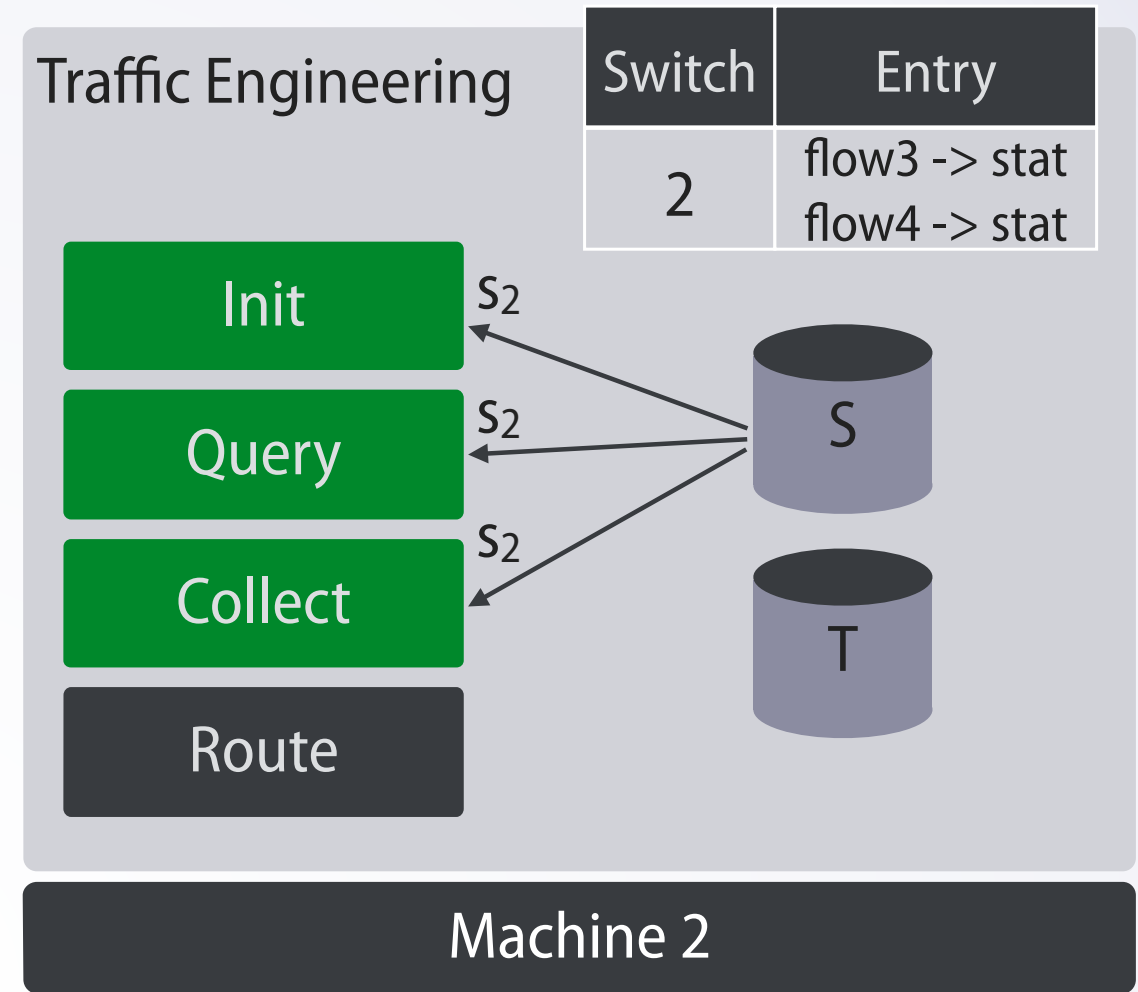
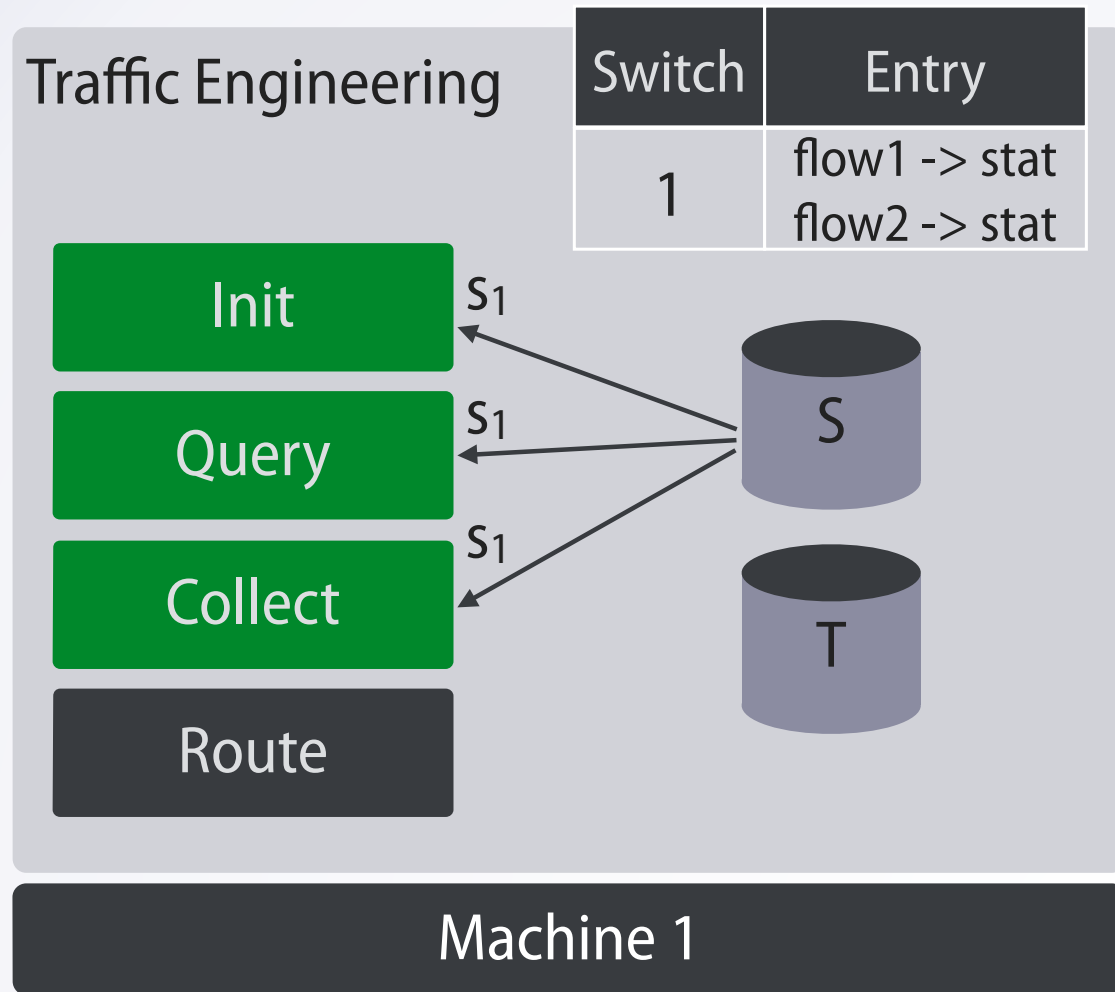


EXAMPLE

Init(), **Query()** and **Collect()** access **S** on a per switch basis.



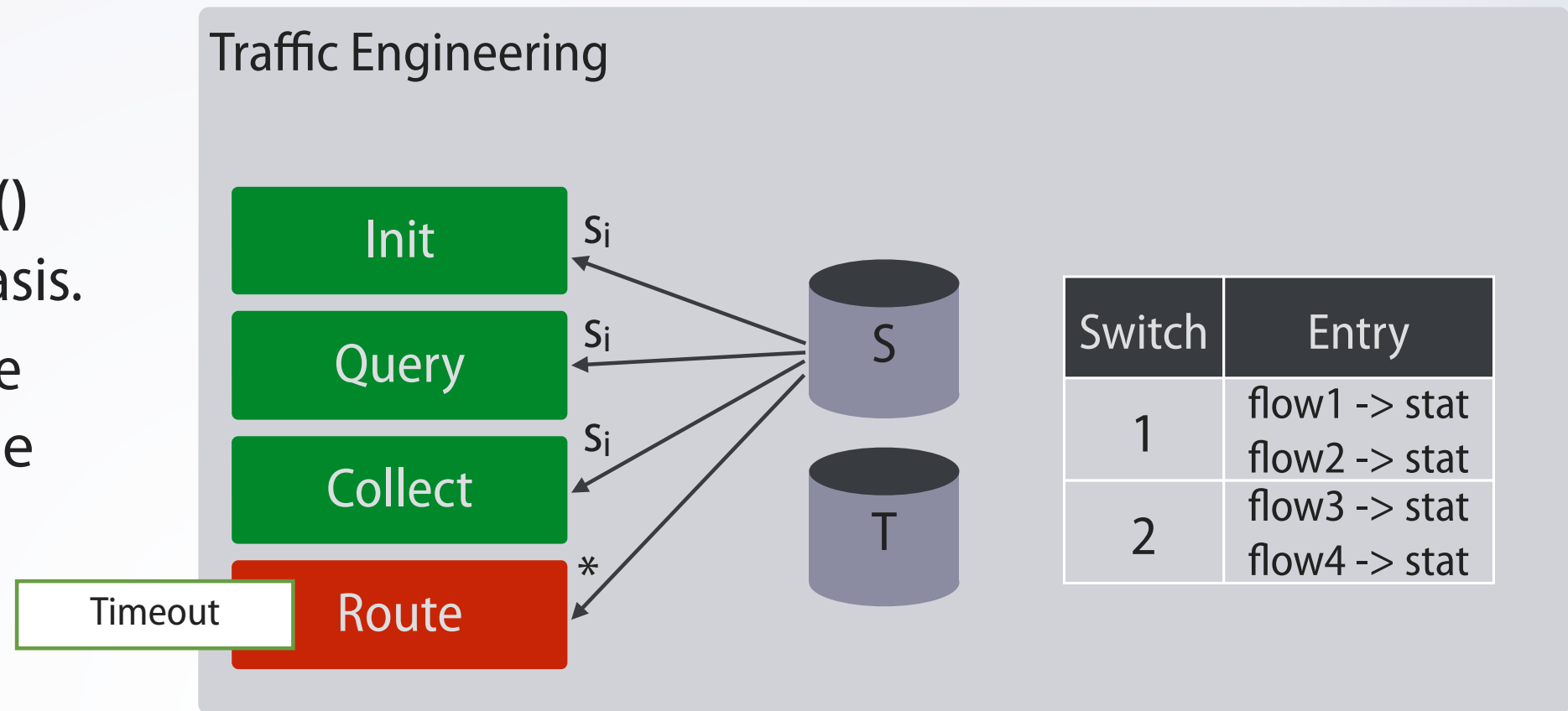
EXAMPLE



EXAMPLE

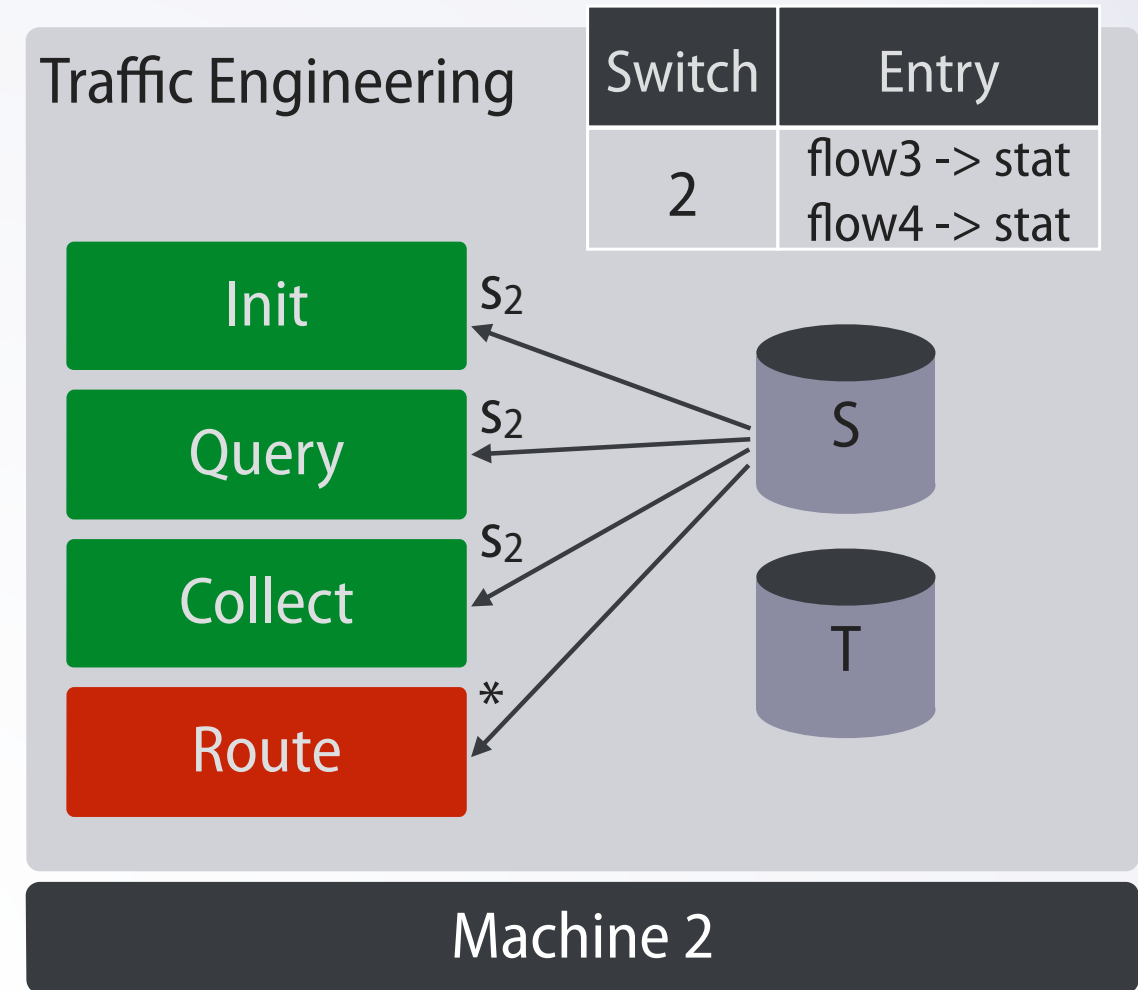
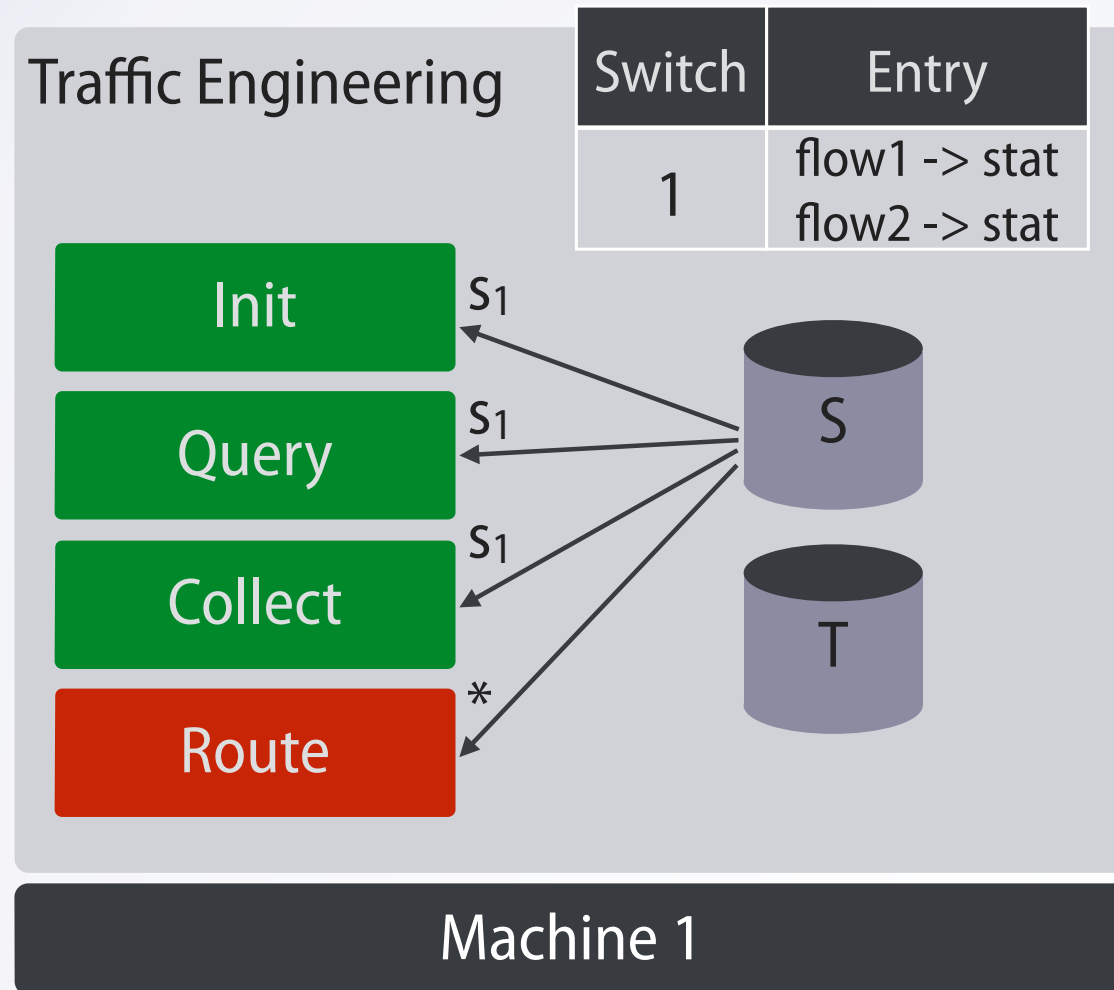
Init(), **Query()** and **Collect()** access **S** on a per switch basis.

Route() accesses the whole dictionary **S** to process the timeout message.

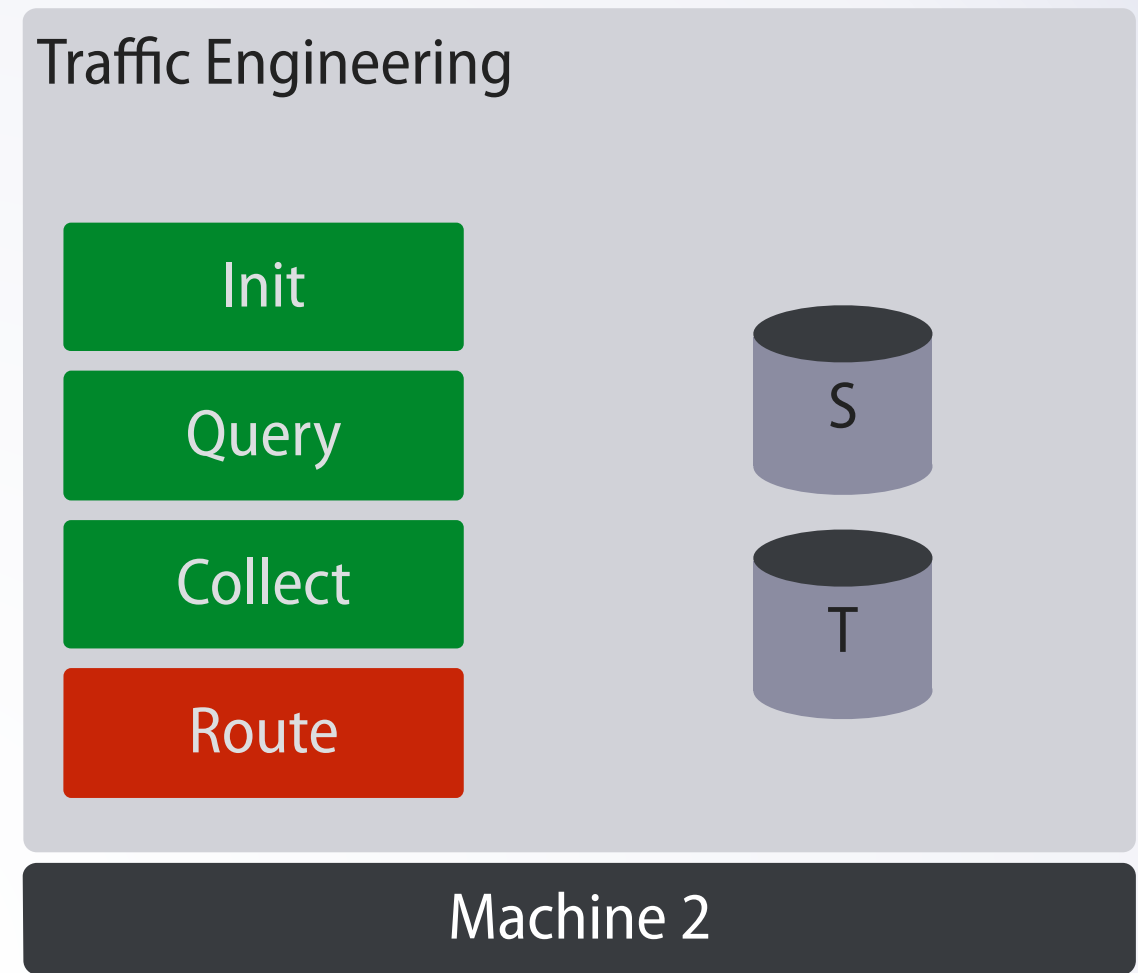
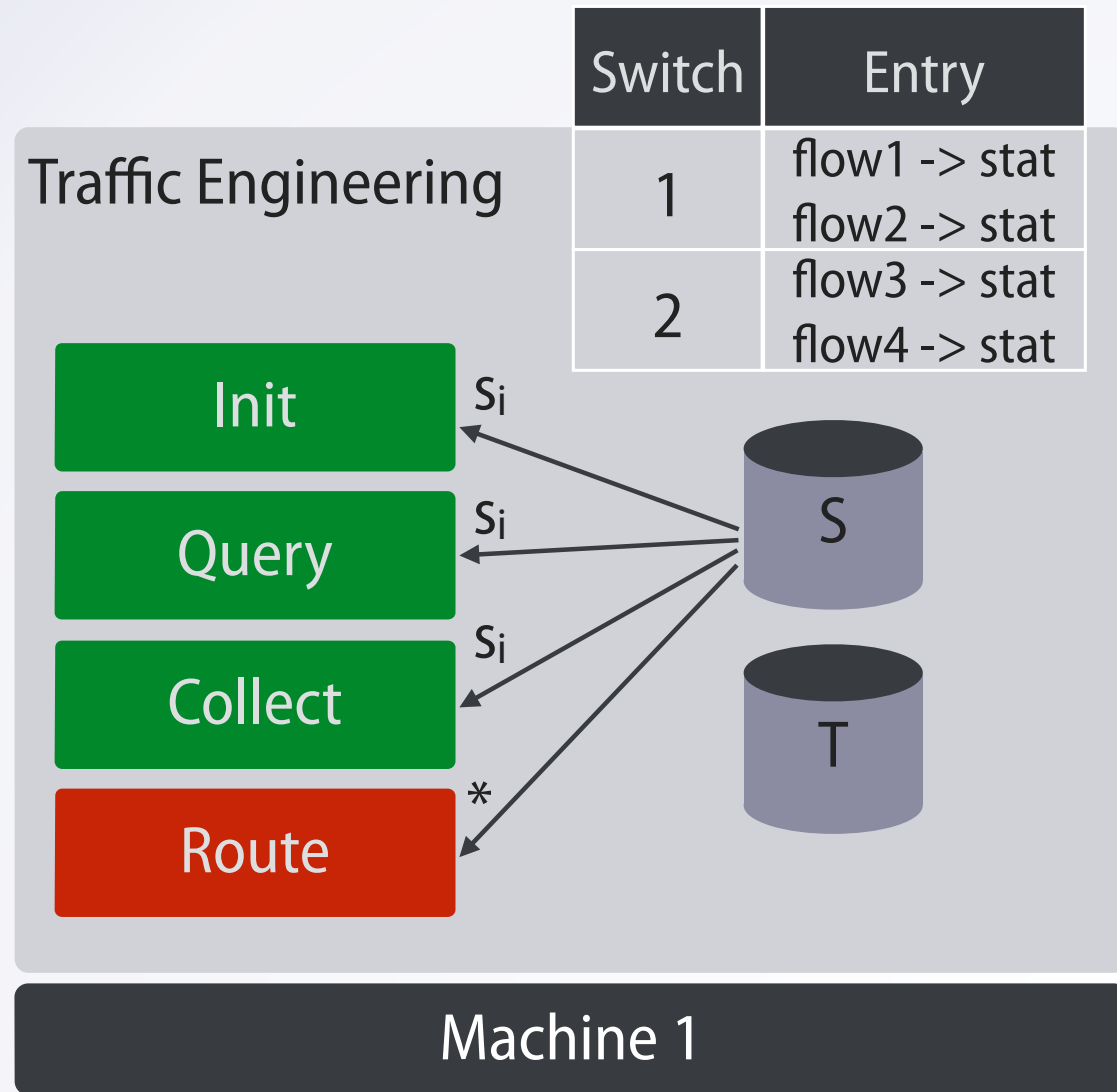


EXAMPLE

This will cause in consistency.

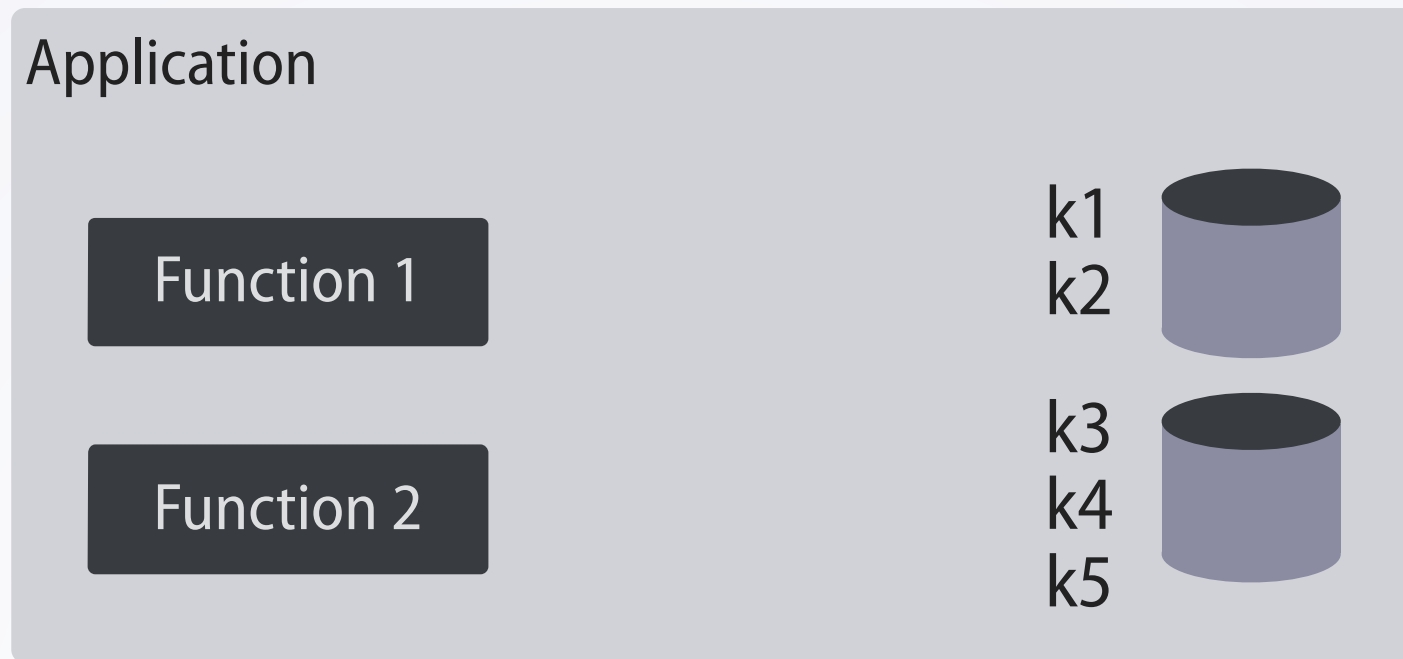


EXAMPLE

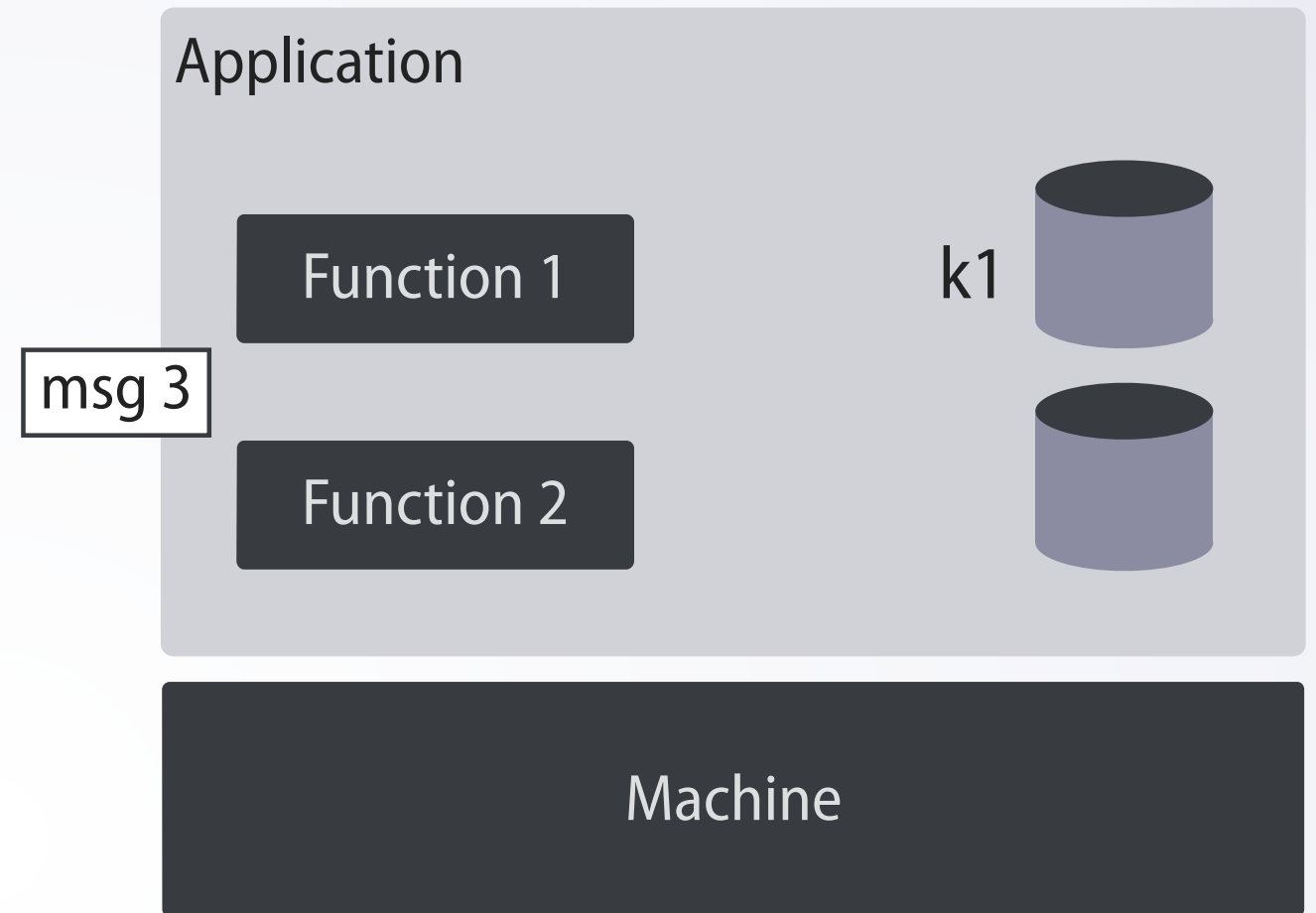
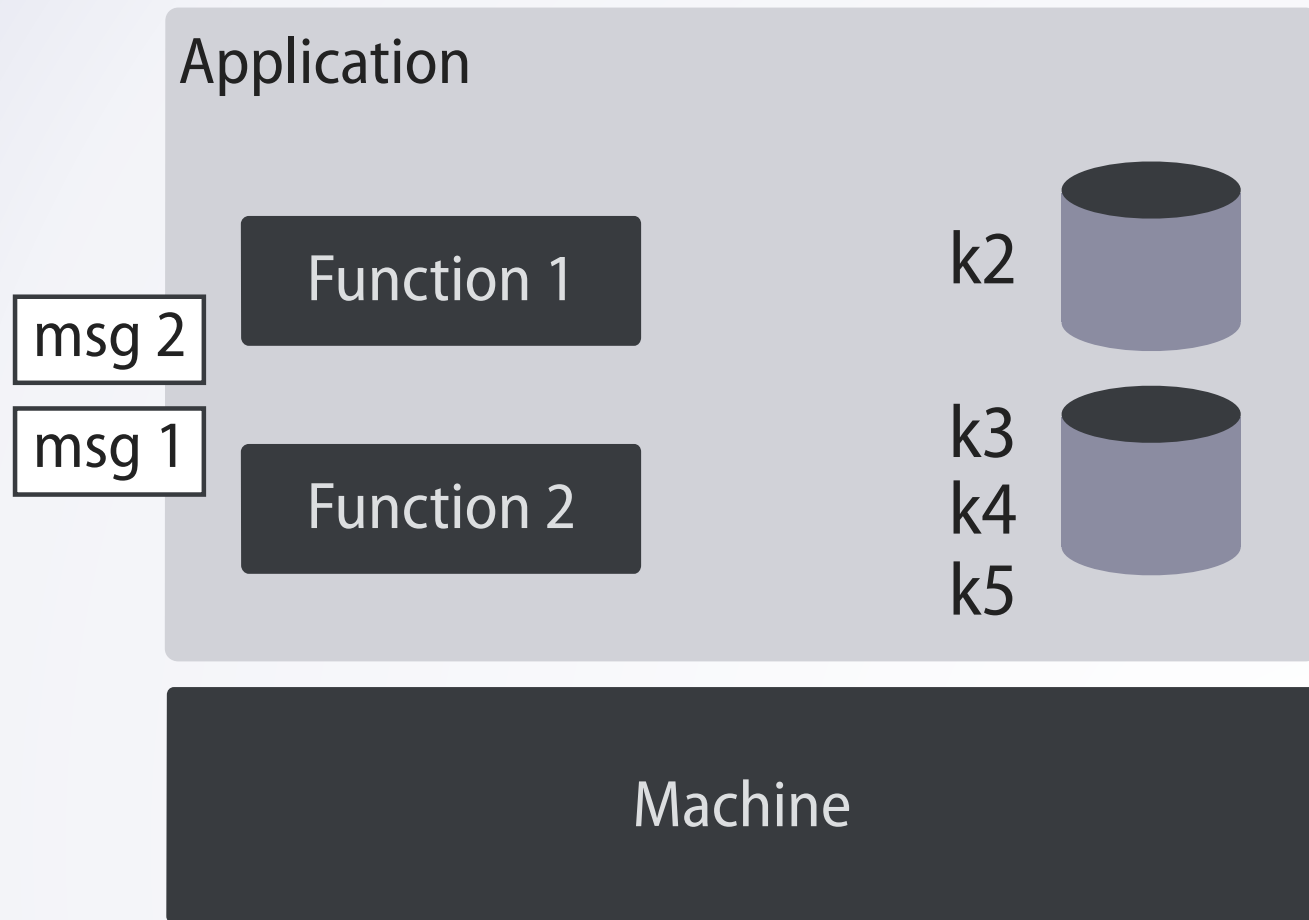


CONSISTENCY

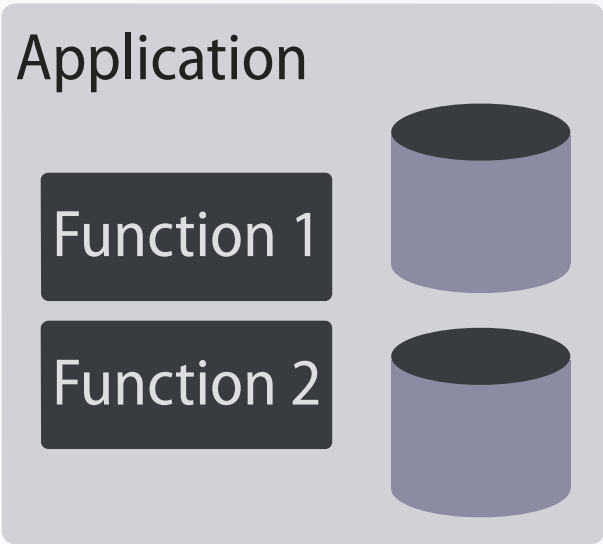
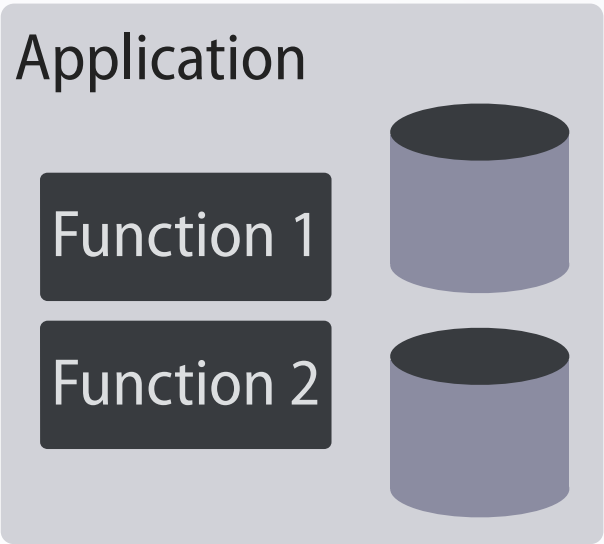
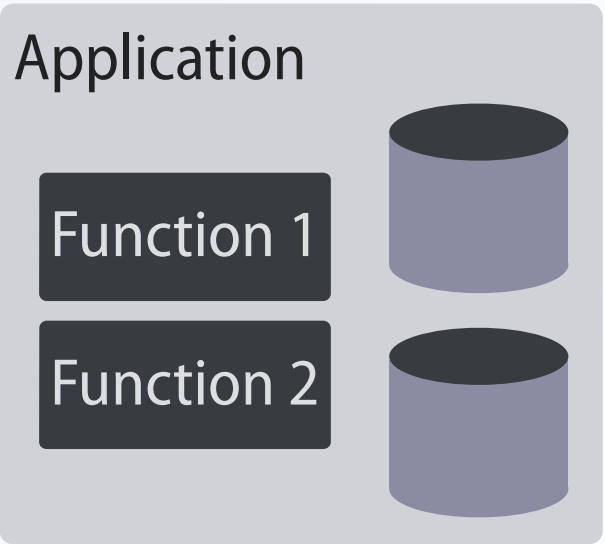
msg 3	k1
msg 2	k2 k3 k5
msg 1	k2 k4



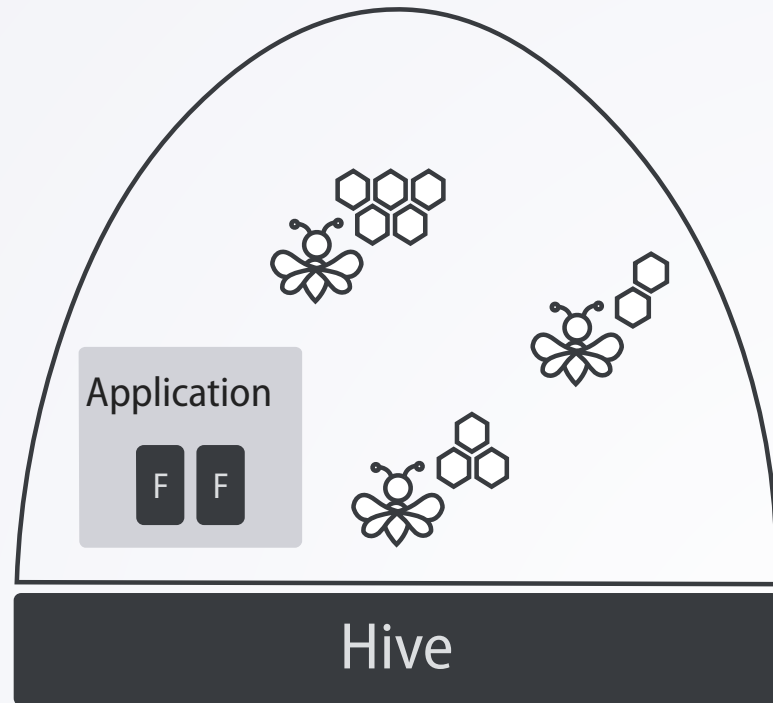
CONSISTENCY



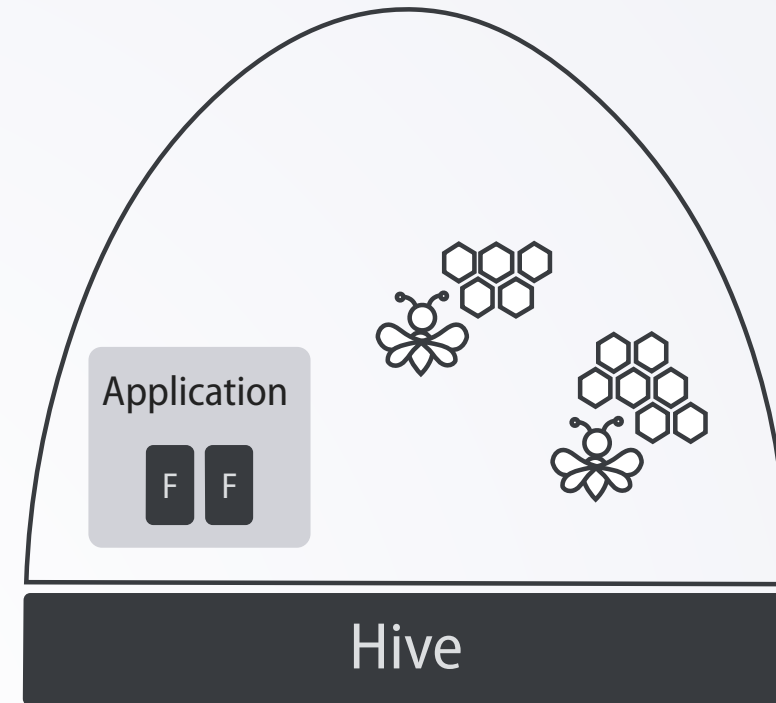
We need a runtime that steers messages among application instances while preserving consistency.



CONTROL PLATFORM

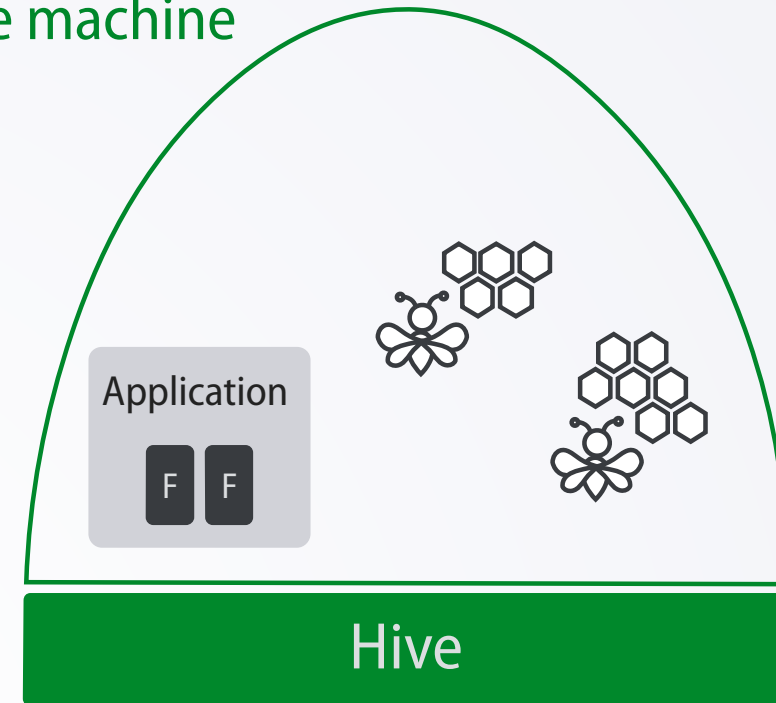
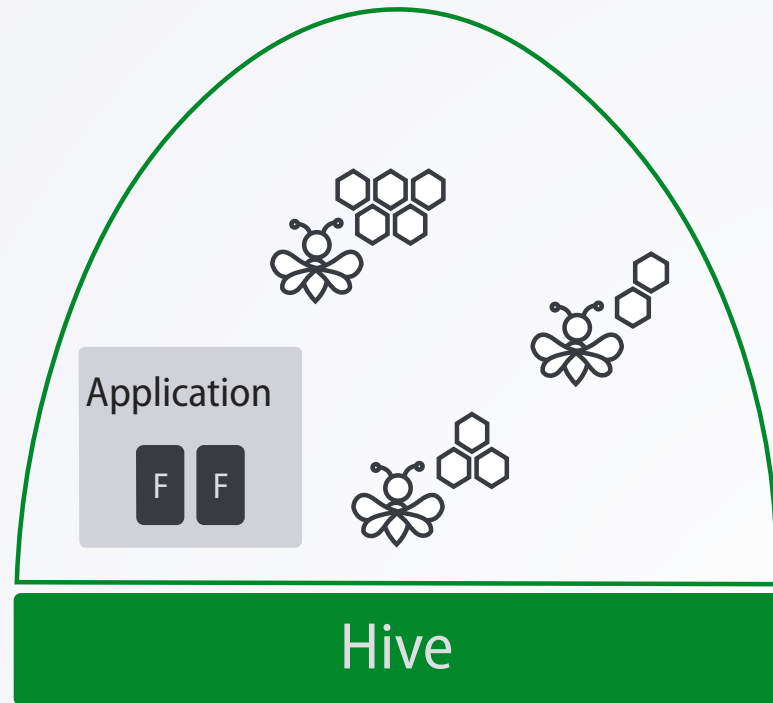


Hive + Cell + Bee



CONTROL PLATFORM

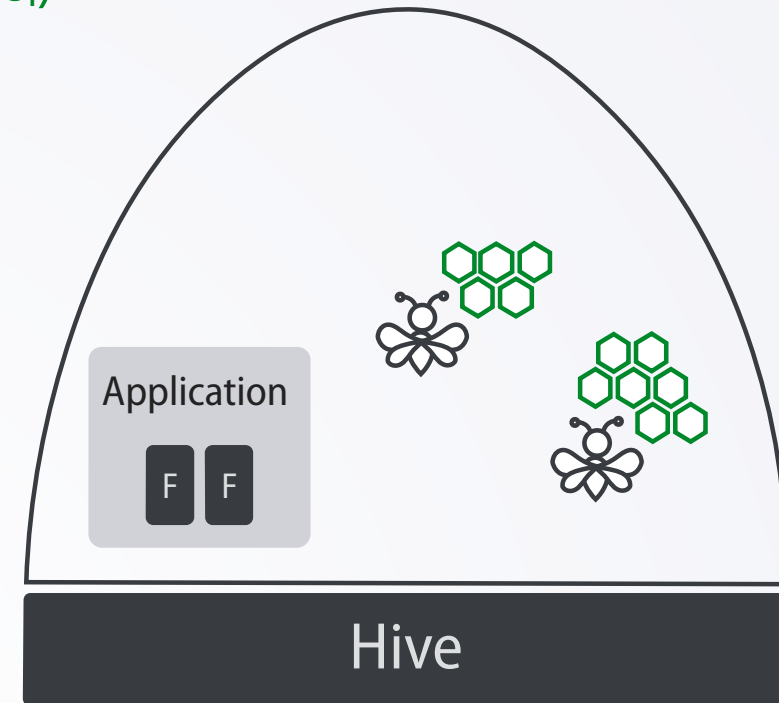
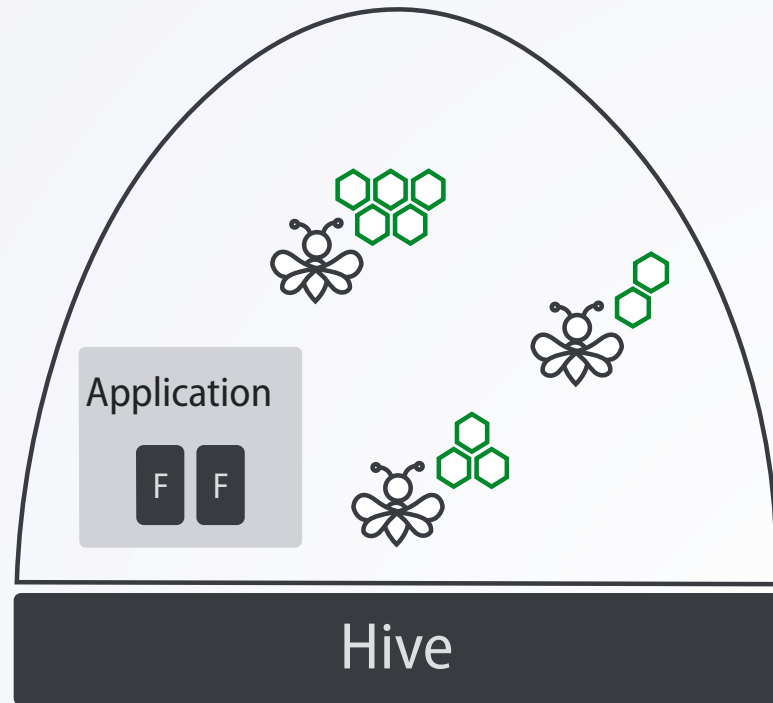
- Hive
- is the controller
 - provides the boilerplates (e.g., locking, consistency, ...)
 - can run on a separate machine



CONTROL PLATFORM

Cell

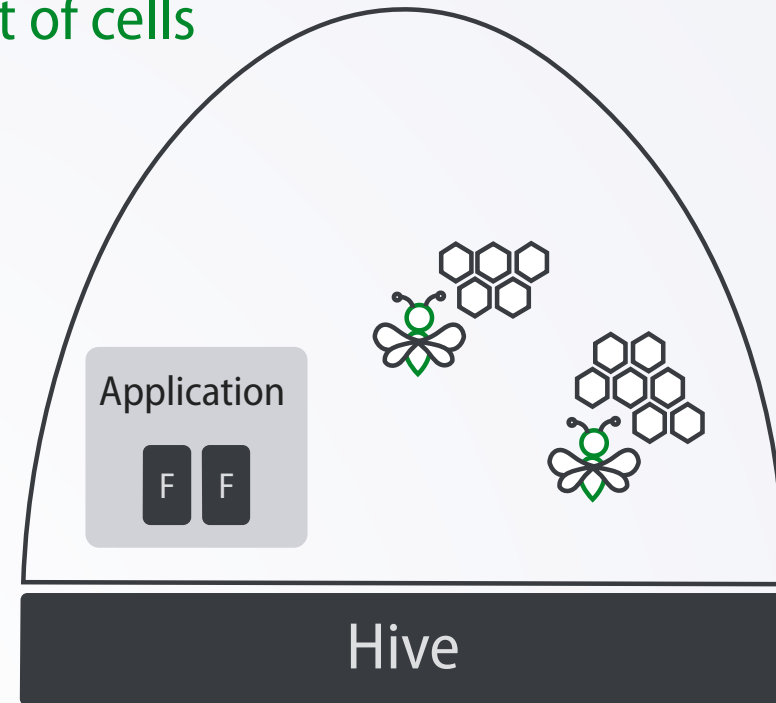
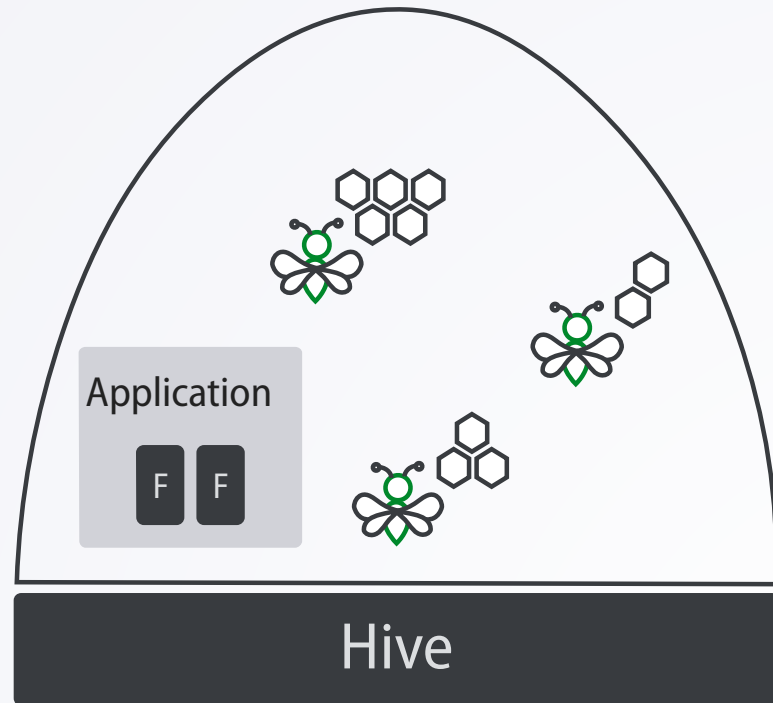
- an entry in a dictionary of a specific application
- e.g., (TE, S, s_i , stats of s_i)



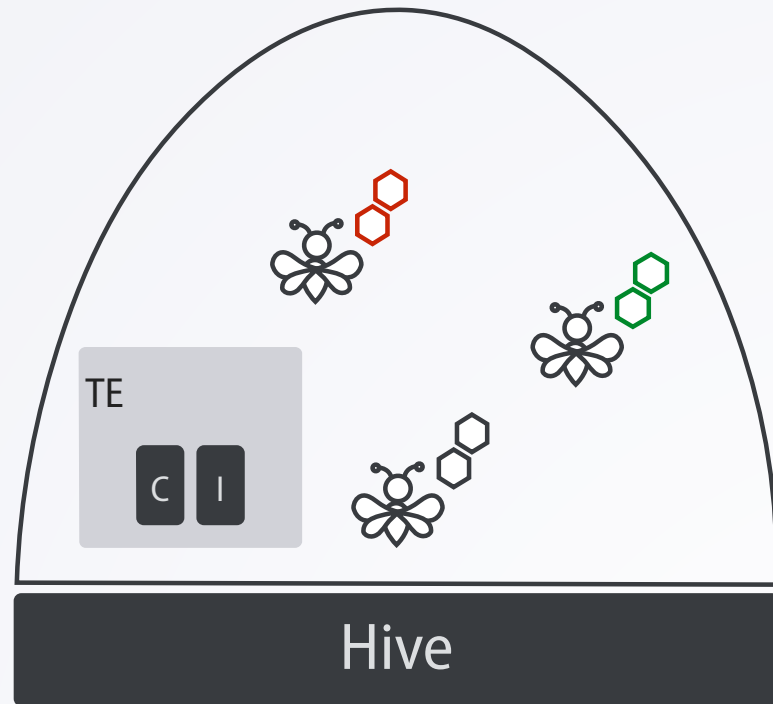
CONTROL PLATFORM

Bee

- a lightweight thread of execution
- process messages
- exclusively owns a set of cells

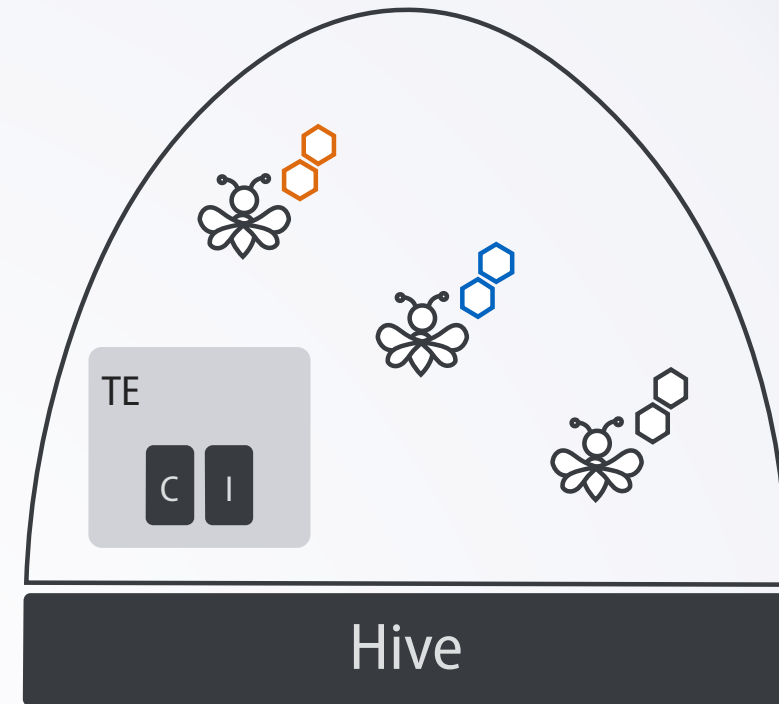


CONTROL PLATFORM



Switch

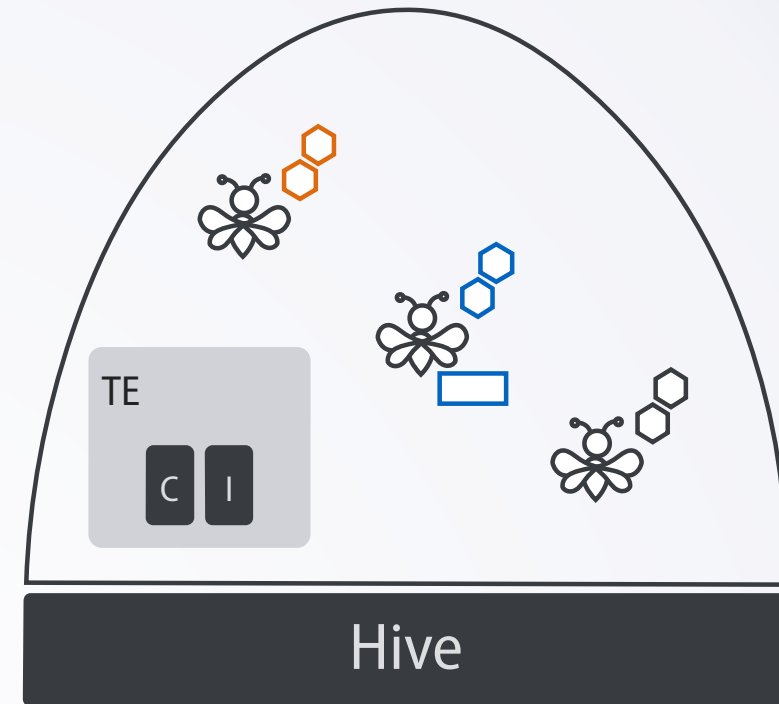
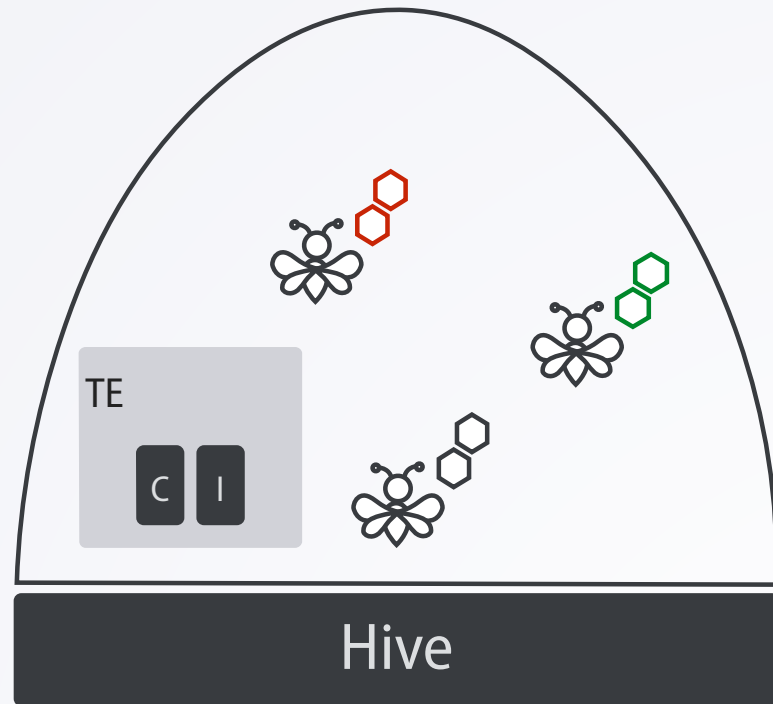
Switch



Switch

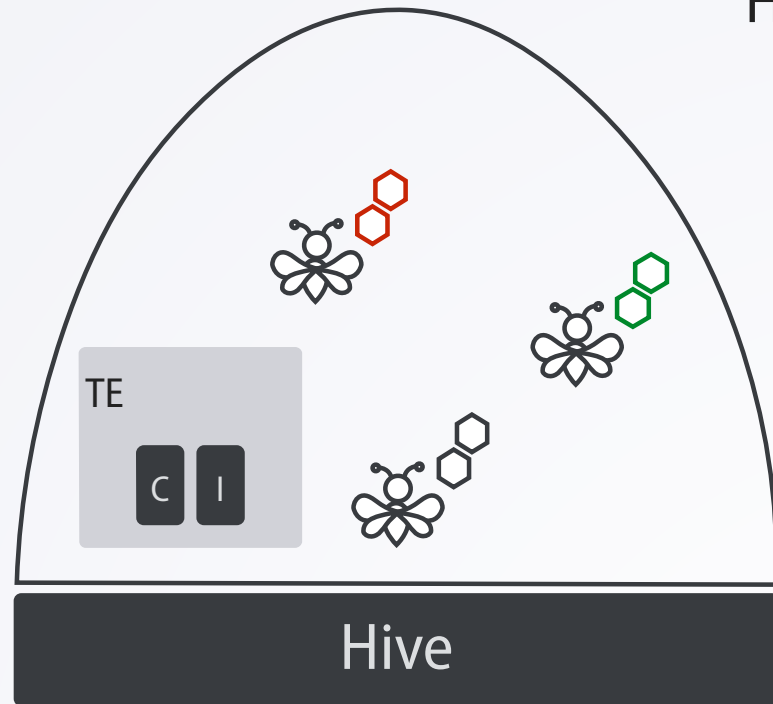
Switch

CONTROL PLATFORM



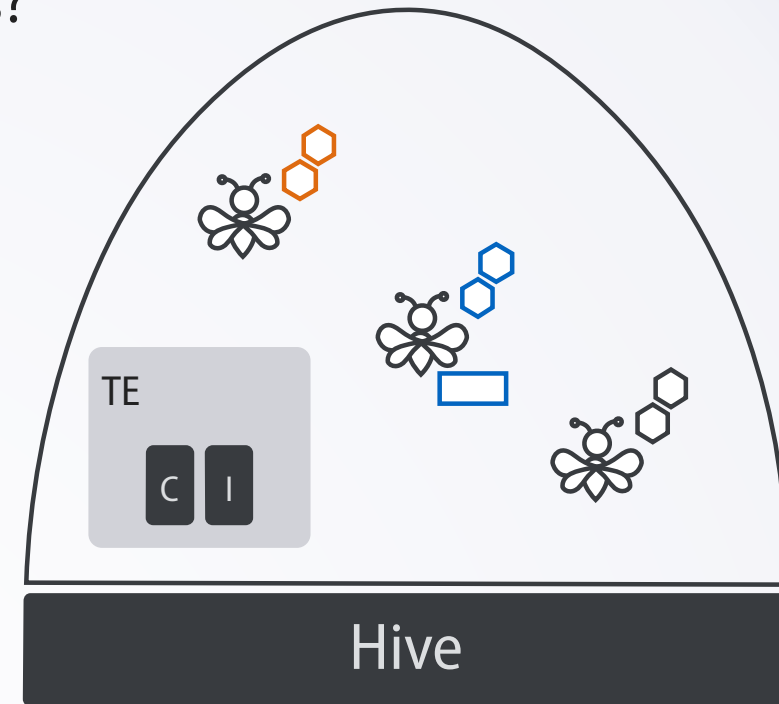
CONTROL PLATFORM

How do we infer the cells?



Switch

Switch



Switch

Switch

CONTROL PLATFORM

How do we infer the cells?

`map(app, msg)` is an application defined function that maps a message to the set of cells used to process that message.

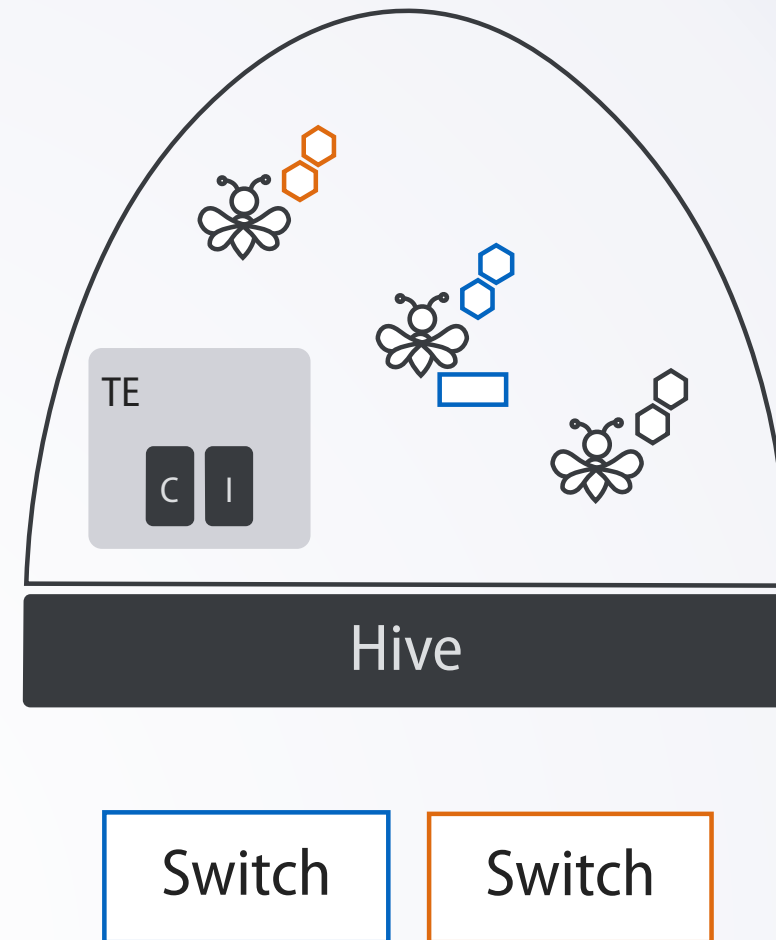
```
func Collect(r, s):  
    s.append(flow stats in r)
```

```
on StatReply(r):  
    Collect(r, S[r.switch])
```

```
map StatReply(r):  
    return (S, r.switch)
```

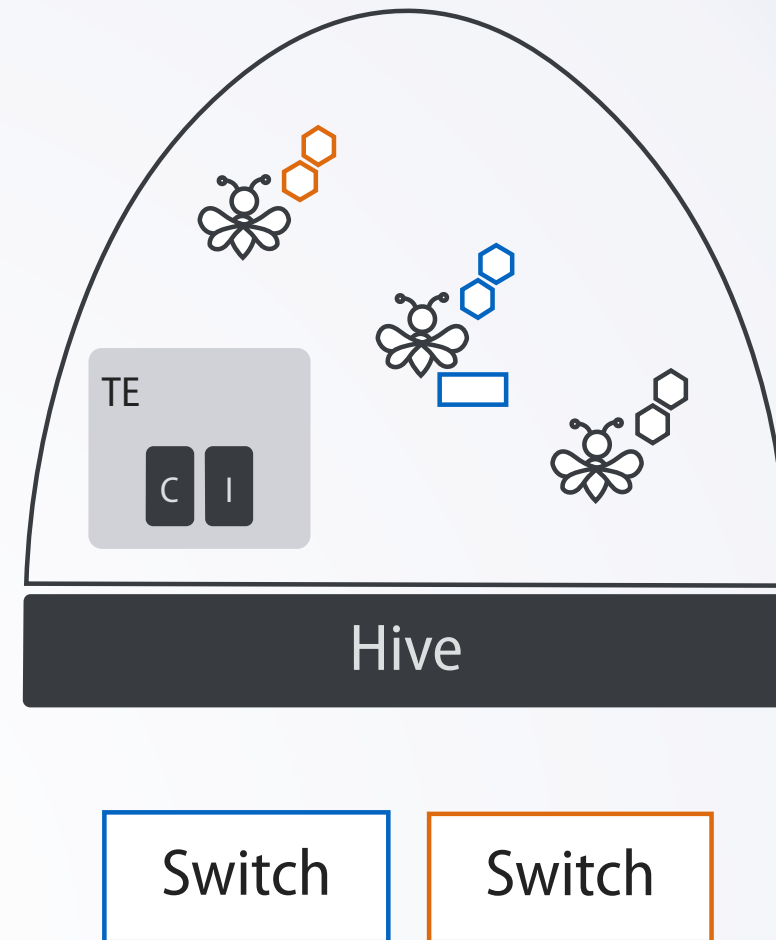
Beehive's compiler can automatically generate the map function.

1-3 lines of code

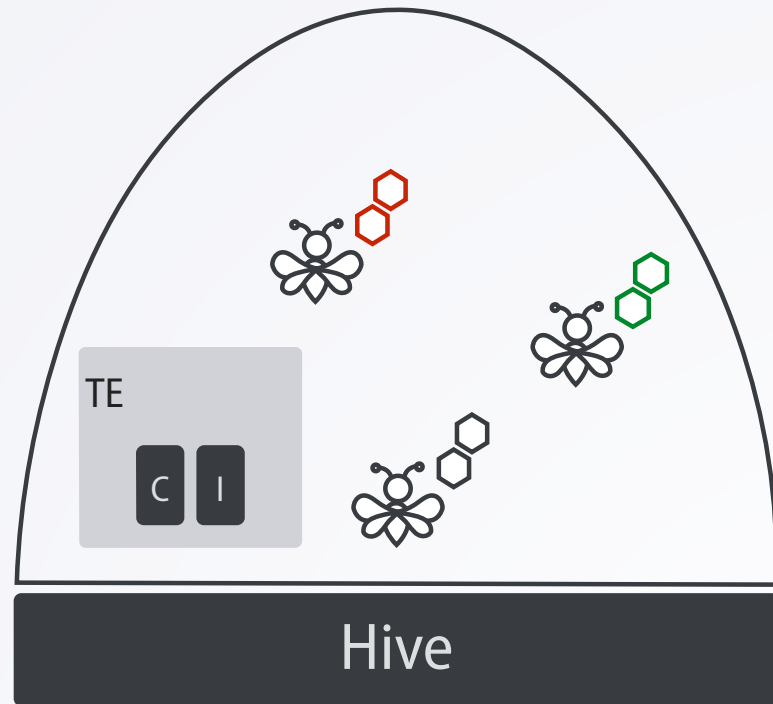


CONTROL PLATFORM

- Function Composition
- Transactions (State + Messages)
- Bee Migration
- Fault tolerance
- Optimized Placement
- Runtime Instrumentation
- Feedback
- Proxied Hives
- ...



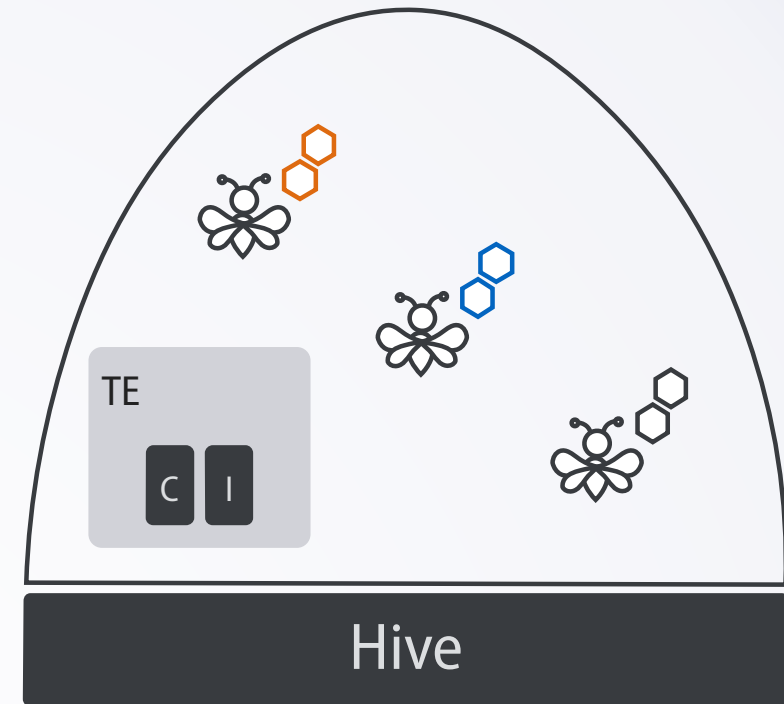
MIGRATION



Switch

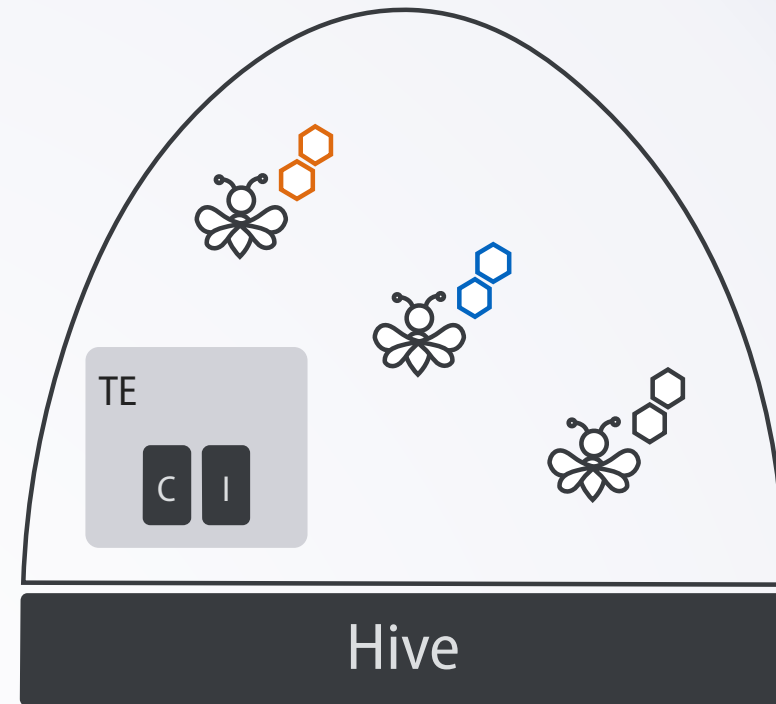
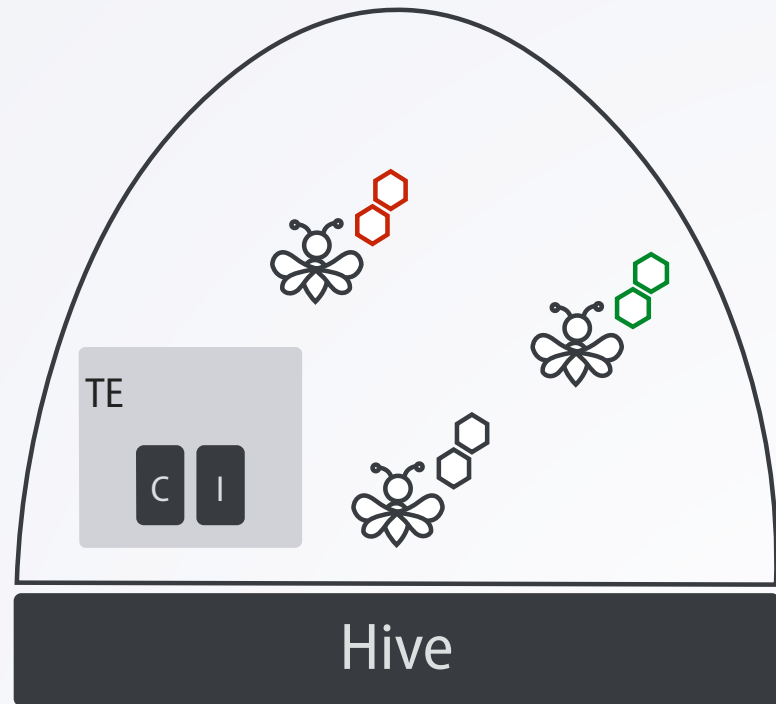
Switch

Switch

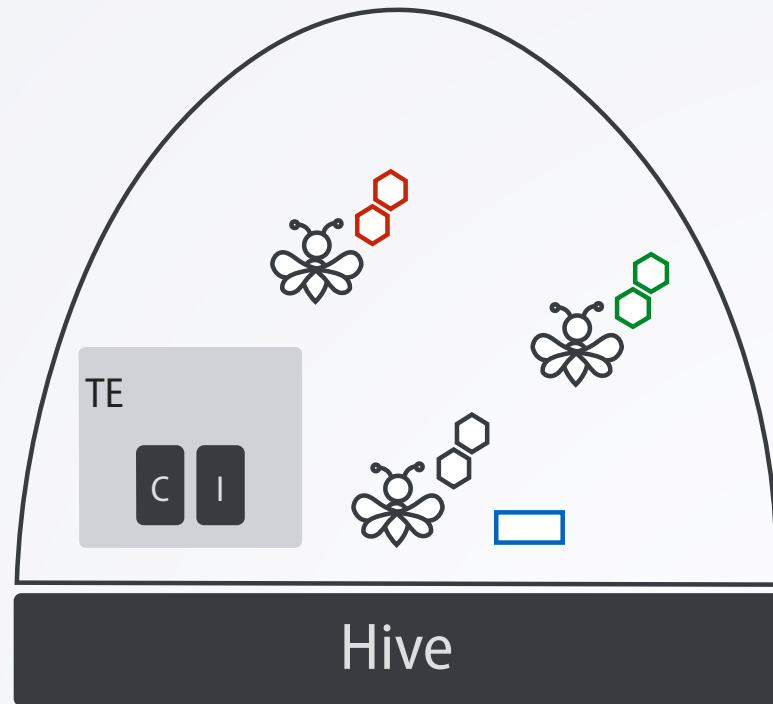


Switch

MIGRATION



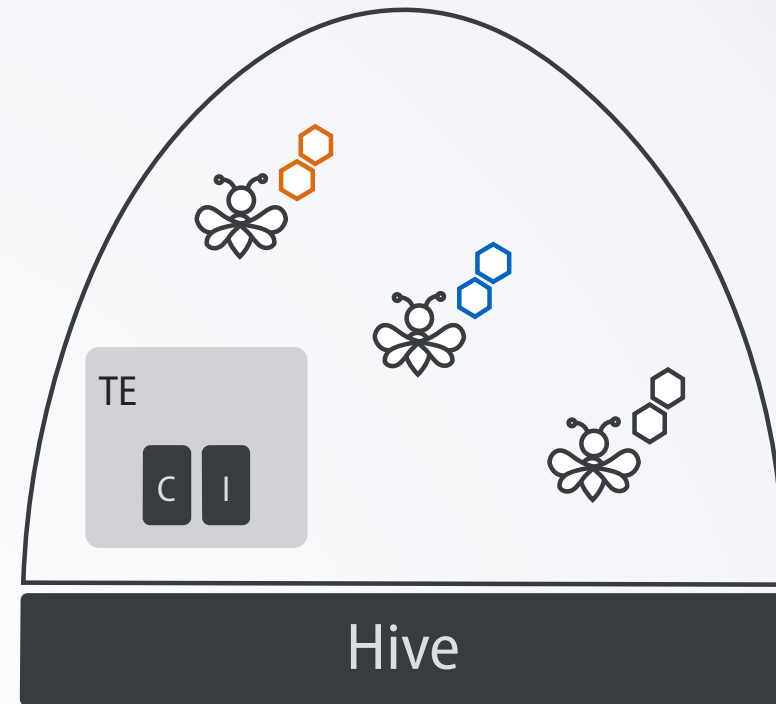
MIGRATION



Switch

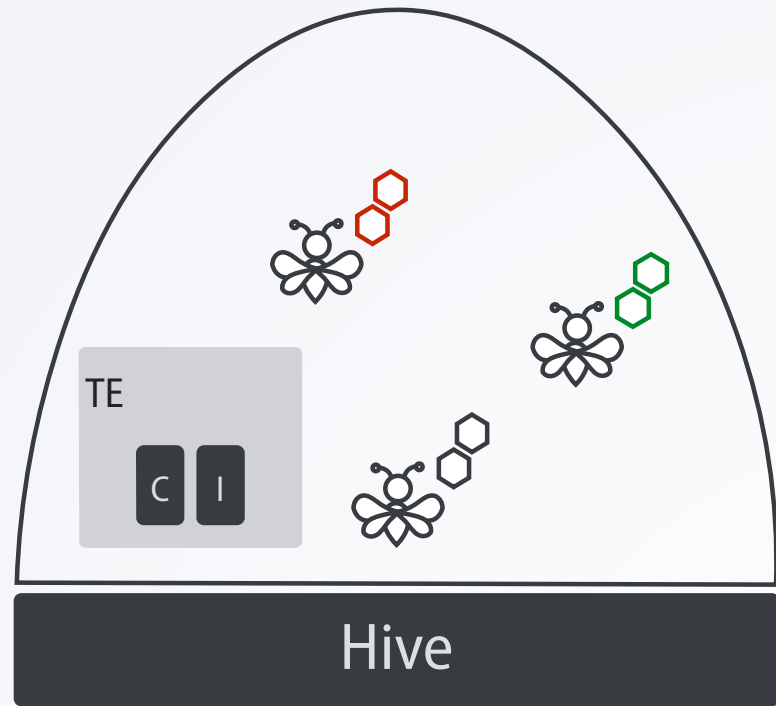
Switch

Switch



Switch

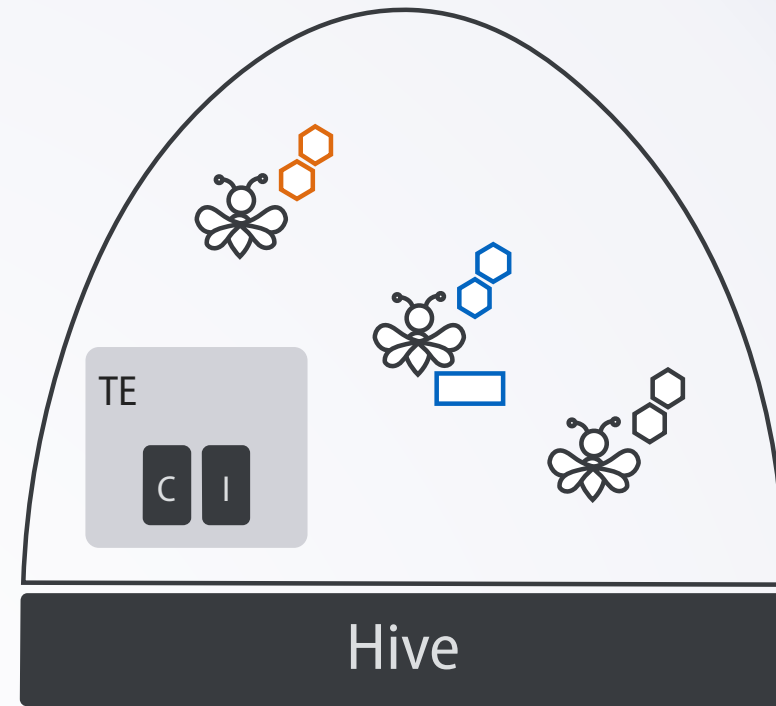
MIGRATION



Switch

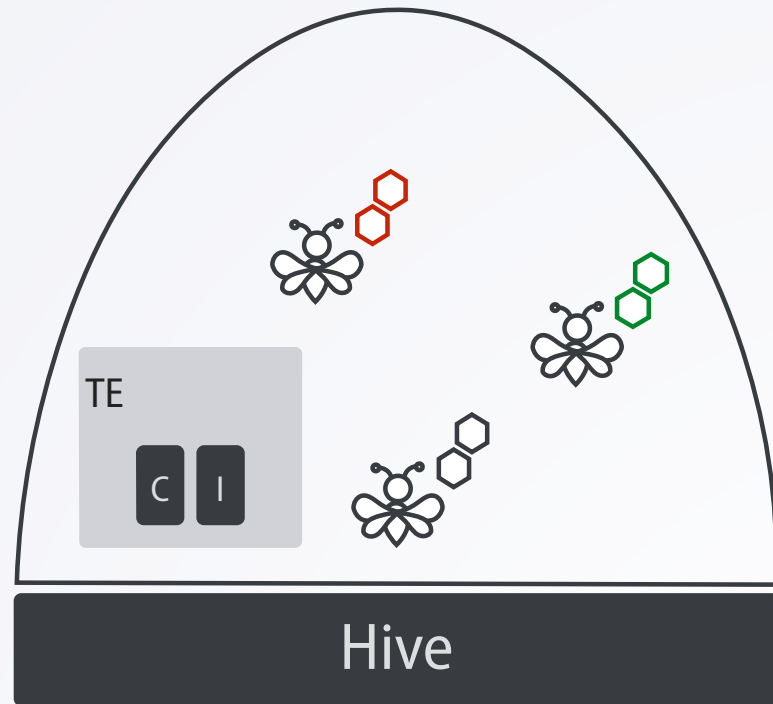
Switch

Switch

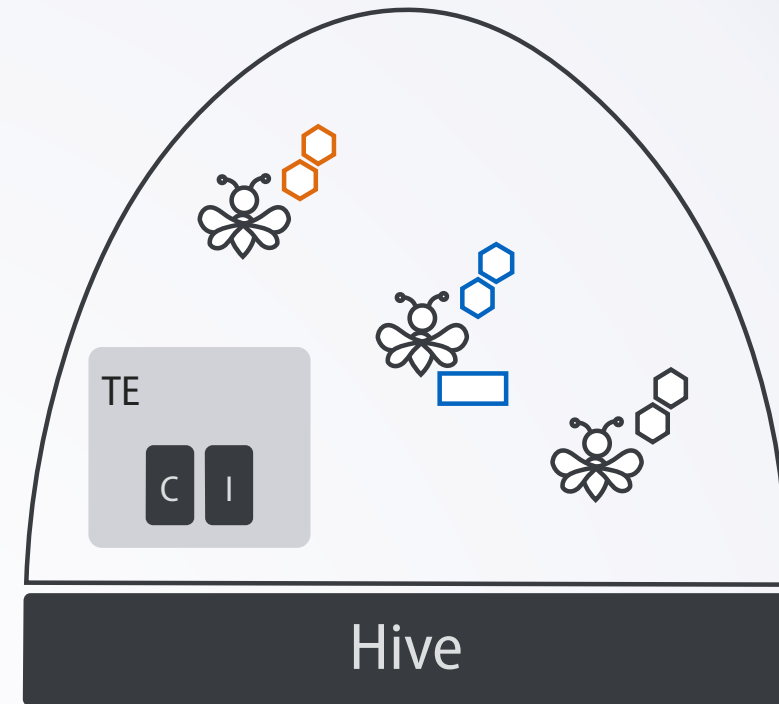


Switch

MIGRATION



This is not optimal and can happen often.



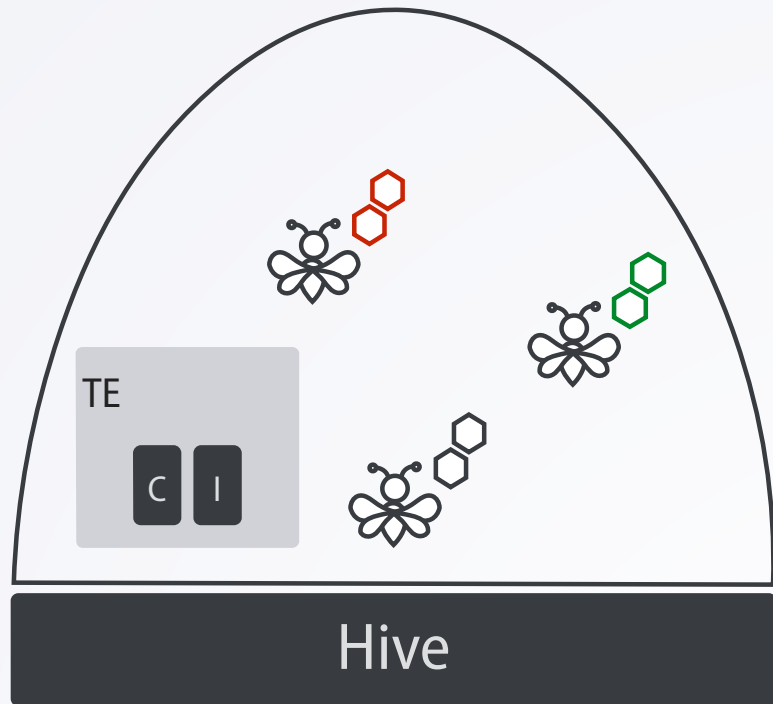
Switch

Switch

Switch

Switch

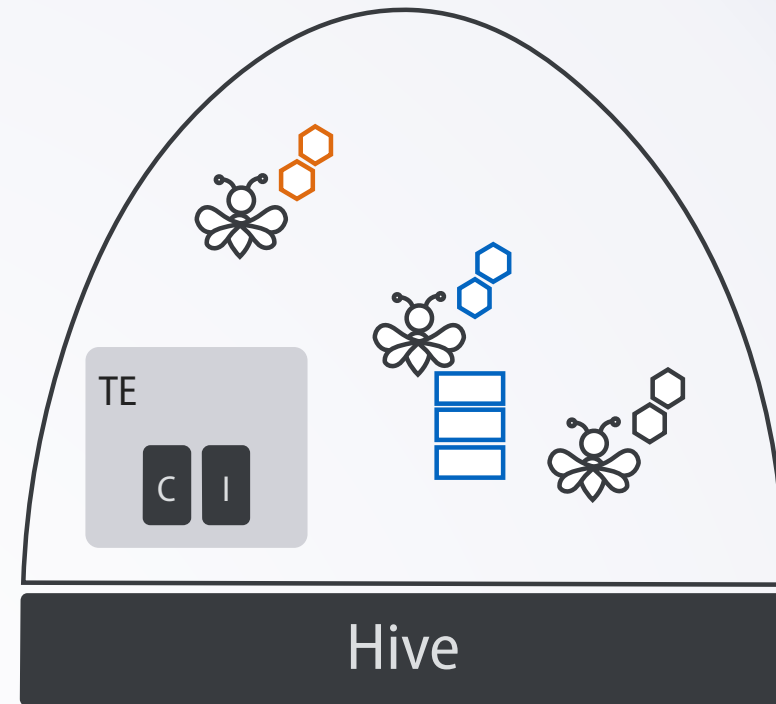
MIGRATION



Switch

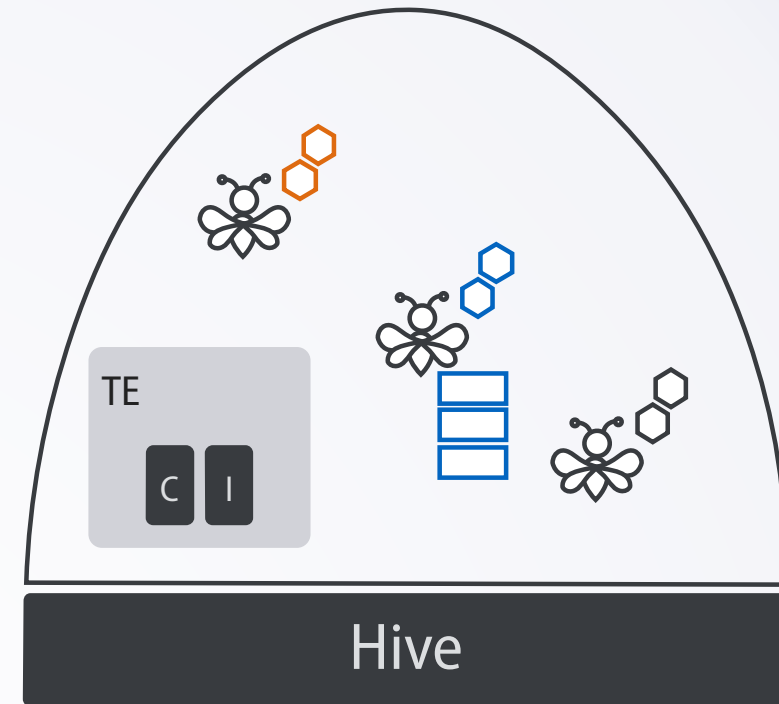
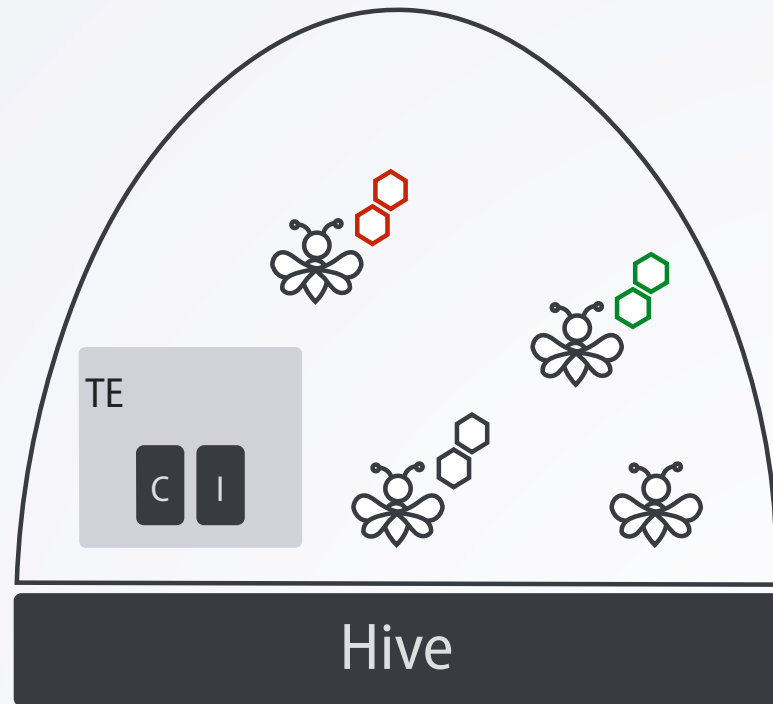
Switch

Switch

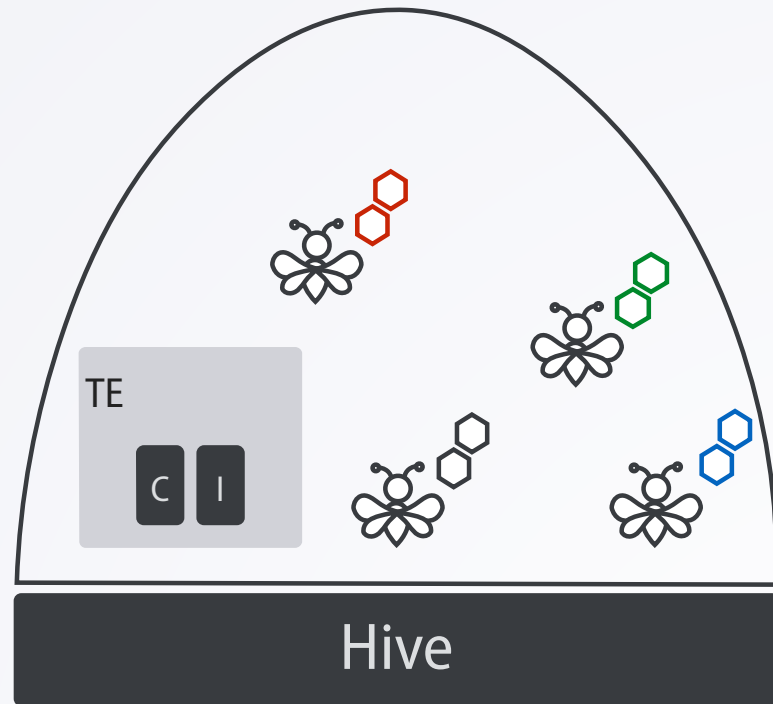


Switch

MIGRATION



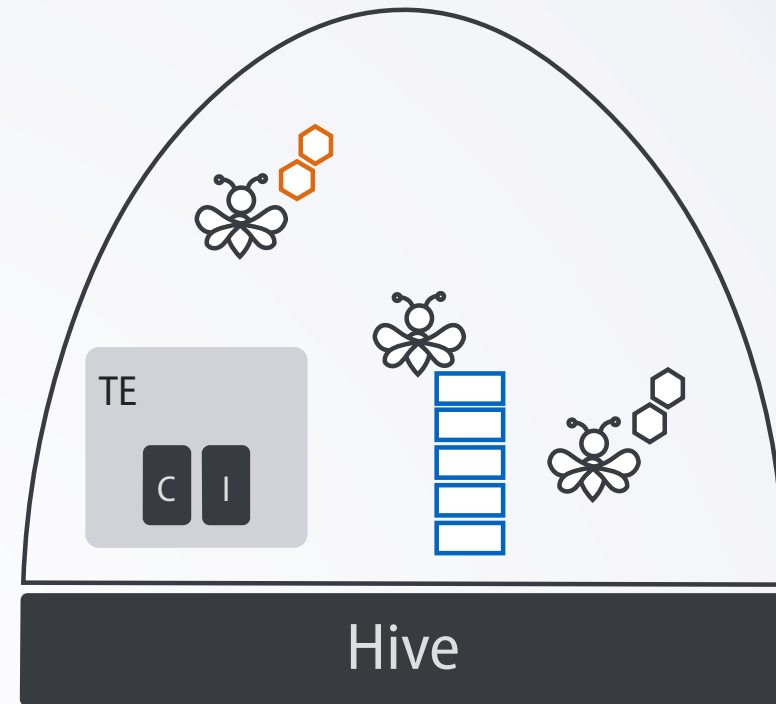
MIGRATION



Switch

Switch

Switch

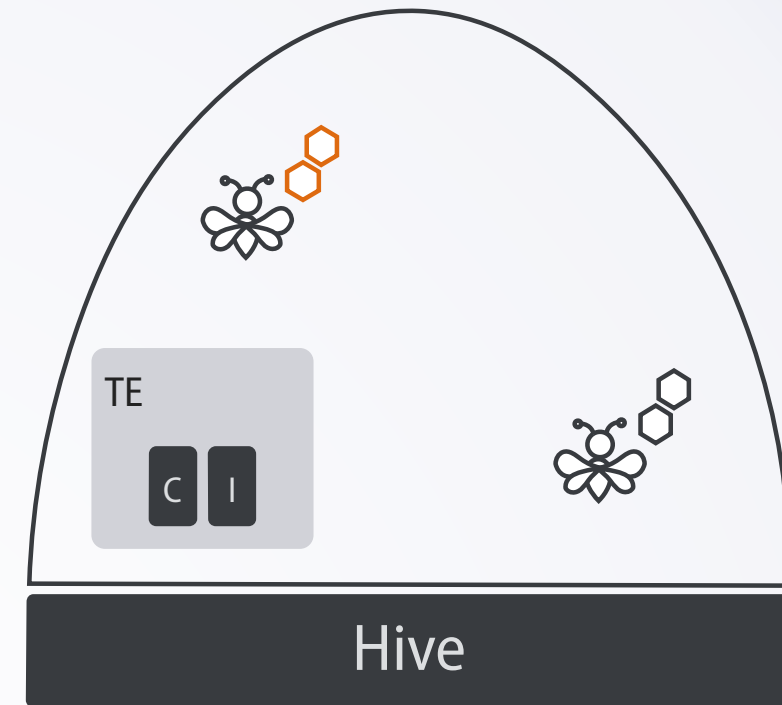
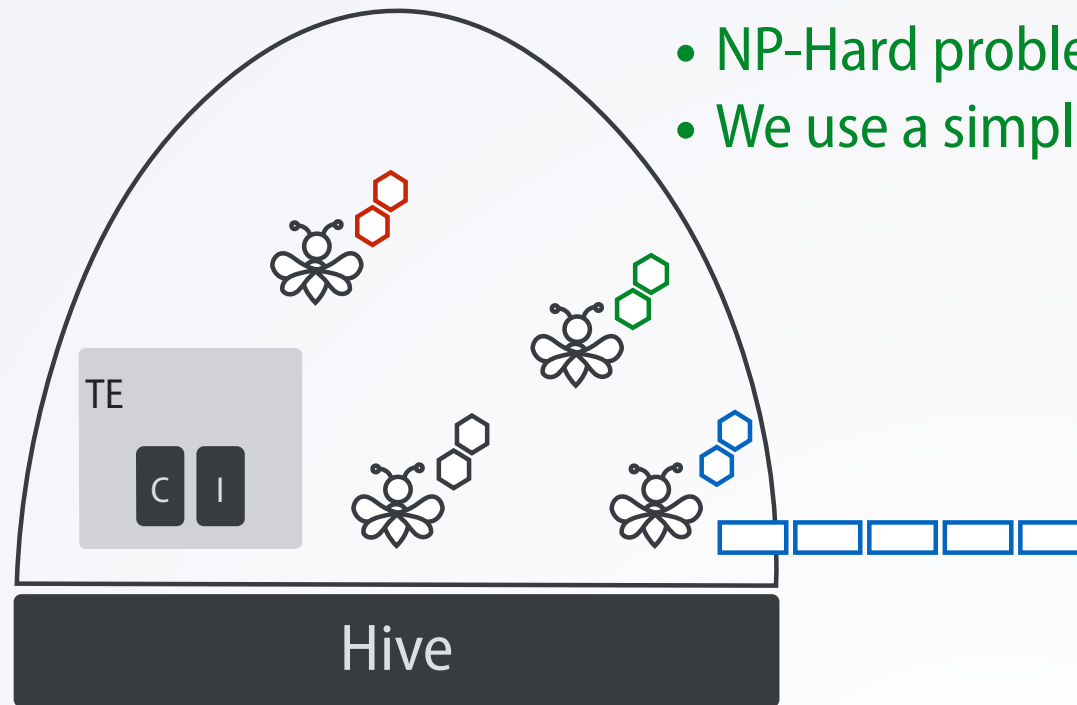


Switch

MIGRATION

When/where should we migrate bees?

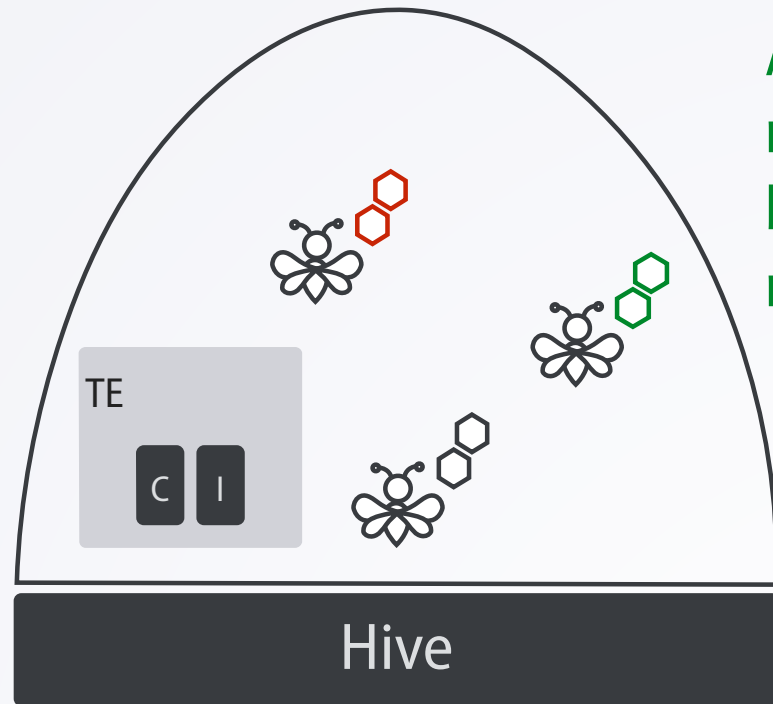
- NP-Hard problem
- We use a simple heuristic



OPTIMIZED PLACEMENT

Our heuristic

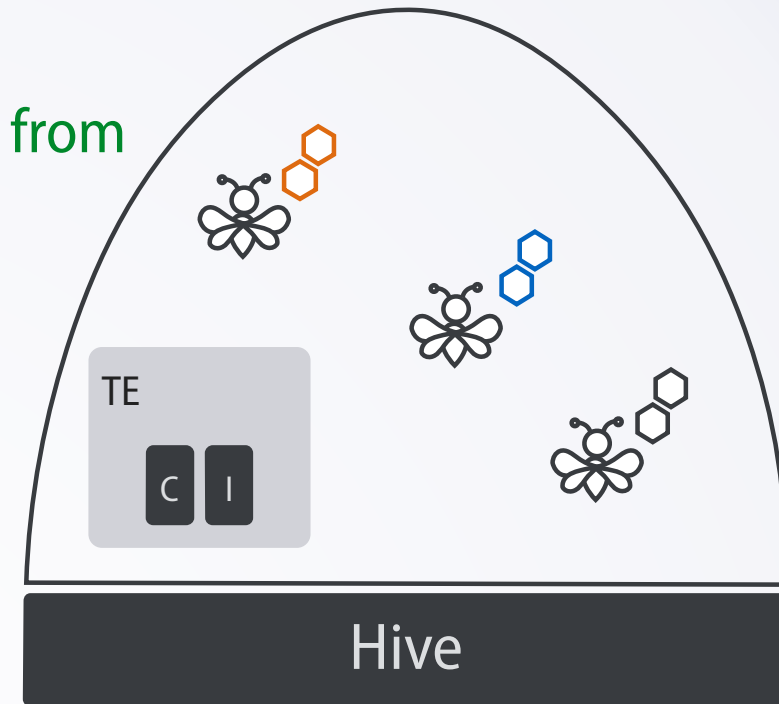
A bee that receives the majority of its messages from bees on another hive is migrated to that hive.



Switch

Switch

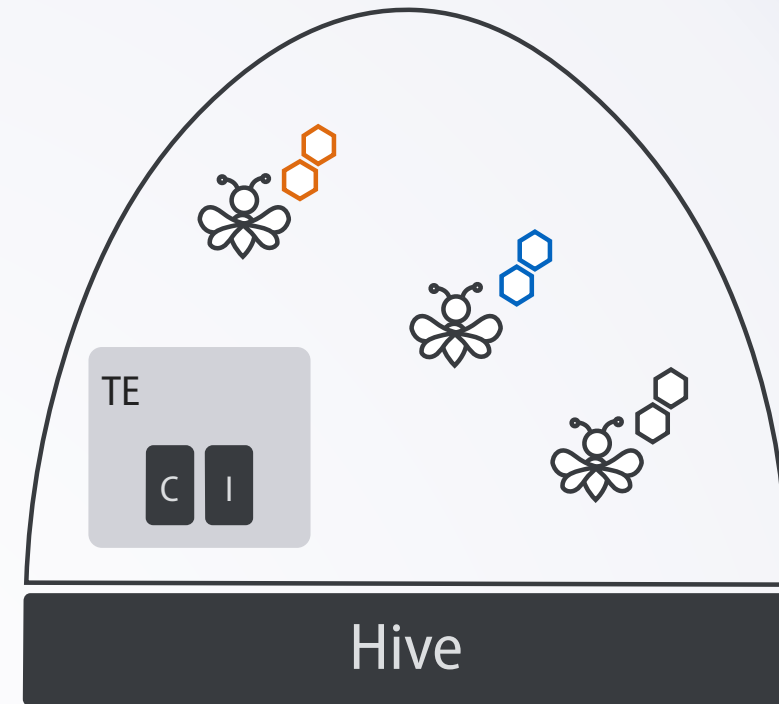
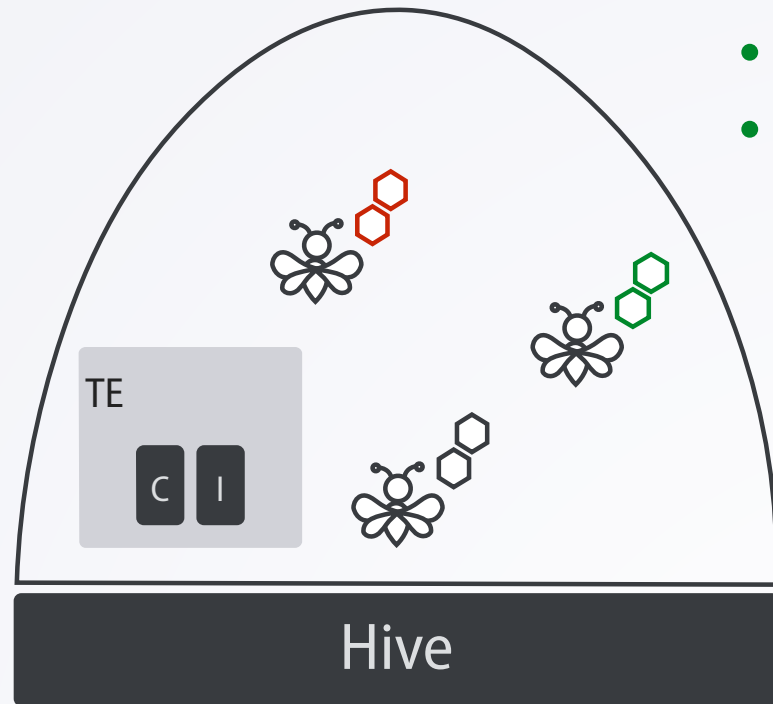
Switch



Switch

RUNTIME INSTRUMENTATION

- traffic matrix among bees
- resource consumption
- message provenance



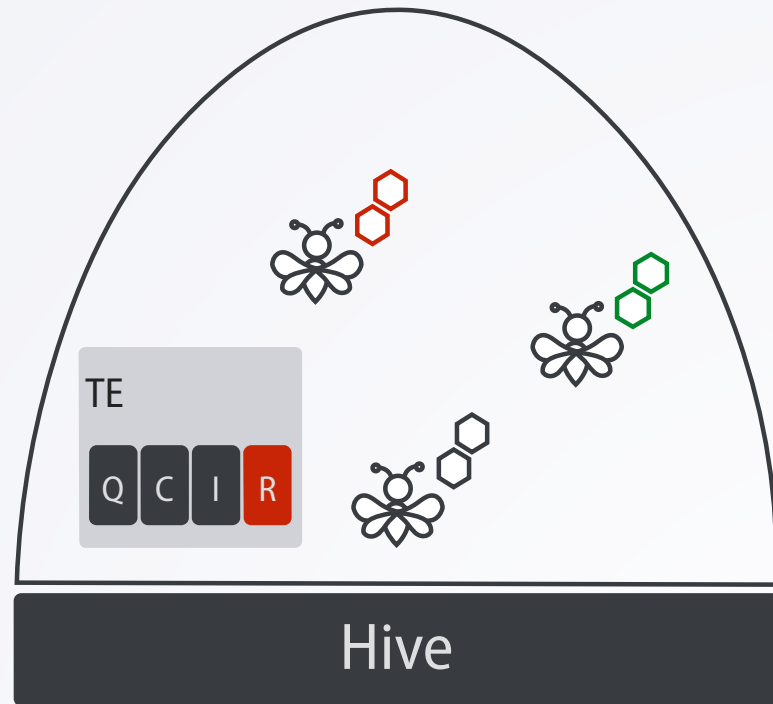
Switch

Switch

Switch

Switch

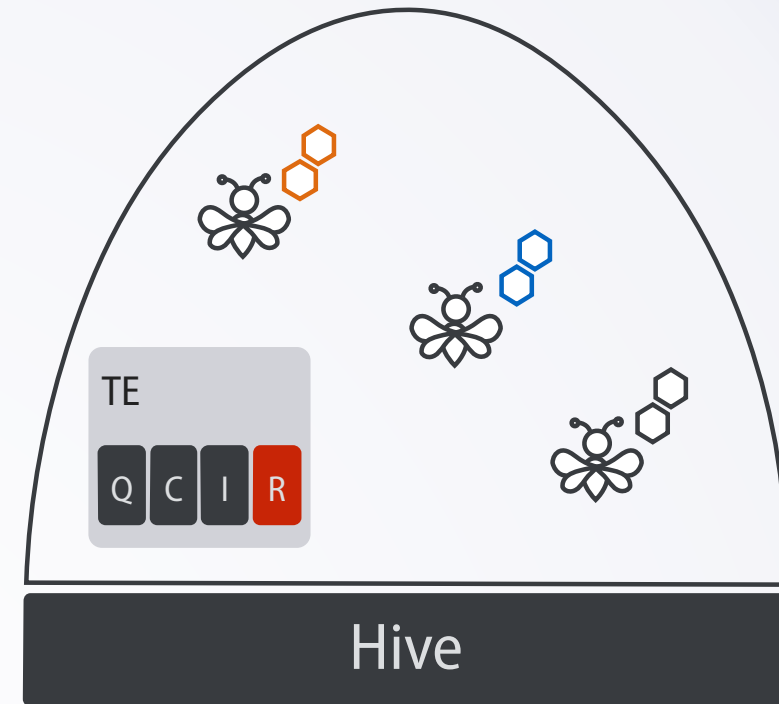
ANALYTICS & FEEDBACK



Switch

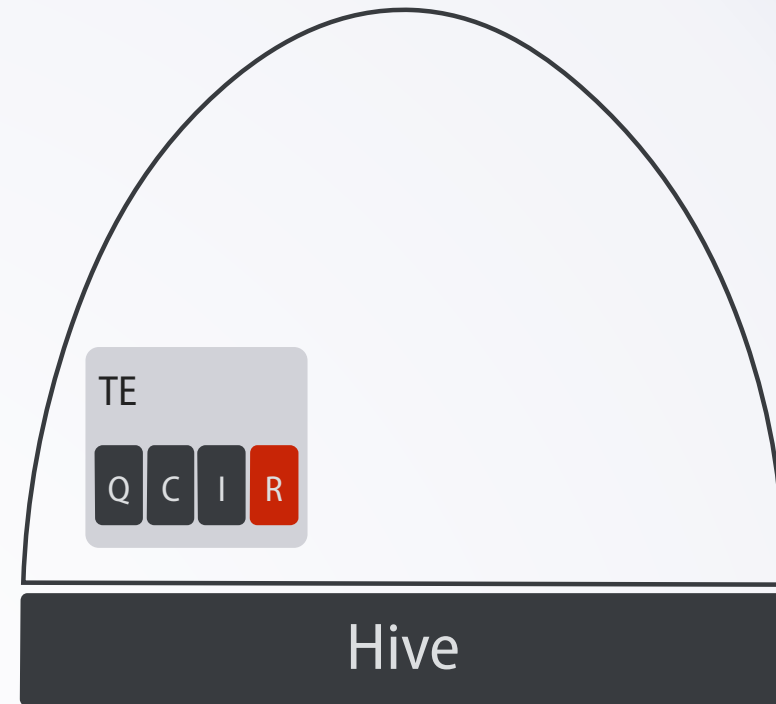
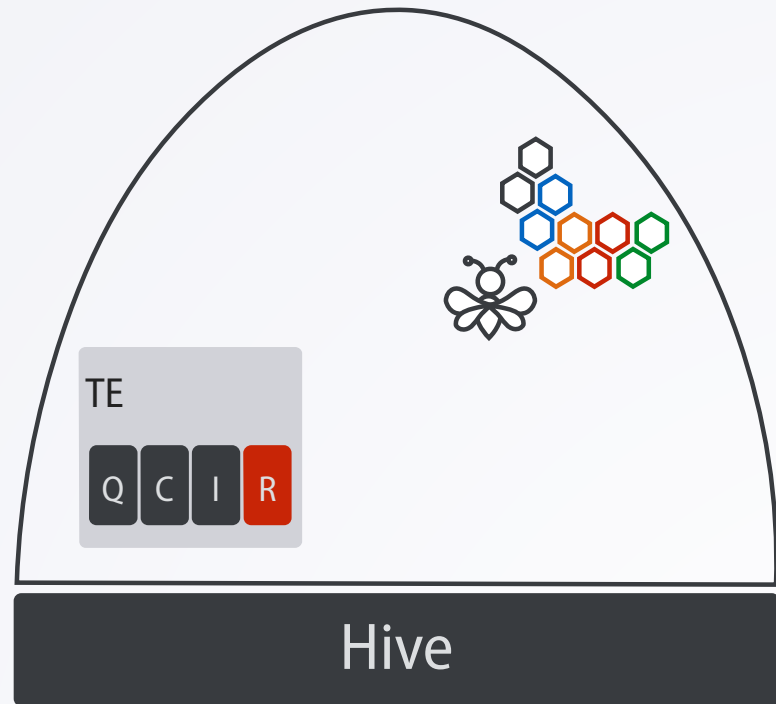
Switch

Switch

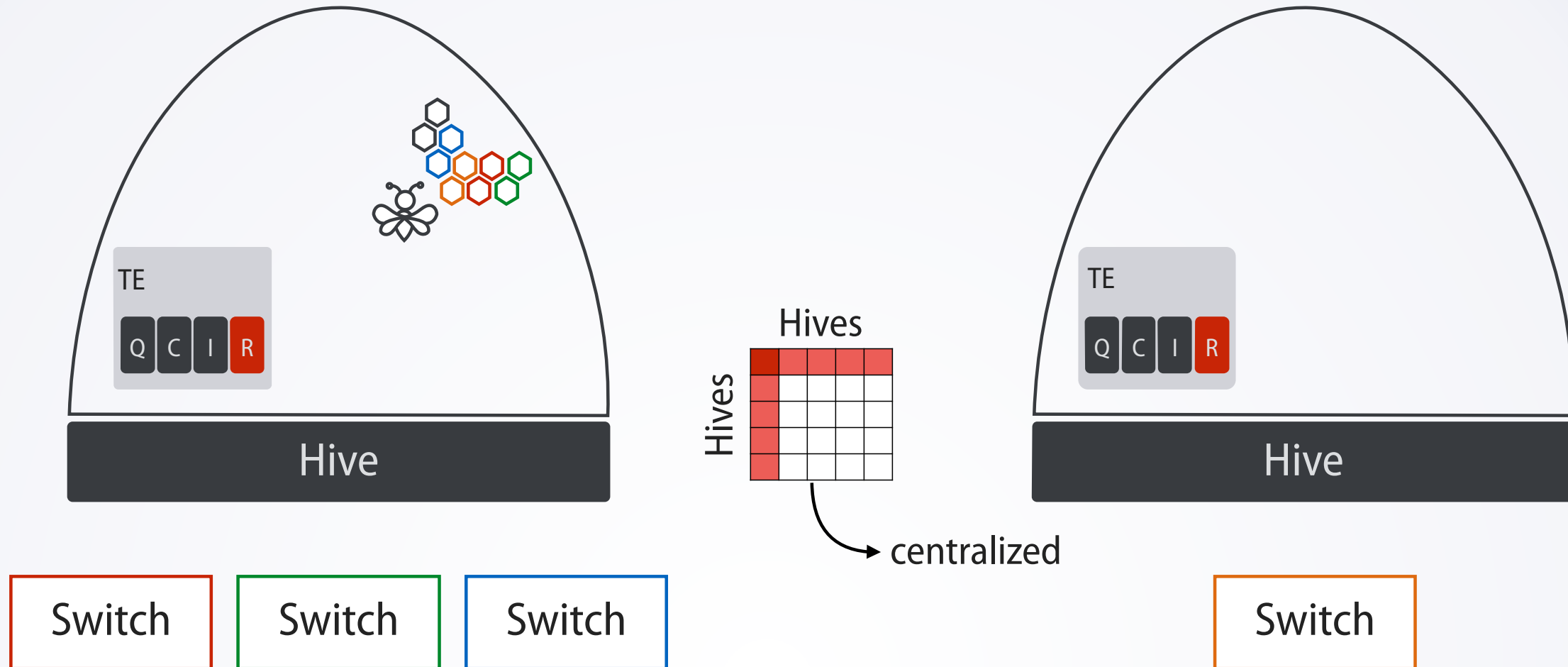


Switch

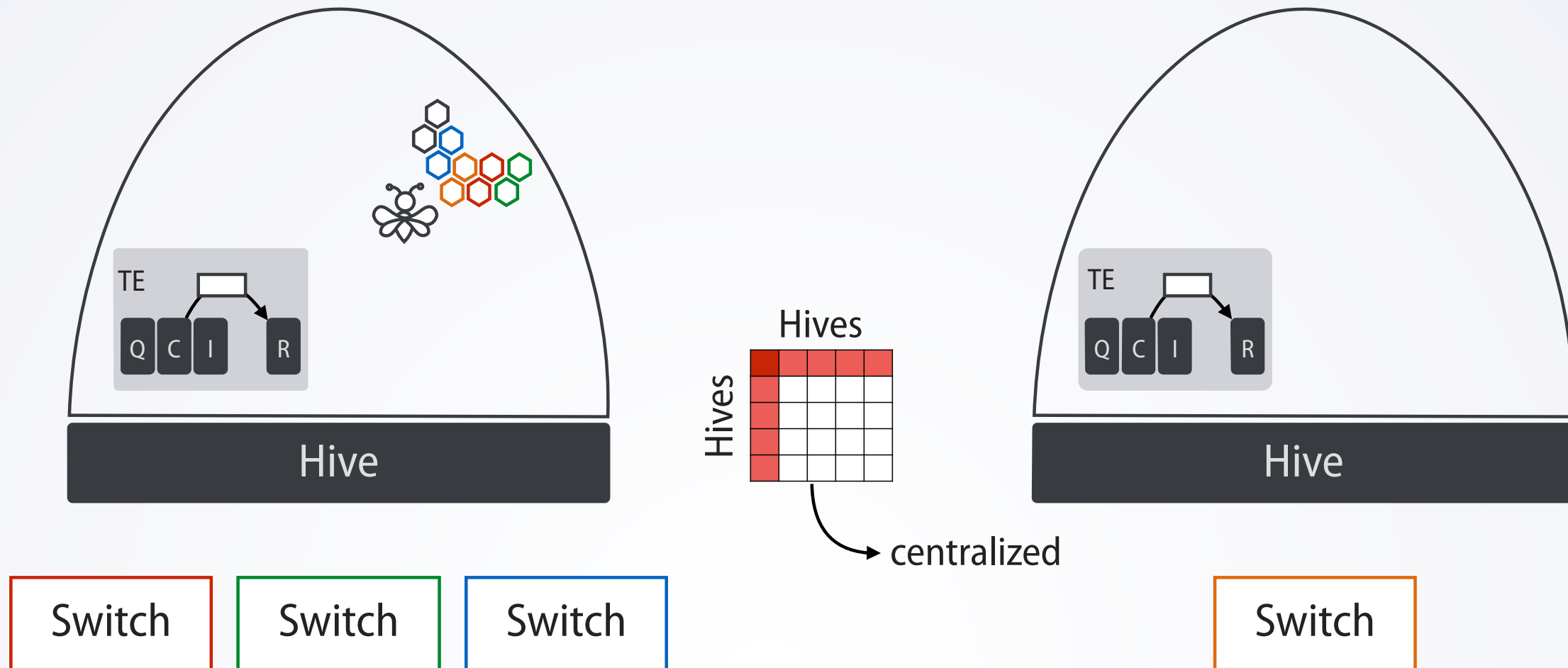
ANALYTICS & FEEDBACK



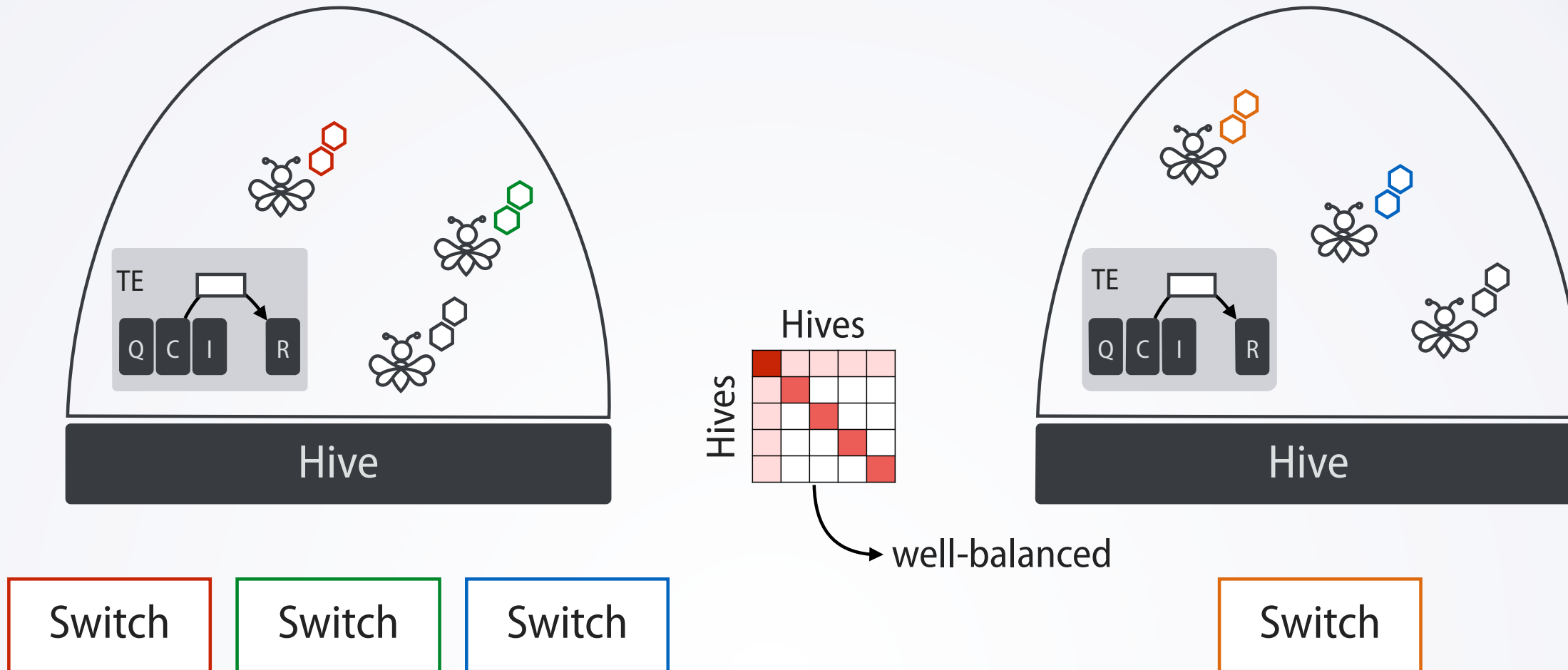
ANALYTICS & FEEDBACK



ANALYTICS & FEEDBACK

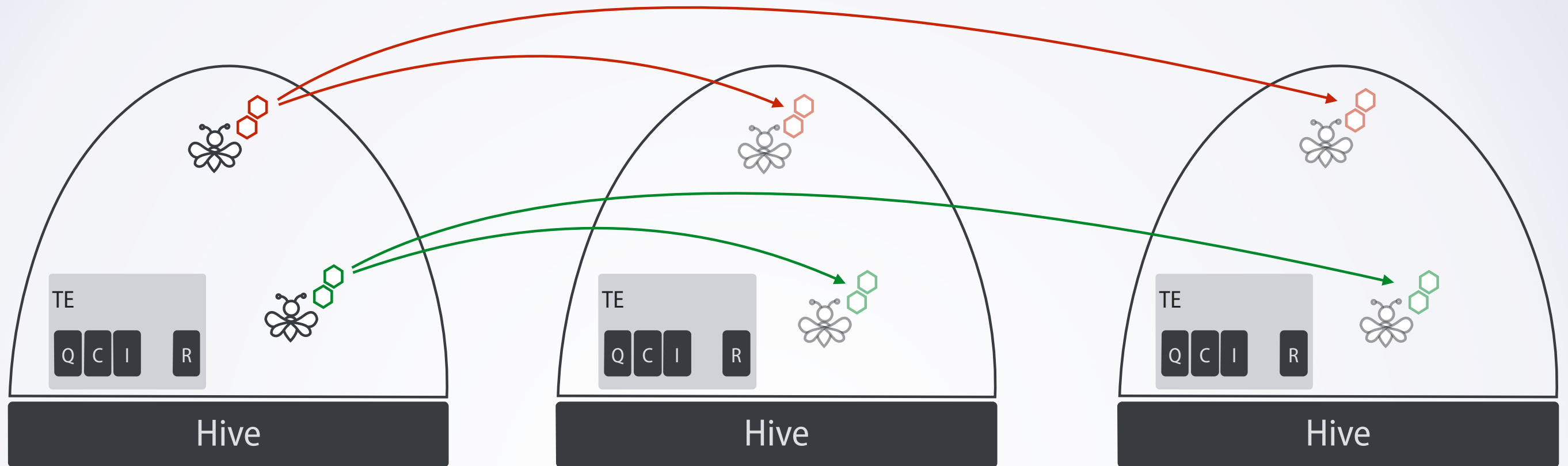


ANALYTICS & FEEDBACK



FAULT TOLERANCE

Colony of replicated bees all in consensus about their state.



GENERALITY

Centralized

```
func Centralized(msg):  
    ...  
  
map Centralized(msg):  
    return {(D, 0)}
```

Virtual Networking

```
func VN(msg):  
    ...  
  
map VN(msg):  
    return {(VN, vnid)}
```

Kandoo

```
func Local(msg):  
    ...  
  
map Local(msg):  
    return {(D, hiveid)}
```

Routing

```
func Router(msg):  
    ...  
  
map Router(msg):  
    return {(Adv, msg.n[0])}
```

NIB

```
func NIB(msg):  
    ...  
  
map NIB(msg):  
    return {(N, nodeid)}
```

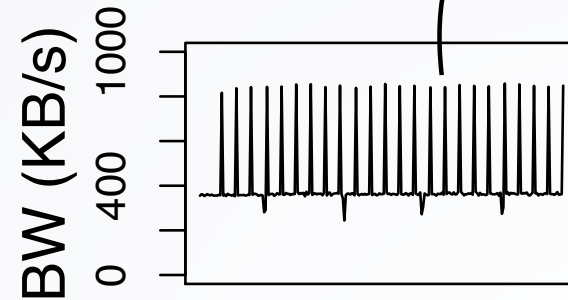
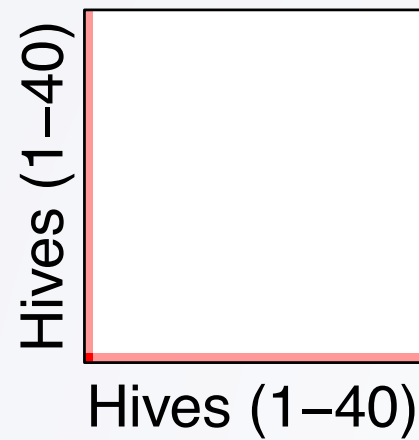
+ you don't need to think about placement and load balancing in most cases.

IMPLEMENTATION

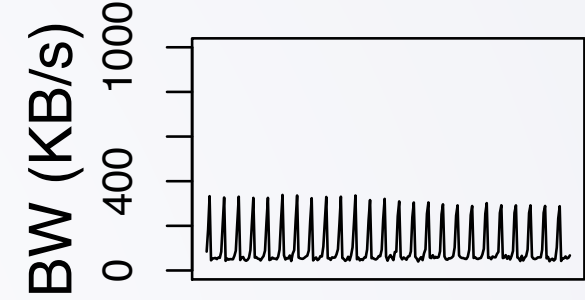
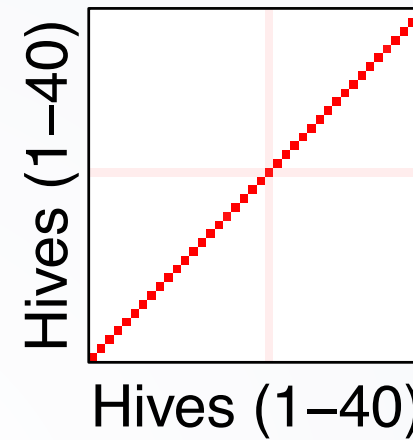
- Free & Open Source, written in Go:
 - <https://github.com/kandoo/beehive>
 - <https://github.com/kandoo/beehive-netctrl>
- No external dependency in the most recent version
- OpenFlow bindings are generated from high level specs:
 - <https://github.com/packet/packet>

EVALUATION

- The TE application
- Simulated environment
- A 40 node cluster on GCE



Centralized

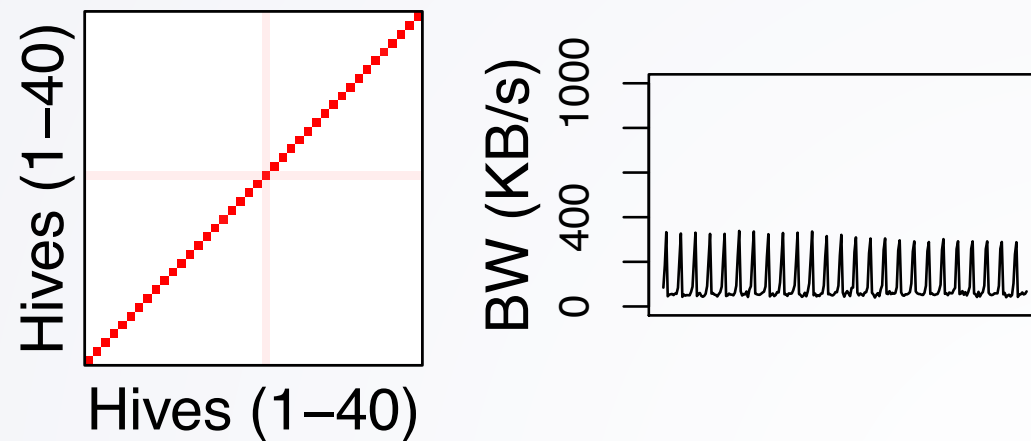


Decoupled

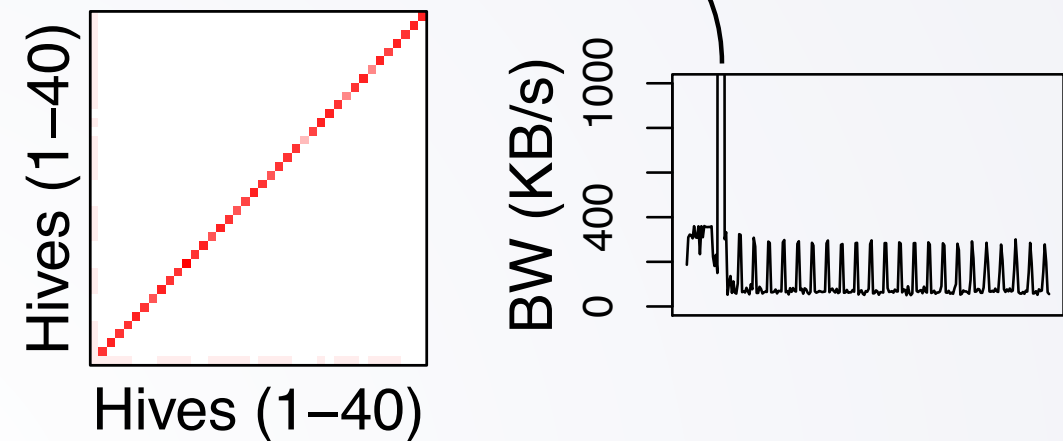
EVALUATION

- The TE application
- Simulated environment
- A 40 node cluster on GCE

This spike is for replicating cells on 40 hives. (~4sec.)



Decoupled



All artificially centralized
then dynamically optimized

FINAL REMARKS

- Beehive = Abstraction + Control Platform
 - Almost identical to centralized controllers
 - Dynamically optimized placement
 - Runtime instrumentation and feedback
- Moving forward
 - Strengthen our evaluation
 - Performance optimizations

Distributed programming in SDN doesn't have to be complicated.

THANKS