FormaliSE'16

Toward Rigorous Design of Domain-Specific Distributed Systems

Mohammed S. Al-Mahfoudh
Ganesh Gopalakrishnan
Ryan Statesman
The University of Utah

Outline

- Intro
- Nowadays
  - situation
  - solutions: difficulties + effectiveness
- DS2
  - offers
  - example
  - completion status
- Conclusion

Intro

- **Distributed Systems gone mainstream**
  - Data centers, cloud, IoT,…etc.
  - Notoriously hard to develop+get right
- **Reasoning? barely supported**
  - more productivity + less reasoning =>
- **Worse? no semantic clarity**

Image credit: www.scorpionpictureguide.com => cute bug is parallel processing, scorpion DS

Background

- **Extreme non-determinism**
  - Common Misconceptions
    - fast access, single time frame, fault-freedom, strong-ordering
  - Sadly, distributed systems violate all these!
- **Language generality/imprecision**
  - Domain specific knowledge often not exploited

This morning's lecture, you saw it!
how much effort, time, and dedication it takes

–From Pamela Zave's Talk

What does it take to specify Distributed Systems

- Proving Raft Linearizability in Verdi
- 45K of lines in complete proof
  - 90 *non-trivial* invariants
- 3 man-years to achieve! (2 ppl x 1.5 yrs = 3)
  - I had a kid + another coming + many things < 3 yrs!
- How many LoC actual Raft implementation?

Complete story in [3]

Well Known Issues, Current Approaches

- **Only good for stable systems**
  - During development needs
    - exploration (loose ends)
    - Visualization (improving understanding)
    - Basic Property Checking (e.g. Linearizability)
- **Not scalable (previous slide)**
- **Not widely known in mainstream community**

Current success stories

DSLs: DeLite, P, P#, …etc (Domain Specific Languages)

- Domain implicits exploitation (case specifics handled)
- Clear syntax and semantics (concise+familiar)
- Highly optimized runnable(s) (Delite)
- Multiple backends (heterogeneity handled - Delite)
- High level language (Scala - Delite, C#-P#)

- No (networked) distributed systems support!

DS2 Infrastructure

Domain Specific Distributed Systems Specification and Synthesis

DS2 Infrastructure (Provides/Enables)

- Actor driven model (easy to understand)
- Semantically guided exploration/testing of distributed systems
- Extensibility, Compose-ability and re-use of algorithms
- Multiple levels (layers) of (non-)faulty operation
- Visualization of schedules/traces (understanding aids)
- Ultimately, Synthesis of dependable distributed systems

More advantages

- **One front-end**
  - All that framework taken care of (for all developers)
  - No fluctuation: a model/proof vs. implementation
  - Implementation is its own model
    - no more separate model/proof activities.

Extra Features

- Snapshot/Resume (to rewind, try other schedules)
  - Full runtime capture
  - Traces untouched (keeping exploration history)
- Tracing Builtin (FULL state capture)
  - For Scheduler: debugging aid
  - For Distributed System: Analysis and Visualization
  - Visualizer/stepper being built!

Limitations

Limitations

- Programming-Language specific
  - Current implementation => specific to Scala
    - Targeting Akka first (checking + synthesis)
  - Infrastructure ported
  - Schedulers ported
  - front-end(s) re-written

Teaser (What if – one rule takes care of code)

One rule - rules them all

replicated[main][s1,s2][primary](d).on(3 updates)

One rule - rules them all

replicated[main][s1,s2][primary](d).on(3 updates)

```
d = 0 // data item
cd = 0 // count of updates to 'd'
vd = 0 // version ID of 'd'
csd = d.hashCode() // check–sum of 'd'
replicatedOn = {d: [s1,s2],...}
alive–agents = [s1,s2]
```

One rule - rules them all

replicated[main][s1,s2][primary](d).on(3 updates)

```
d = 0 // data item
cd = 0 // count of updates to 'd'
vd = 0 // version ID of 'd'
csd = d.hashCode() // check-sum of 'd'
replicatedOn = {d: [s1,s2],...}
alive-agents = [s1,s2]
```

```
cd++; vd++; csd += d.hashCode()
if (cd%3 == 0) {
  m = Message("Replicate", payload = [d, vd]); ds.send(main, m, s1);
  ds.send(main, m, s2)
}
```

One rule - rules them all

replicated[main][s1,s2][primary](d).on(3 updates)

```
d = 0 // data item
cd = 0 // count of updates
vd = 0 // version ID of 'd'
csd = d.hashCode() // ch
replicatedOn = {d: [s1,s2
alive–agents = [s1,s2]
```

```
cd++;
```

```
// 'd ' was updated ; recvr needs to catchup
if (m.payload(3) > recvr.vd)
  // just one batch update happened
  if(recMsg.payload(2) – recvr.vd ==3)
     update(recvr.locals , recMsg)
  // > 1 batch update , recvr missed >= 1 update
  else if (recMsg.payload(2) – recvr.vd >3 )
     updateElaborated ( recvr , recMsg )
  // recvr ahead, let other's know
  else if (recMsg.payload(2) – recvr.vd < 0 )
  { m = Message("Replicate", payload = [d,vd, csd]);
    replicateTo(replicatedOn, m)}
else // more sophisticated fault–tolerance work
somethingIsWrong (m) / / use checksum+others (raft)
```

```
); ds.send(main, m, s1);
```

Architecture+Lang. Design

Communication Patterns & Events

- **Send (communication)**
  - Fire and forget message send
- **Ask (communication+synchronization)**
  - Fire and return handle to (optionally) block on later/immediately
  - Handle is a (Future) object.
- **LOCK/UNLOCK (event)**
  - model network partition
- **Primitives differ** from parallel programming (list on next slide)

DS2 - Kinds of Events

$$\mathcal{K} \in \{none, send, ask, resolve, create, start, stop, kill, lock, unlock, stop-consume, resume-consume, become, unbecome, stash, unstash, unstash-all, get, get-timed, bootstrap, bootstrap-all, modify-state\}$$

$\mathcal{A}$ set of all agents

$\mathcal{M}$ message type

$\mathcal{B}$ basic block of code (to execute)

$\mathcal{C} \in \mathcal{M} \times \mathcal{A} \rightarrow \mathcal{K} \times \mathcal{B}$

statement type (plus hidden meta data)

we need ONE model representing ALL

Process (shared mem.)

we need ONE model representing ALL

Threads (shared mem.)

Process (shared mem.)

we need ONE model representing ALL

Process (shared mem.)

we need ONE model representing ALL

Threads (shared mem.)

What more?! PL's Mem. Models

Process (shared mem.)

we need ONE model representing ALL

Threads (shared mem.)

What more?!
PL's Mem. Models

Actors
(No Shared mem. + comm.)

Process (shared mem.)

we need ONE model representing ALL

Threads (shared mem.)

What more?!
PL's Mem. Models

Actors
(No Shared mem. + comm.)

MPI Process
(shared mem. + Comm.)

Process (shared mem.)

we need ONE model representing ALL

Threads (shared mem.)

What more?!
PL's Mem. Models

Actors
(No Shared mem. + comm.)

MPI Process
(shared mem. + Comm.)

Event-Driven Threads
(shared mem. + Events)

Process (shared mem.)

we need ONE model representing ALL

Threads (shared mem.)

What more?! PL's Mem. Models

MPI Process (shared mem. + Comm.)

Actors (No Shared mem. + comm.)

Event-Driven Threads (shared mem. + Events)

Actors (Some with Shared mem. + comm.)

Process (shared mem.)

we need ONE model representing ALL

Threads (shared mem.)

What more?!
PL's Mem. Models

MPI Process
(shared mem. + Comm.)

Actors
(No Shared mem. + comm.)

Event-Driven Threads
(shared mem. + Events)

Actors
(Some with Shared mem. + comm.)

Replicated State Machines
(shared mem. + Events + Transitions)

Threads (shared m

MPI Process
(shared mem. + Co

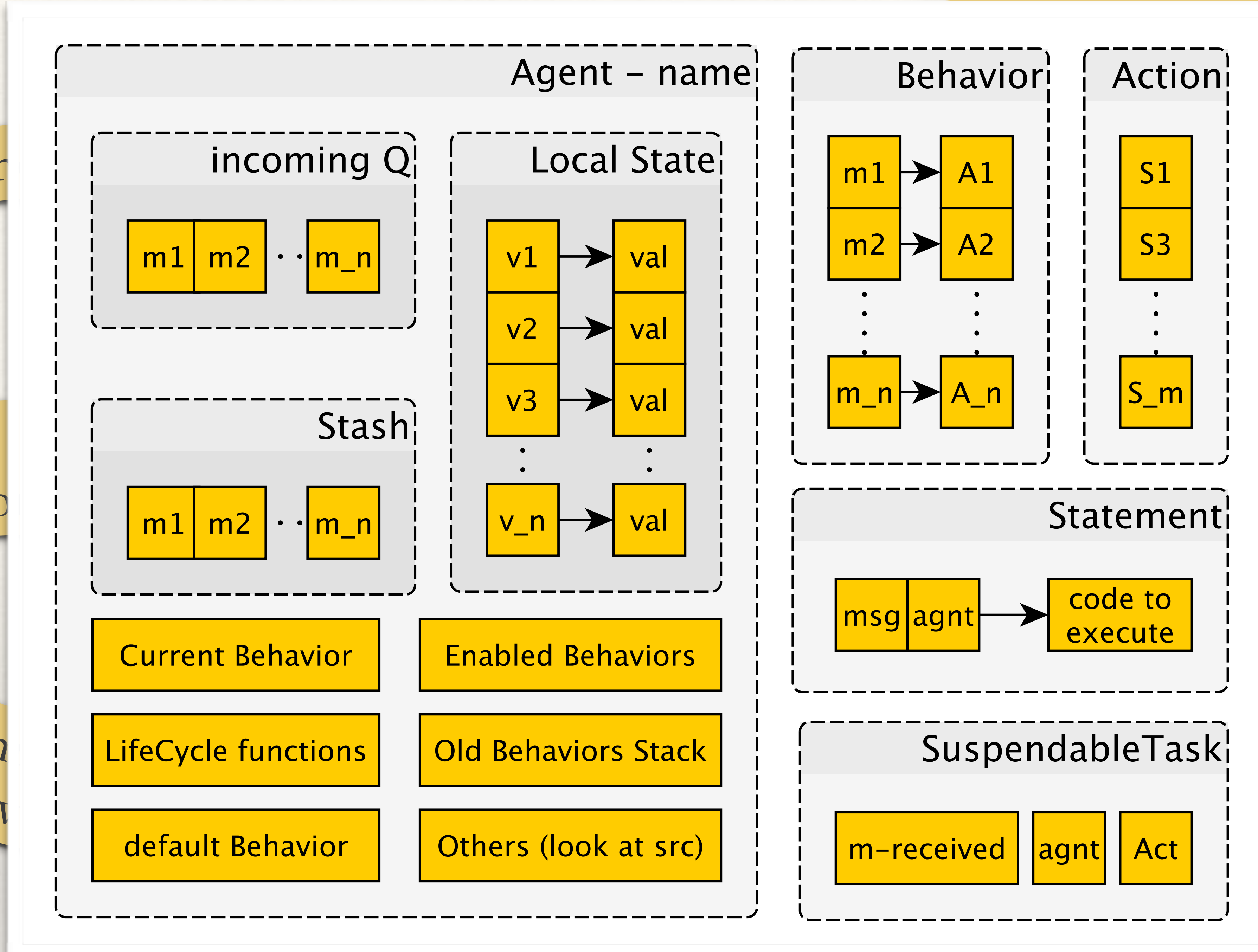
Event-Driven Th
(shared mem. + Ev

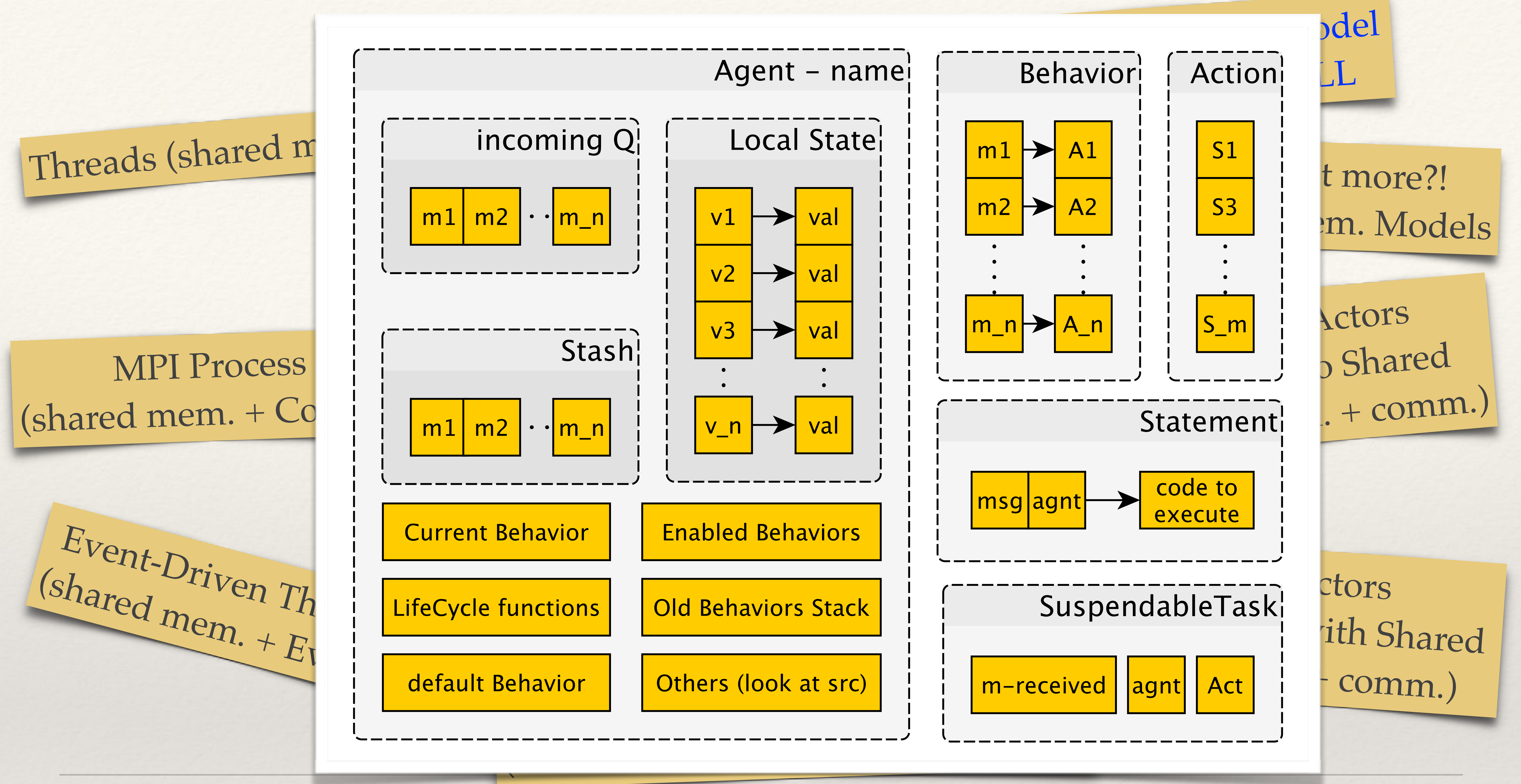
odel
LL

t more?!
em. Models

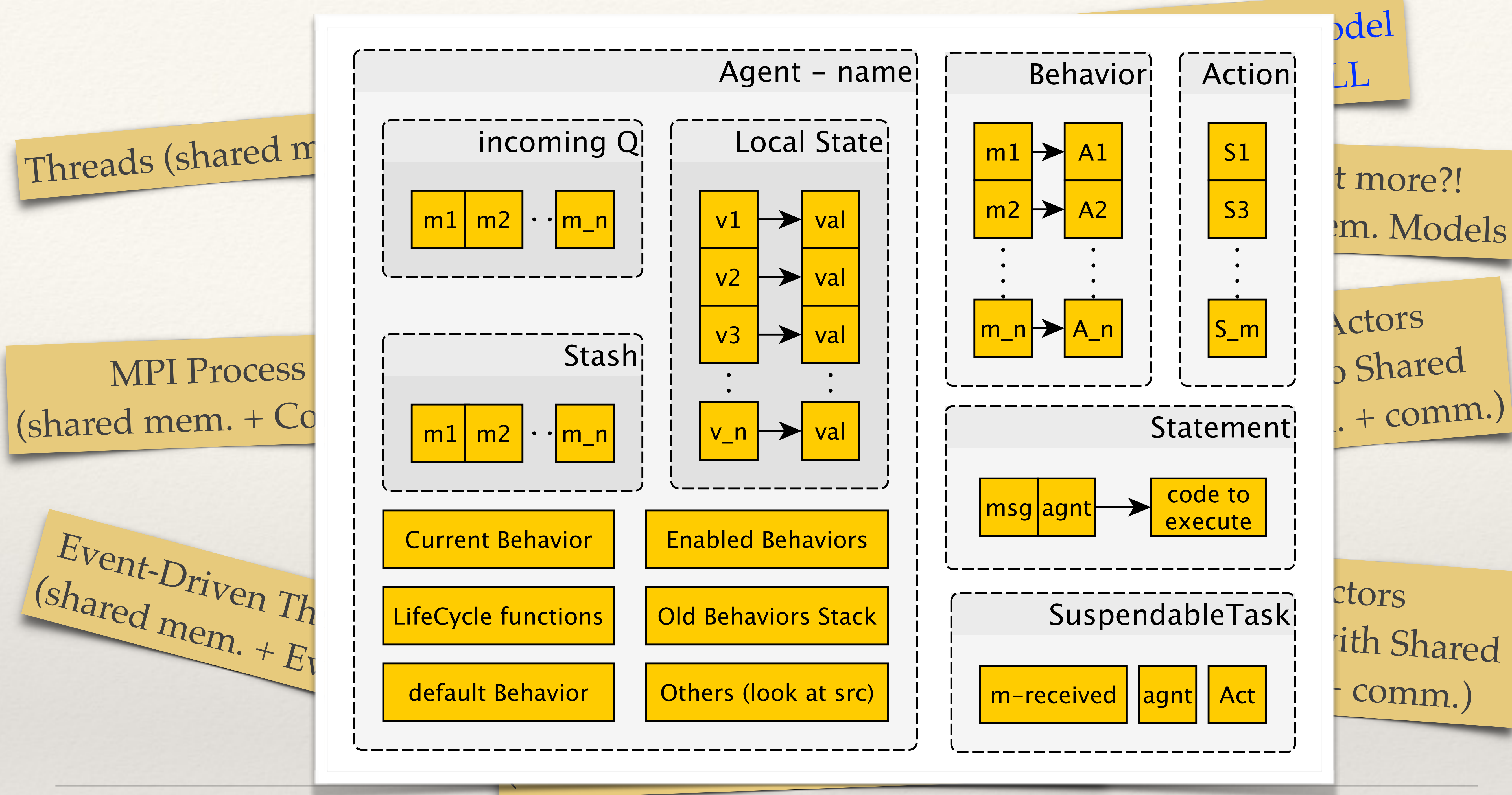
Actors
o Shared
. + comm.)

ctors
ith Shared
- comm.)



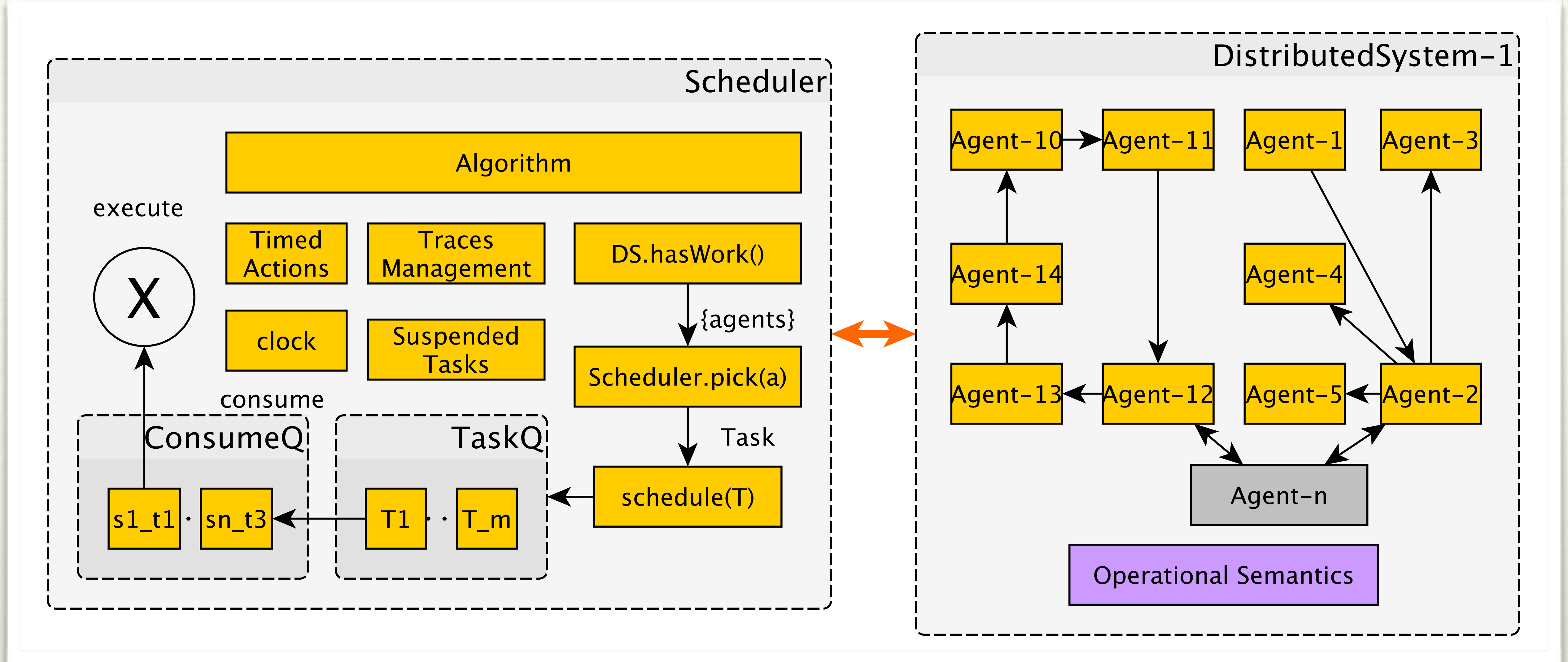


DS2 Architecture- an Agent



DS2 Architecture- an Agent

A single process model with: Self contained state, communication, Behaviors,
other helper functions.
Accommodating all kinds of processes.



DS2 Architecture - A Strategy on a Context

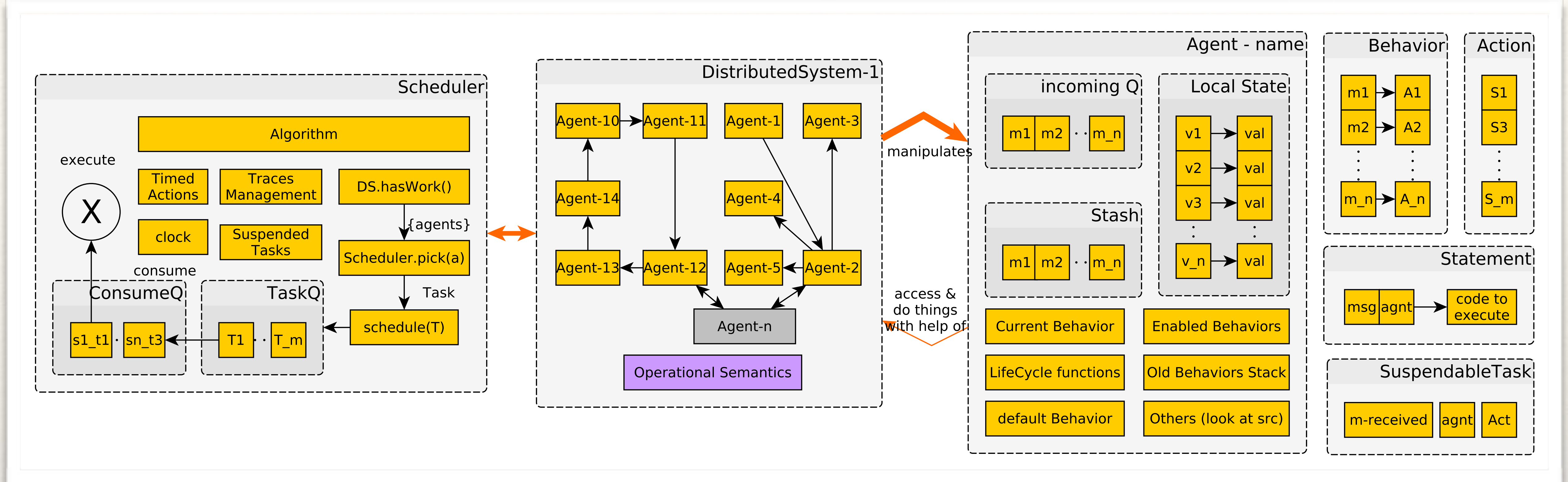
Scheduler+DistributedSystem

Strategy OO Design Pattern

Scheduler = Strategy

Dist. Sys = Context

Simple, extensible, effective separation of concerns



DS2 Architecture - Semantic-aware scheduling

Inter-related entities in a

Strategy OO Design Pattern

Scheduler = Strategy

Dist. Sys = Context

Simple, extensible, effective separation of concerns

Example driven benefit illustration (Animated from FMI paper)

High level example

Echo Server-client interaction:

1. Server => started (bootstrapped) => unlocked
2. Client => started => unlocked => send request => waits confirmation
3. Server => process request => sends confirmation
4. Client => is happy

Scenarios:

- No bugs schedule (above)
- Deadlock 1
- Deadlock 2

Example

```
val ds = new DistributedSystem("Echo-ack")

val s = new Agent("Server")

val c = new Agent("Client")

val act1, act2, act3 = new Action

// Client setup

act1 + Statement(UNLOCK,c) // unlocks the agent incoming q

act1 + Statement(ASK,c,new Message("Show","Hello!"),s, "vn")

act1 + Statement(GET,c,"vn","vn2")

act1 + Statement(println("I'm Happy!"))

c.R("Start") = act1 // (Start, act1) to reactions map

// Server setup

act2 + Statement(UNLOCK, s)

act2 + Statement(println("Greetings!"))

act3 + Statement((m:Message,a:Agent)=>println(m.p))

act3 + Statement((m:Message,a:Agent)=>send(s,m(p = true),m.s))

s.R("Start") = act2 ; s.R("Show") = act3

ds += Set(s,c) // adding agents to system

ds.attach(BasicScheduler)
```

Correct Schedule

```
val sch = ds.scheduler
sch.boot(s); sch.boot(c) // sends Start msg to s and to c
sch.schedule(s) // schedule start-task from s
sch.schedule(c) // schedule start-task from c
sch.consume(s)  // consume UNLOCK stmt from s-task
sch.consume(s)  // consume "greeting" stmt from s-task
sch.consume(c)  // consume UNLOCK stmt from c-task
sch.consume(c)  // consume ASK stmt from c-task
sch.executeOne  // UNLOCK s-stmt, IsLocked(s) == false
sch.executeOne  // "greeting" s-stmt
sch.executeOne  // UNLOCK c-stmt, IsLocked(c) == false
sch.executeOne  // ASK s-stmt, T = {t} temporary agent
            // and s.q == [Show("Hello",s=t)]
sch.schedule(s) // schedule "Show" task from s
sch.consume(s)  // consume print("Hello") stmt
sch.consume(c)  // consume GET stmt from c-task
sch.consume(s)  // consume resolving send(..) stmt
            // note GET blocks, then it is resolved
sch.consume(c)  // consume "happy" stmt from c-task
sch.executeOne  // s print("Hello")
sch.executeOne  // c blocks on GET, doesn't progress
            // putting back all stmts after it
            // from cq back to front of task.xq in order
sch.executeOne  // resolving send(..), t.q != empty
            // things happen to t.L("vn")-future resolved
            // and then c.q = [RF(f,s=s)], note sender
            // is s, not t
sch.handel(c)   // handling the RF message, unblocking c
sch.consume(c)  // consuming GET from c again
sch.consume(c)  // consuming "happy" stmt from c
sch.executeOne  // R-GET c-stmt, won't block (resolved)
            // c.L("vn2") = c.L("vn").val
sch.executeOne  // print("I'm happy")
// DONE happy schedule, other schedules are not this happy
```

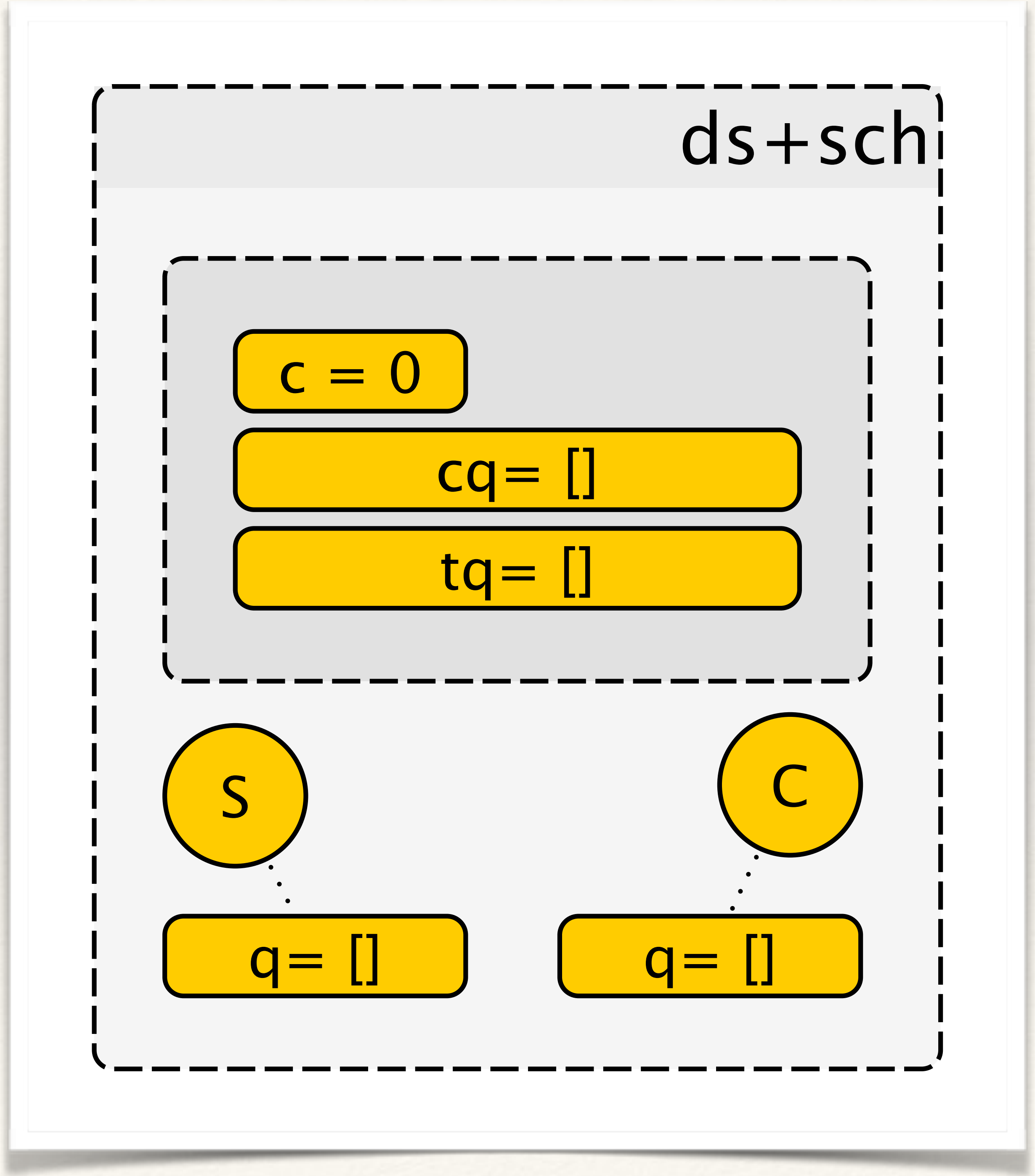
animated schedule

Initial state (nothing executed)

To Execute:

sch.boot(s)

sch.boot(c)



animated schedule

Executed:

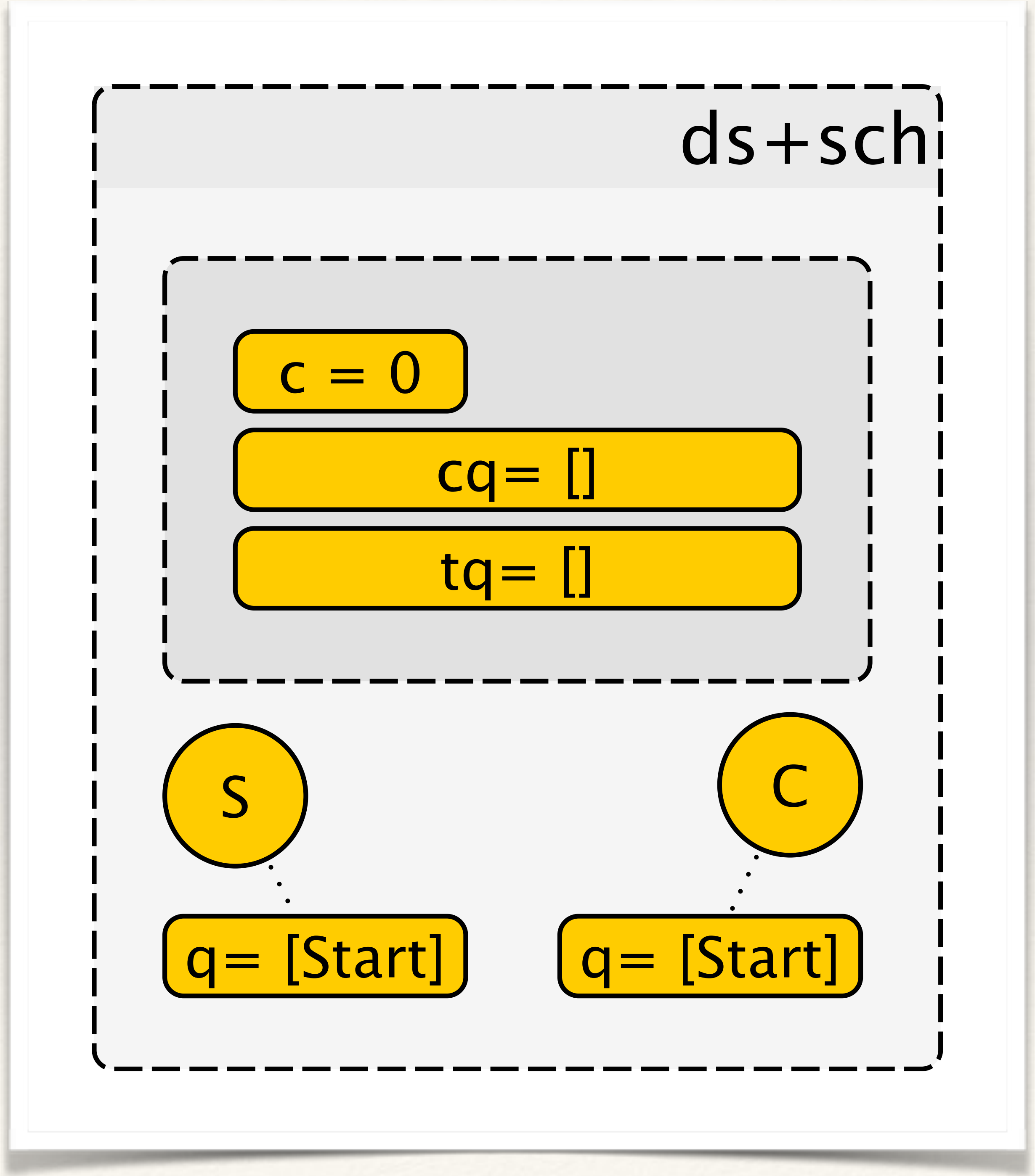
sch.boot(s)

sch.boot(c)

To Execute:

sch.schedule(s)

sch.schedule(c)



animated schedule

Executed:

sch.schedule(s)

sch.schedule(c)

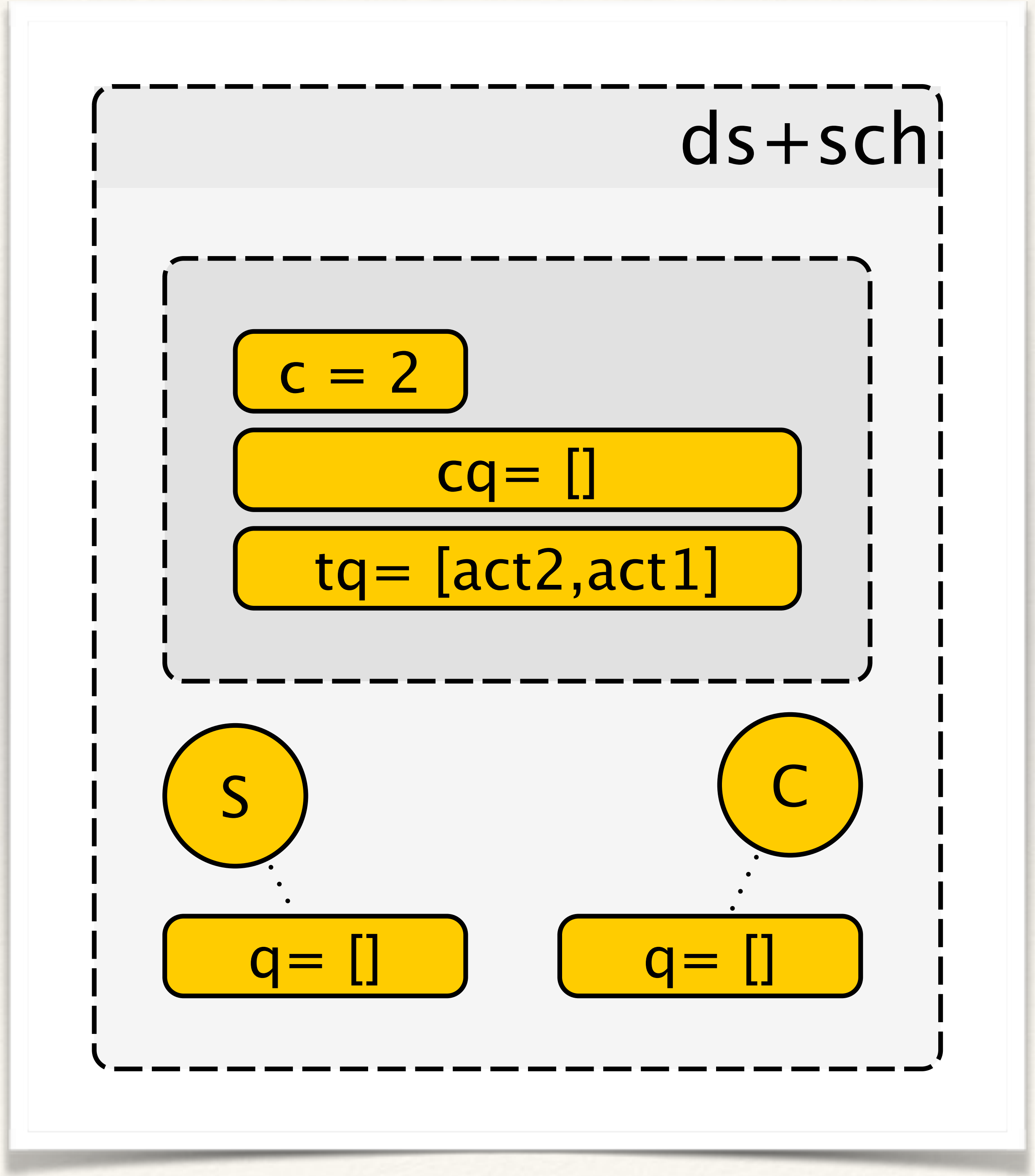
To Execute:

sch.consume(s)

sch.consume(s)

sch.consume(c)

sch.consume(c)



animated schedule

Executed:

sch.consume(s)

sch.consume(s)

sch.consume(c)

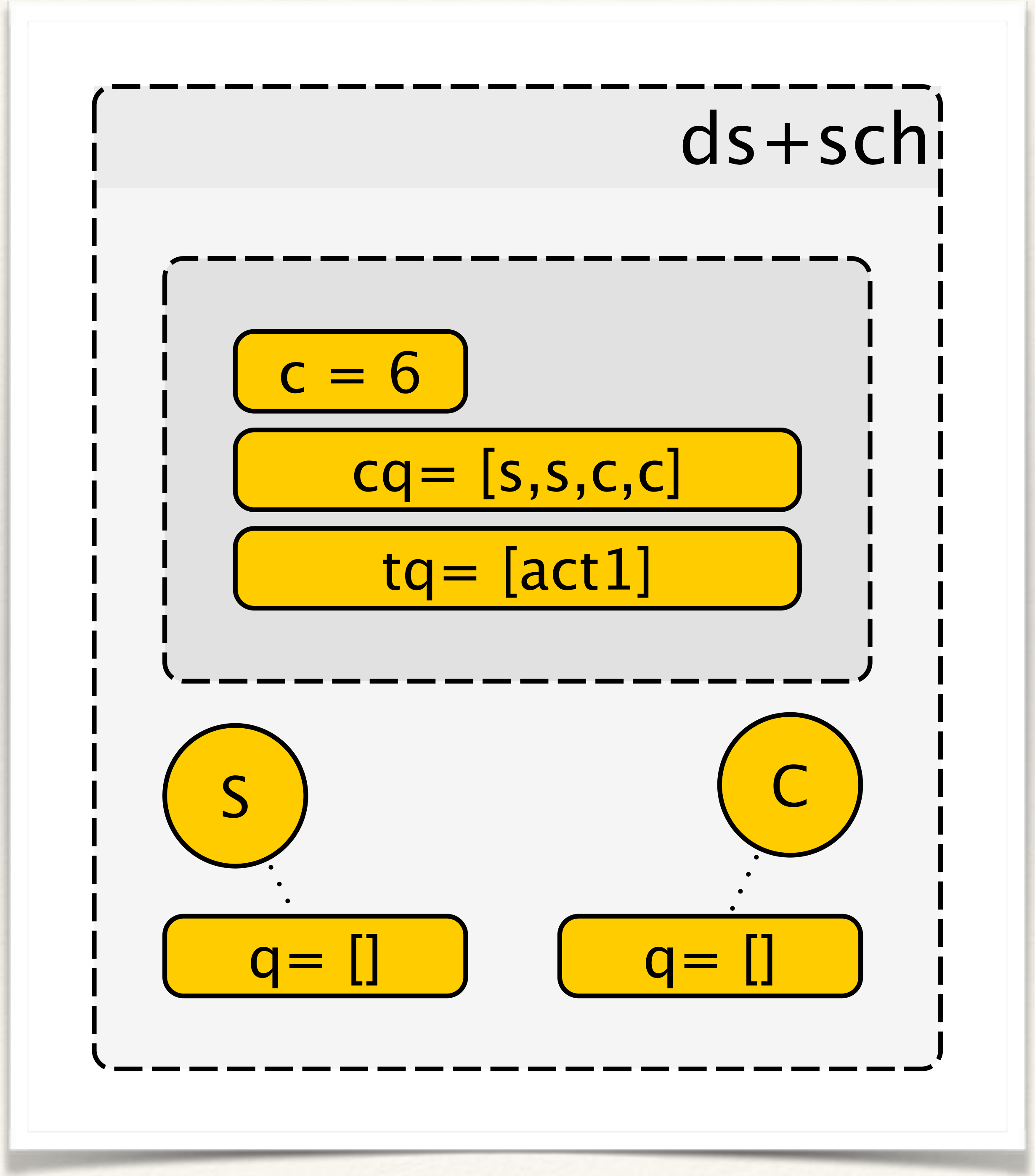
sch.consume(c)

To Execute:

sch.executeOne

sch.executeOne

sch.executeOne



animated schedule

Executed:

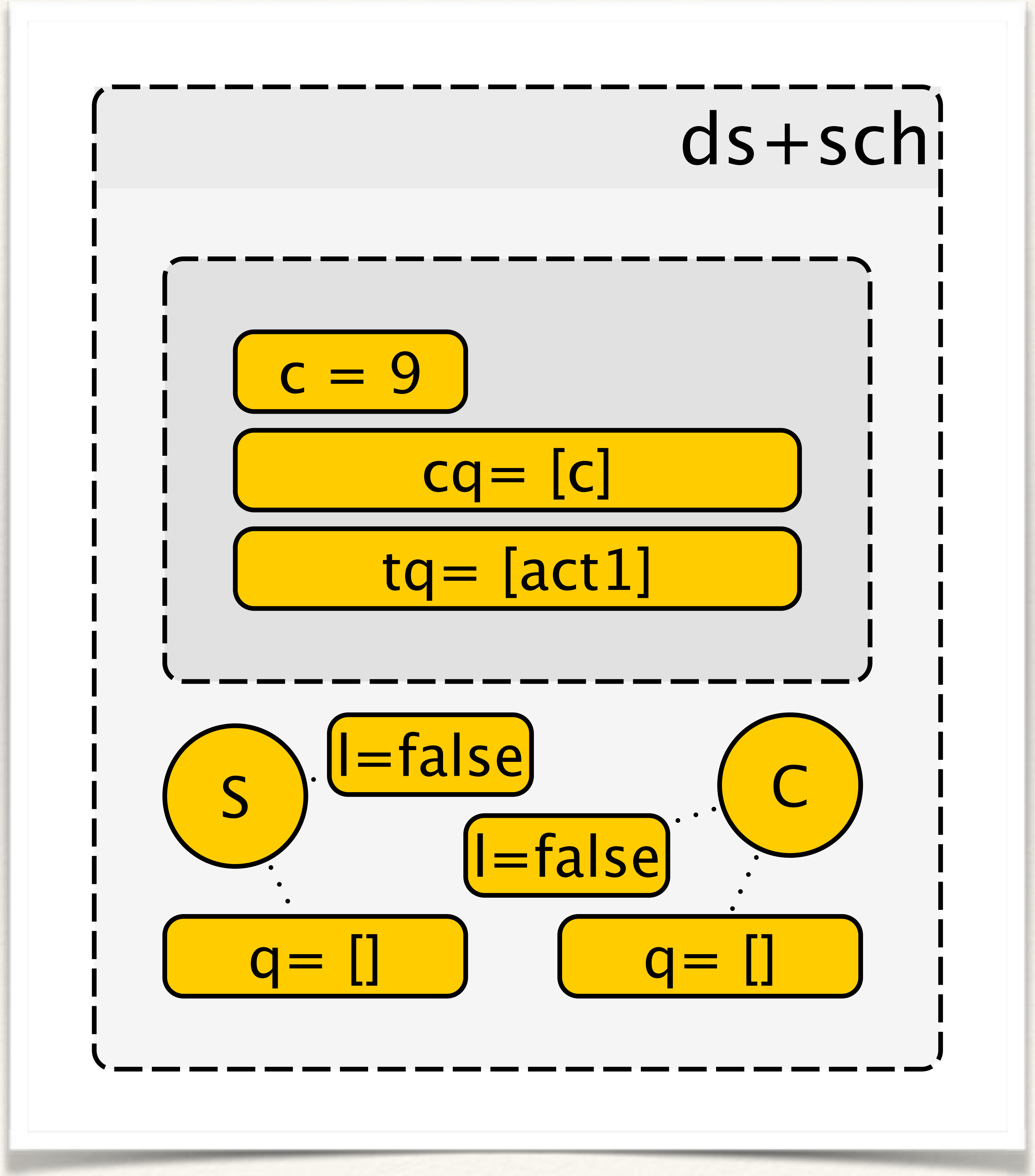
sch.executeOne

sch.executeOne

sch.executeOne

To Execute:

sch.executeOne // ask stmt



animated schedule

<u>Executed:</u>

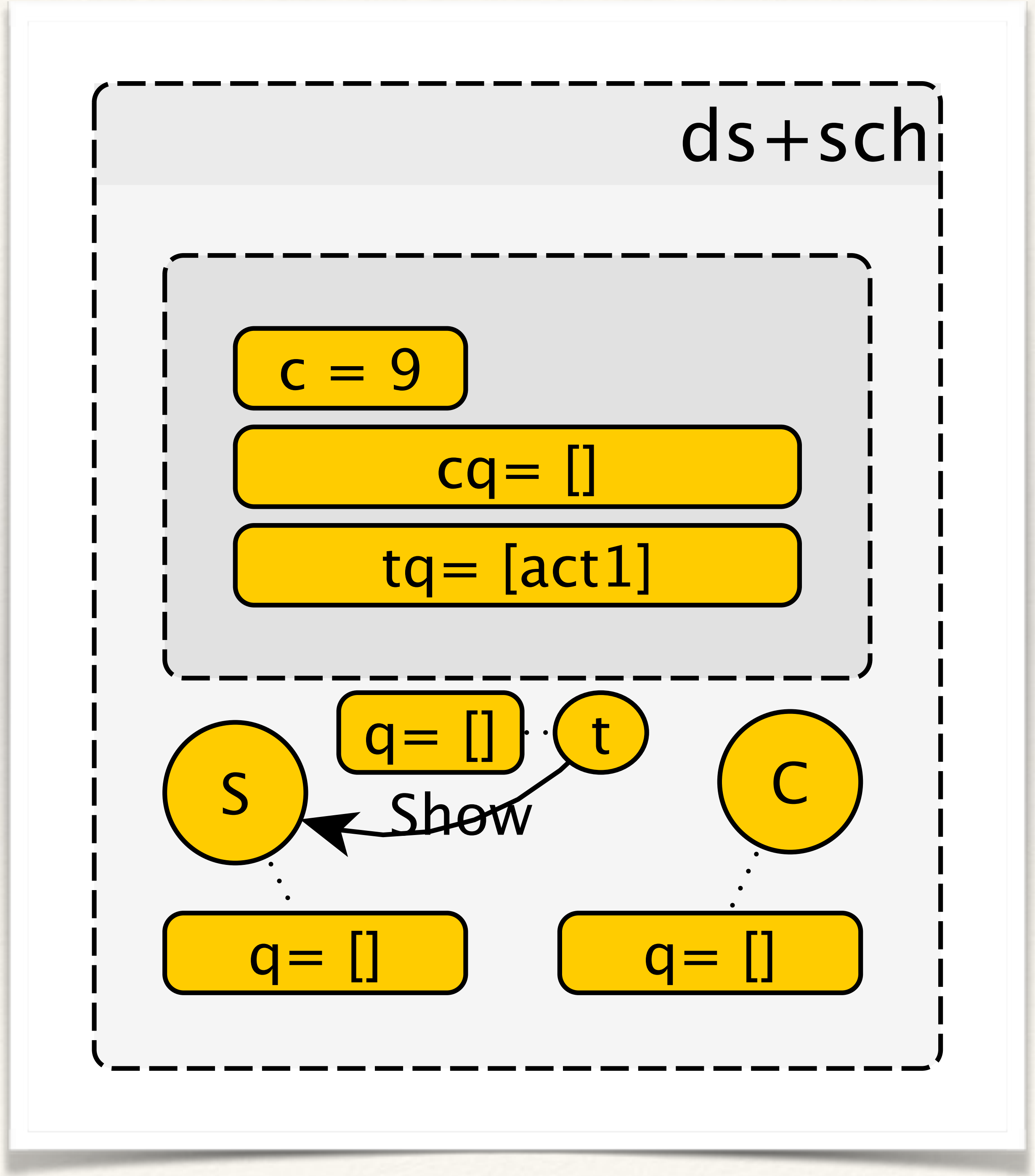
sch.executeOne

sch.executeOne

sch.executeOne

<u>To Execute:</u>

sch.executeOne / / ask stmt



animated schedule

<u>**Executed:**</u>

sch.executeOne // ask stmt

<u>**To Execute:**</u>

sch.schedule(s) // "Show" task

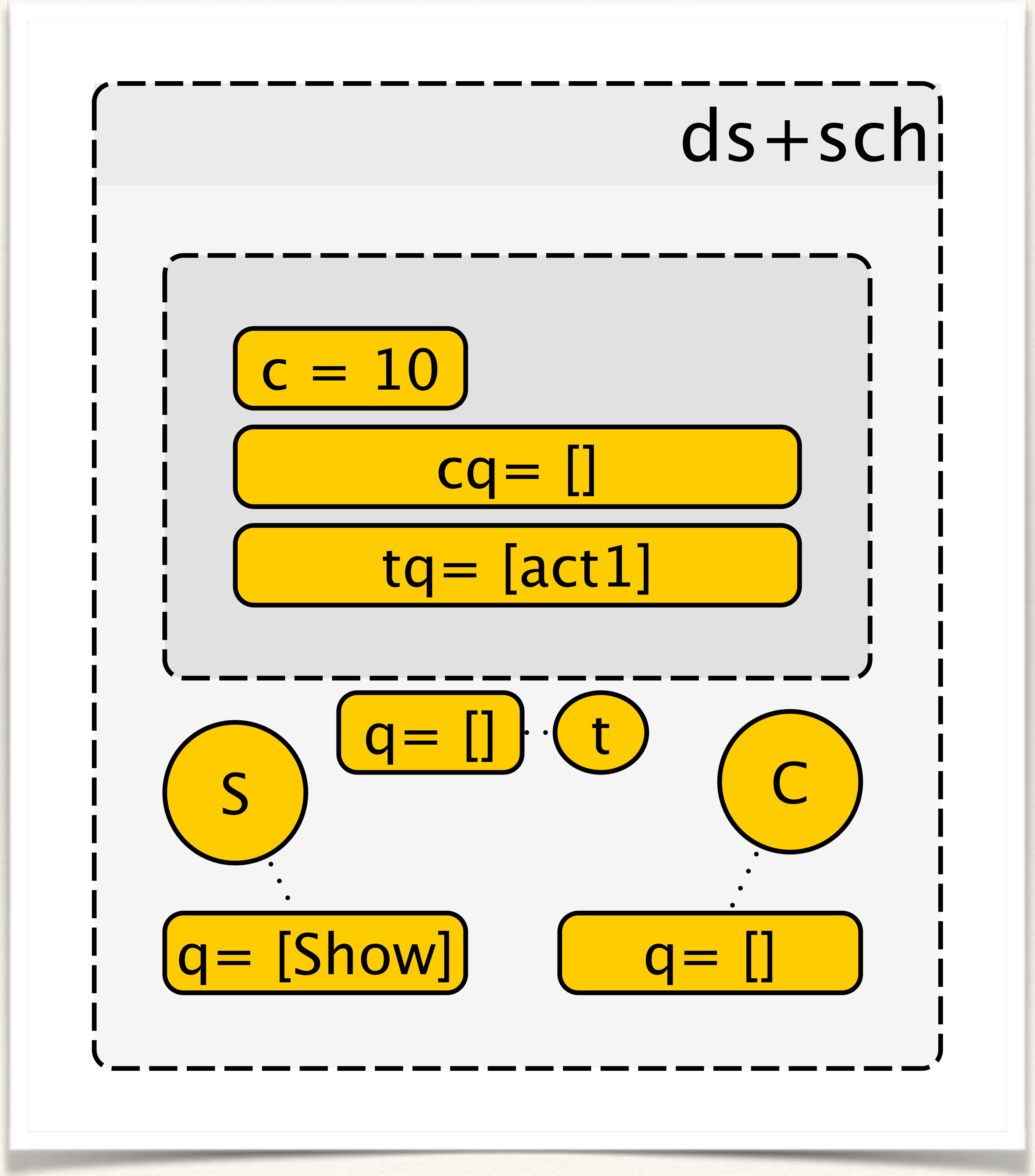
sch.consume(s) // print("Hello")

sch.consume(c) // consume GET

sch.consume(s) // consume r-send

// note GET blocks, then it is resolved

sch.consume(c) // consume "happy"



After some time …

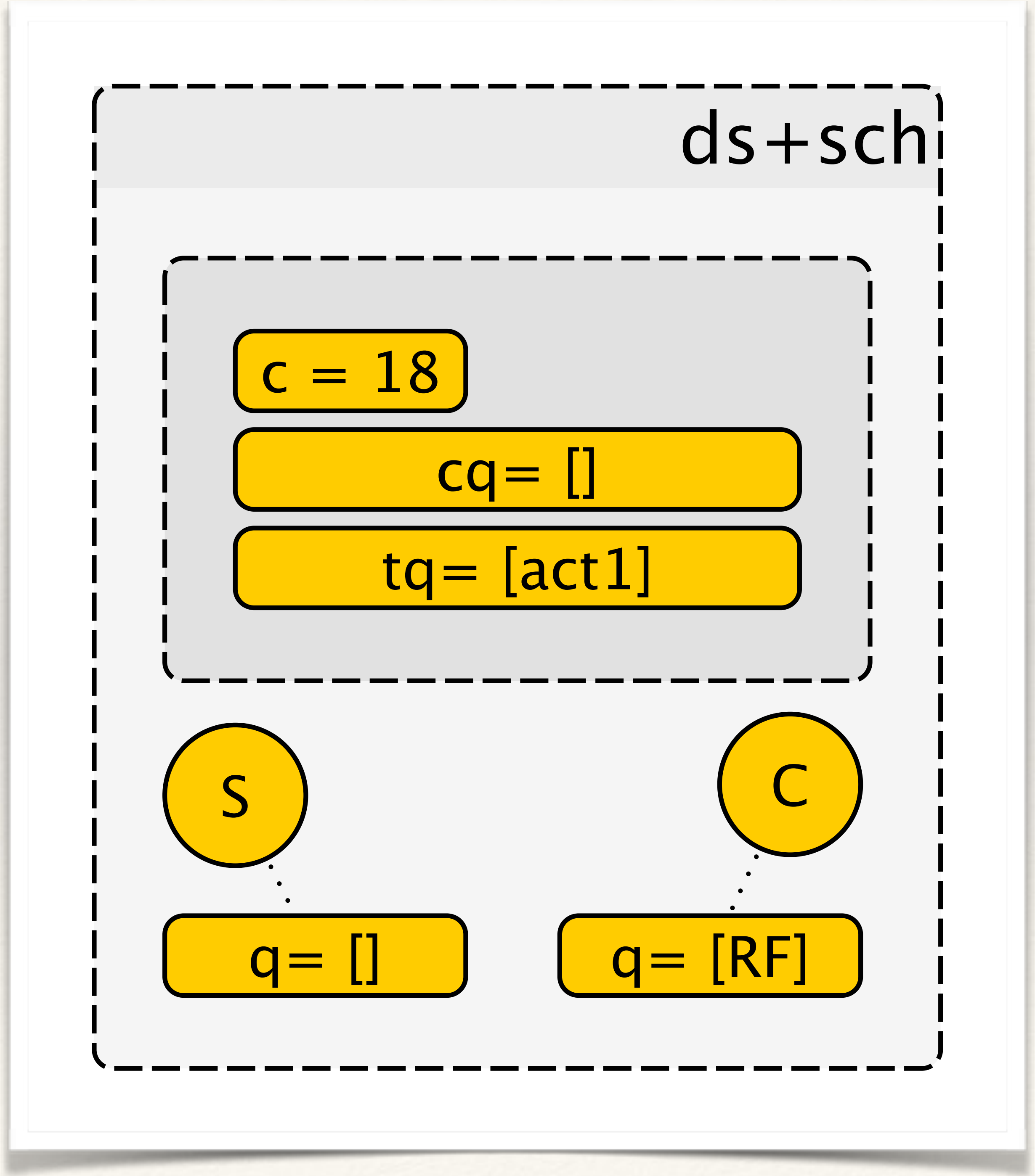
animated schedule

Executed:

sch.executeOne // r-send(..)

To Execute:

sch.handel(c) // RF



animated schedule

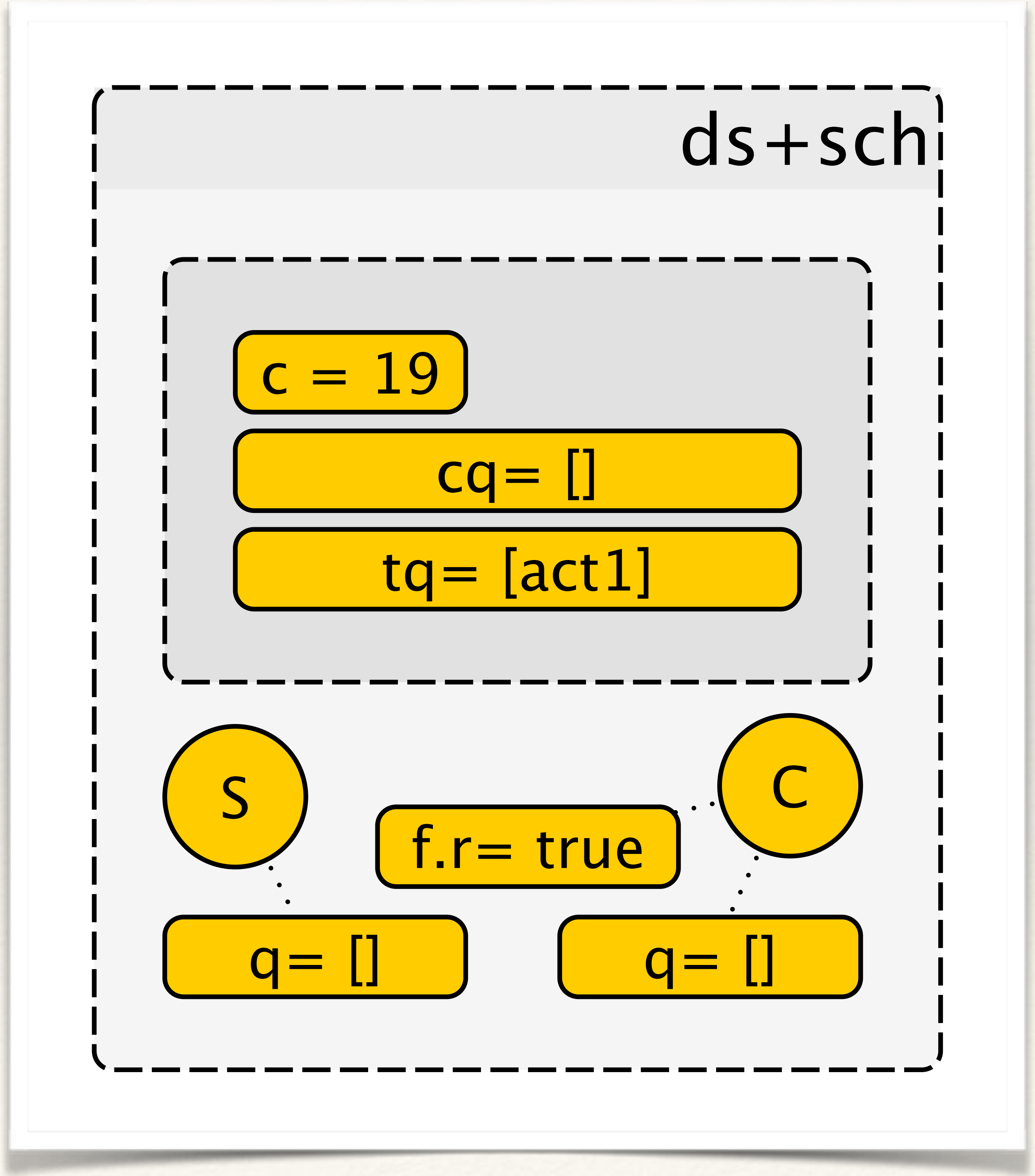
Executed:

sch.handel(c) / / RF

To Execute:

sch.consume(c) / / GET

sch.consume(c) / / "happy" stmt



animated schedule

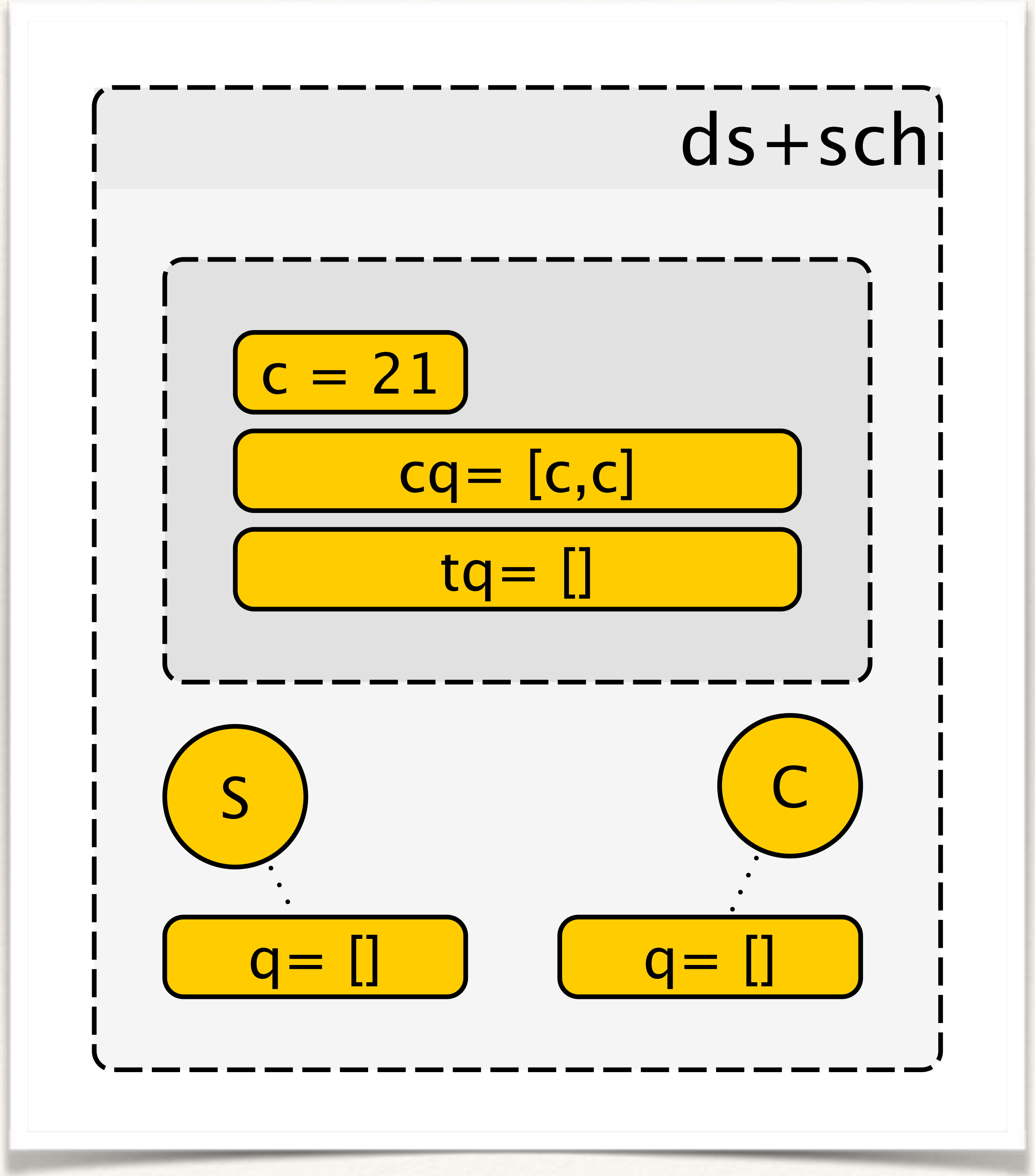
Executed:

sch.consume(c) // GET

sch.consume(c) // "happy" stmt

To Execute:

sch.executeOne // R-GET



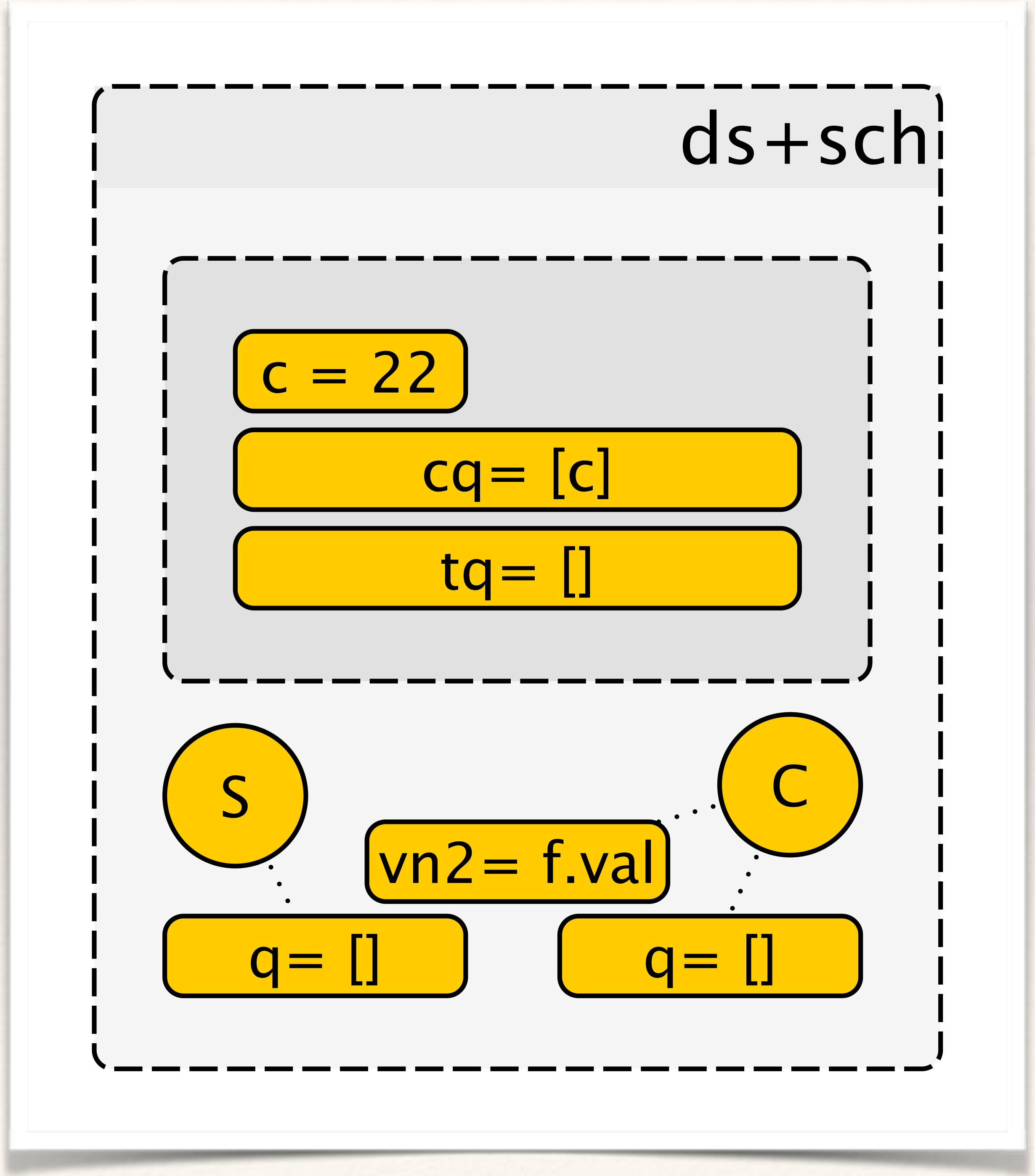
animated schedule

<u>**Executed:**</u>

sch.executeOne // R-GET

<u>**To Execute:**</u>

sch.executeOne //"I'm happy"

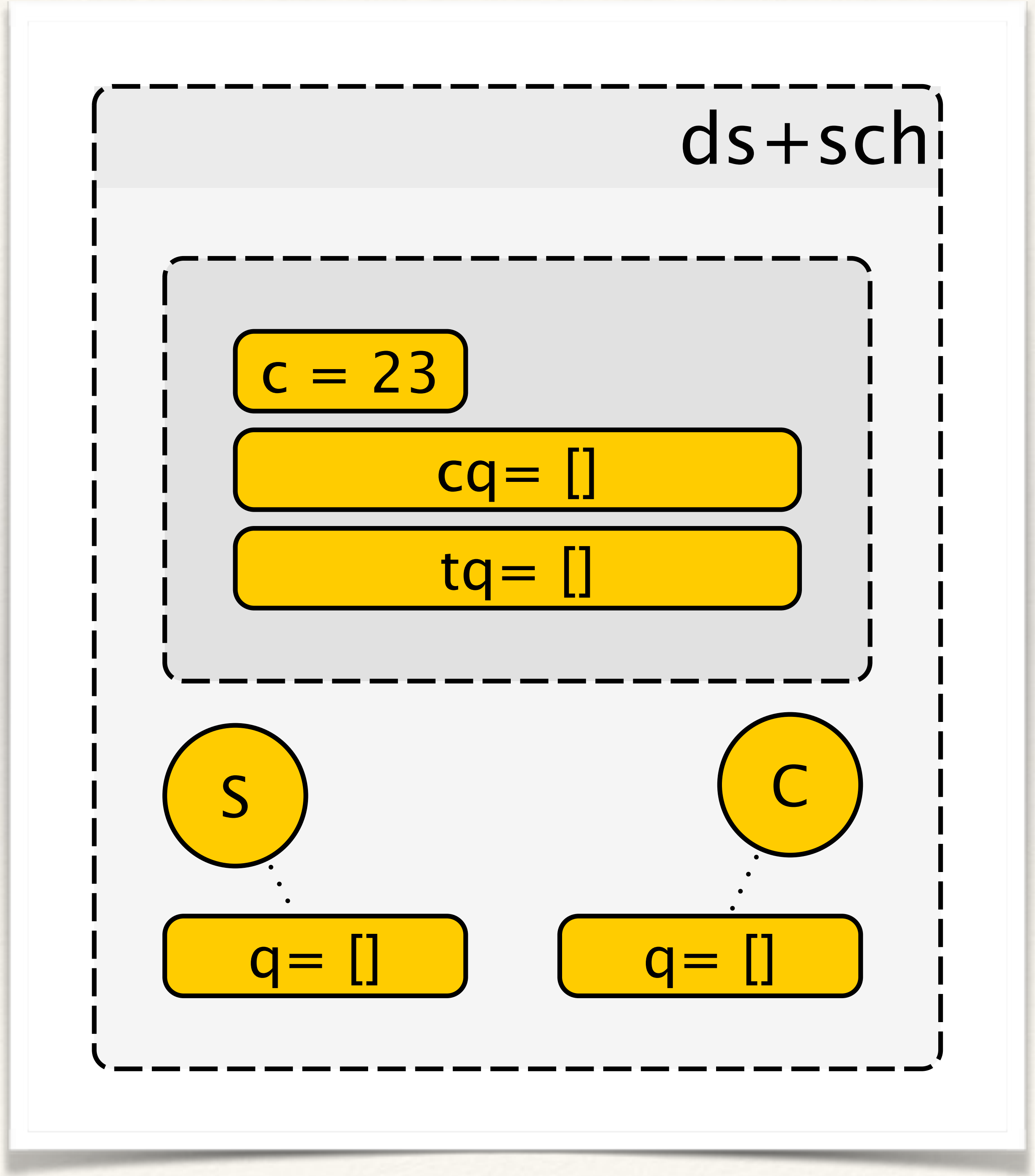


animated schedule

Executed:

sch.executeOne   //"I'm happy"

To Execute:



What could have gone wrong?

May Go Wrong

- Client could have blocked first
  - Before server resolves: it crashes => deadlock
  - After server resolves: RF dropped => deadlock
    - Messages in Agent’s queue are still *in-flight*
    - Till they are handled/stashed, then *delivered*
  - Both avoidable by *timed-get on future.*

Deadlock1 Schedule (dropped resolve future msg)

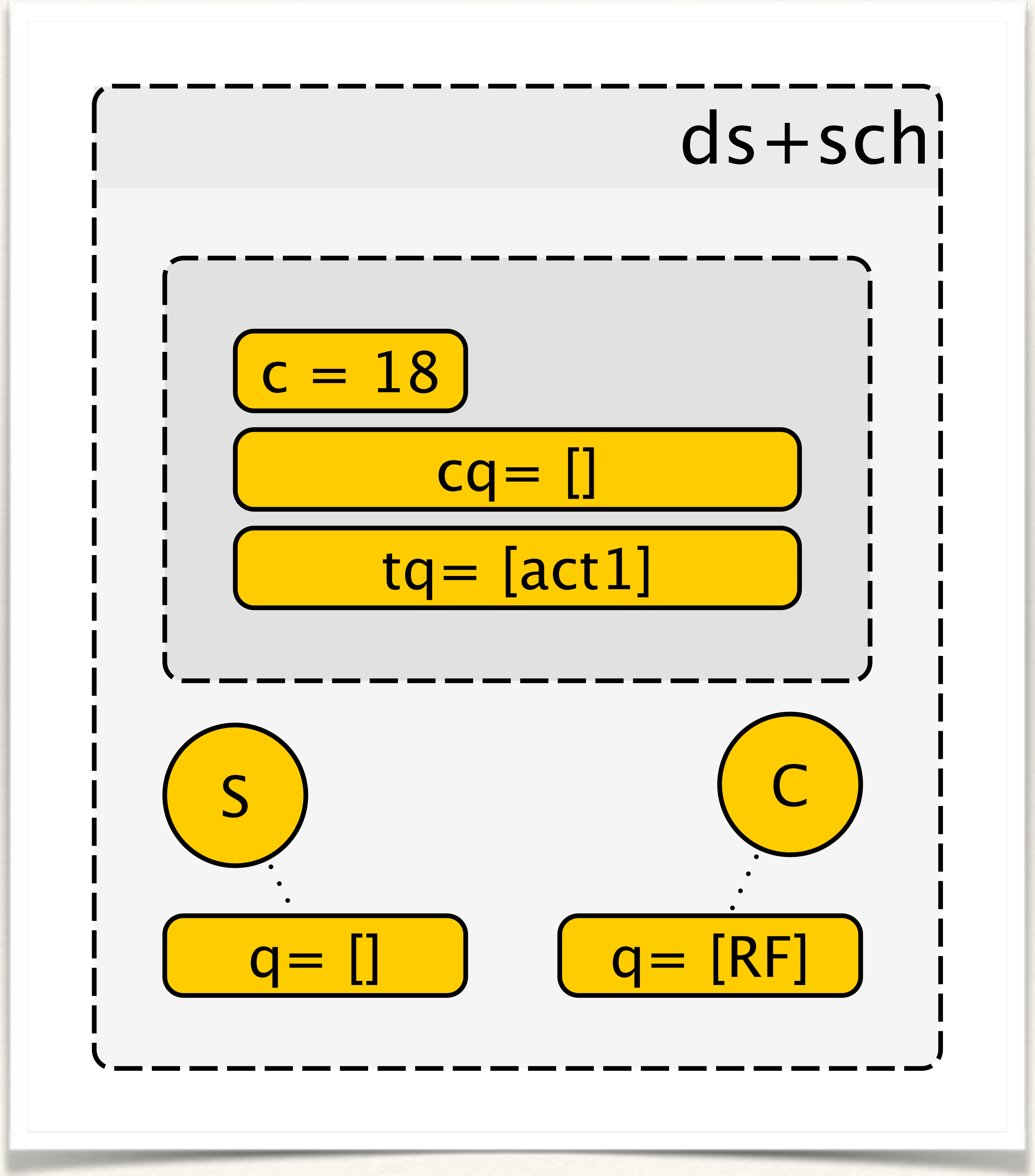
About to drop a message!

Executed:

sch.executeOne   // r-send(..)

To Execute:

sch.handel(c) // RF

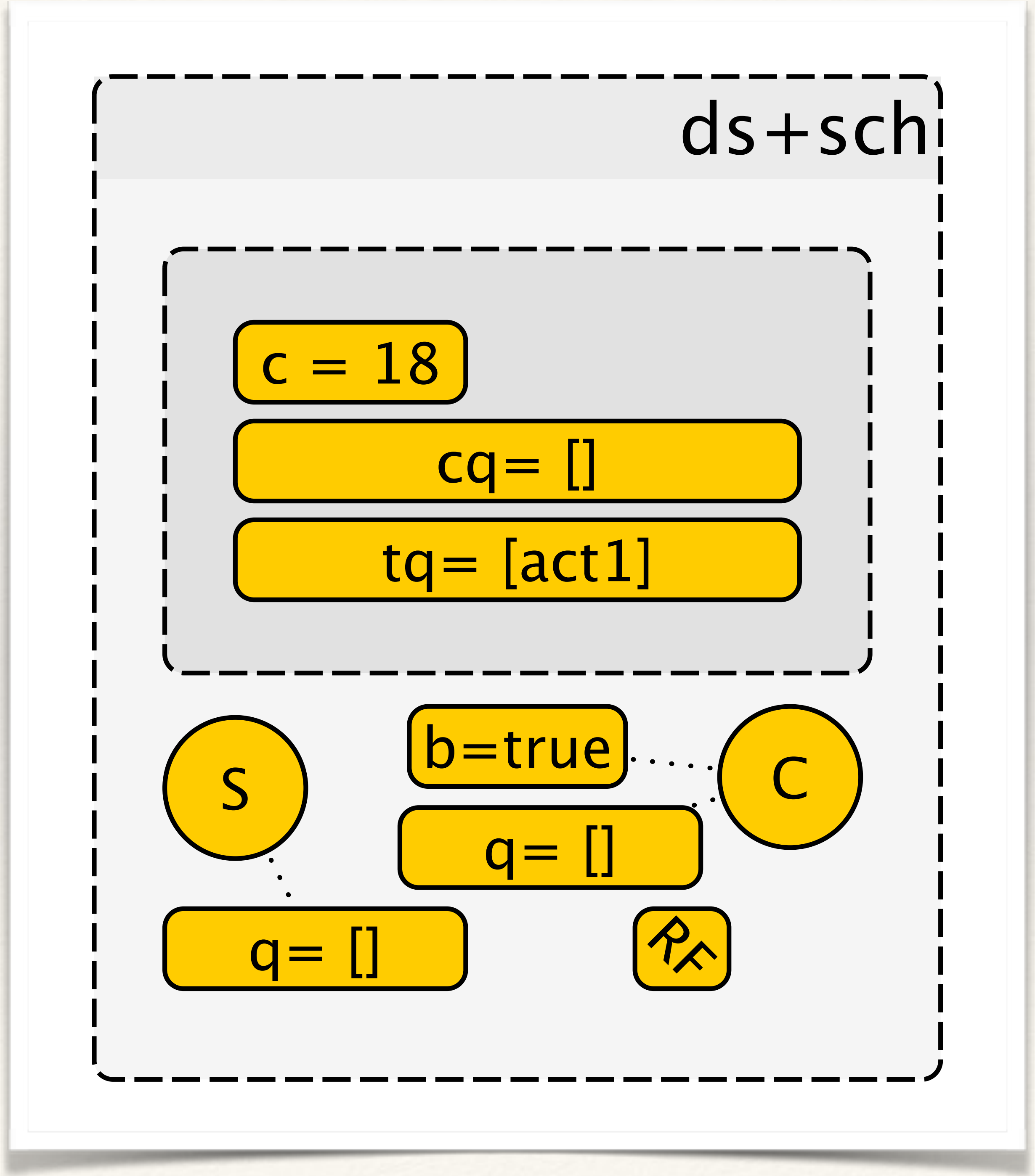


RF message dropped!

Executed:

simulated-RF-msg-drop

To Execute:



Deadlock2 Schedule (crashed server before resolve)

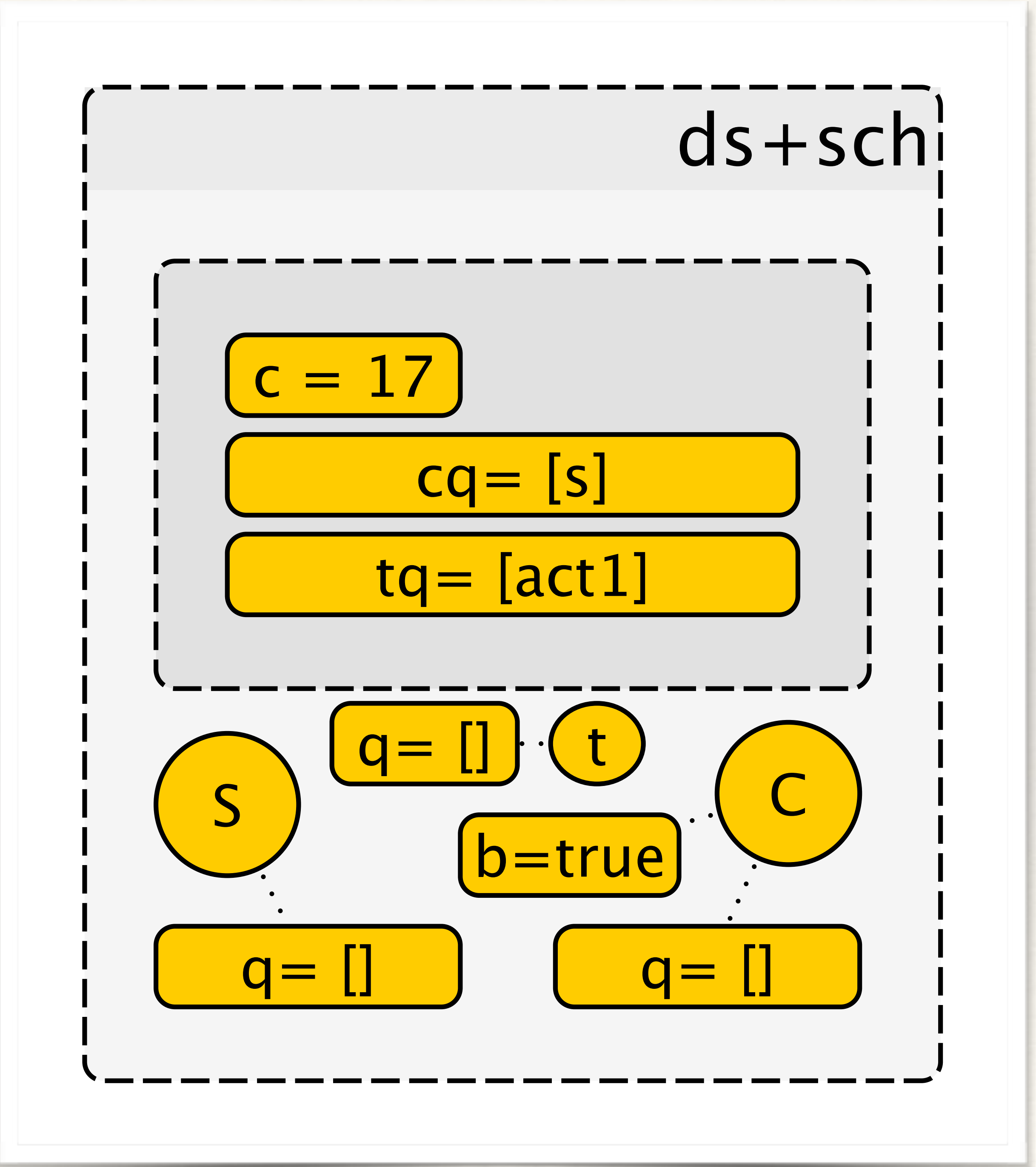
Client is blocked

Executed:

sch.executeOne // c blocks

To Execute:

sch.executeOne // r-send(..)



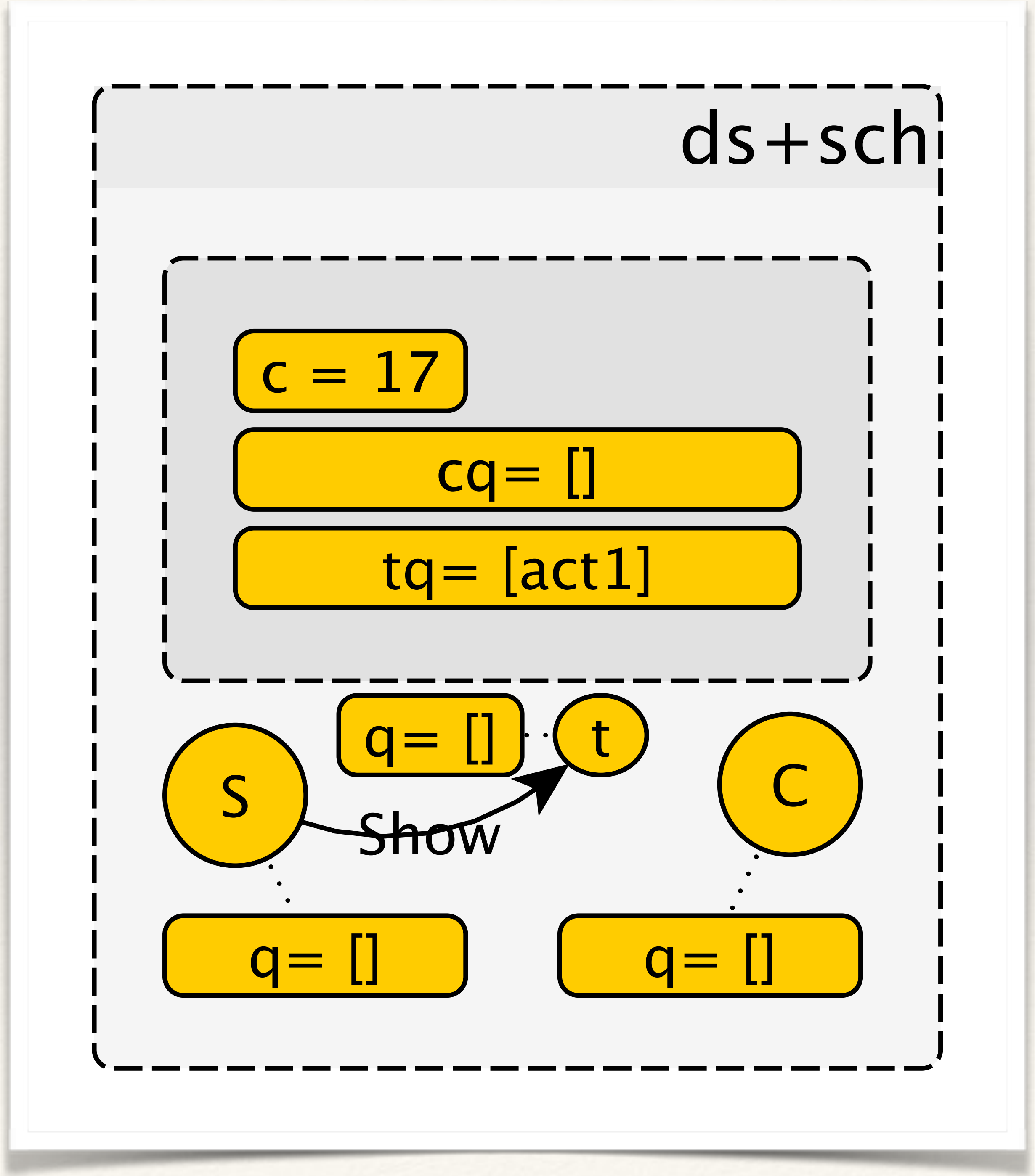
Server about to resolve but…

Executed:

sch.executeOne // c blocks

To Execute:

sch.executeOne // r-send(..)



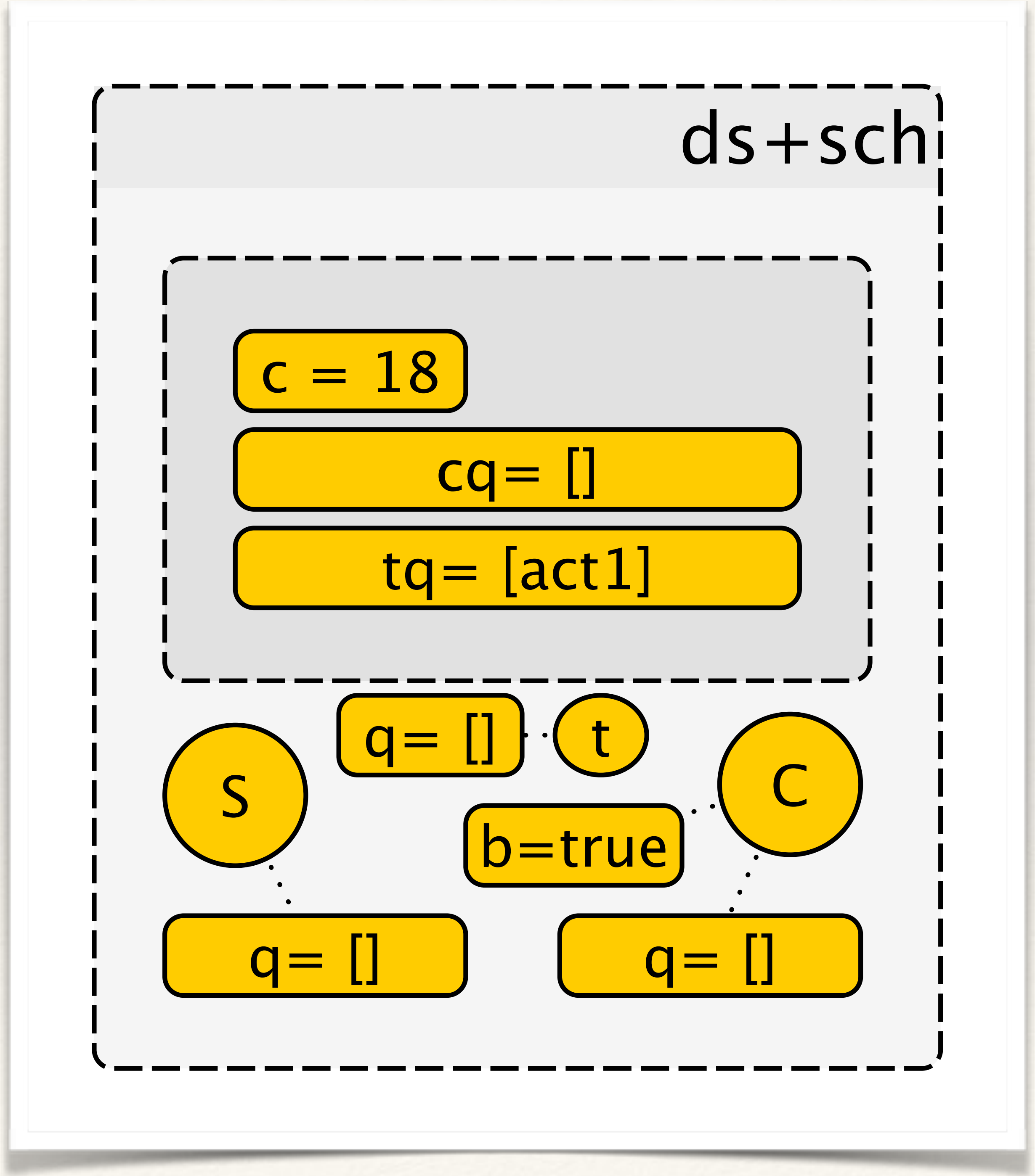
Server crashed before resolve …

Executed:

simalted-crash

server-came-back (empty hand)

To Execute:



That *simple* example taught us:
"more erroneous interleaving
than correct ones!"

Completion Status

Implementation/Completion Status

- DS2 model (shown here)
- Tracing
- Snapshot/Resume
- Basic Scheduler
- Chord, Zab, Multi-Paxos, Raft
- DS2 Lang. Spec.
- Visualization
- Akka front-end
- Linearizability Sch.
- DS2 Lang. impl.
- Synthesis

not started | started | partial completion / in progress | completed

Conclusion

Conclusion

- Motivated the need for an integrated solution
- Presented our model
- How it solves the issues stated
- Walk through example(s)
- Sneak peak towards synthesis
- Future work: Formal Operational Semantics (under review), Tool for Akka (with multiple alg.), Synthesis of Akka from DS2.

References

[1] ”Toward Rigorous Design of Domain-Specific Distributed Systems”, Mohammed S. Al-Mahfoudh, Ganesh Gopalakrishnan, Ryan Stutsman.

[2] http://formalverification.cs.utah.edu/ds2/

[3] ”Planning for Change in a Formal Verification of the Raft Consensus Protocol”, Doug Woos, Zachary Tatlock, James R. Wilcox, Michael D. Ernst, Steve Anton, Thomas Anderson.

Q/A

Thank you!

Removed frames follow

animated schedule

<u>Executed:</u>

sch.schedule(s) // "Show" task

sch.consume(s) // print("Hello")

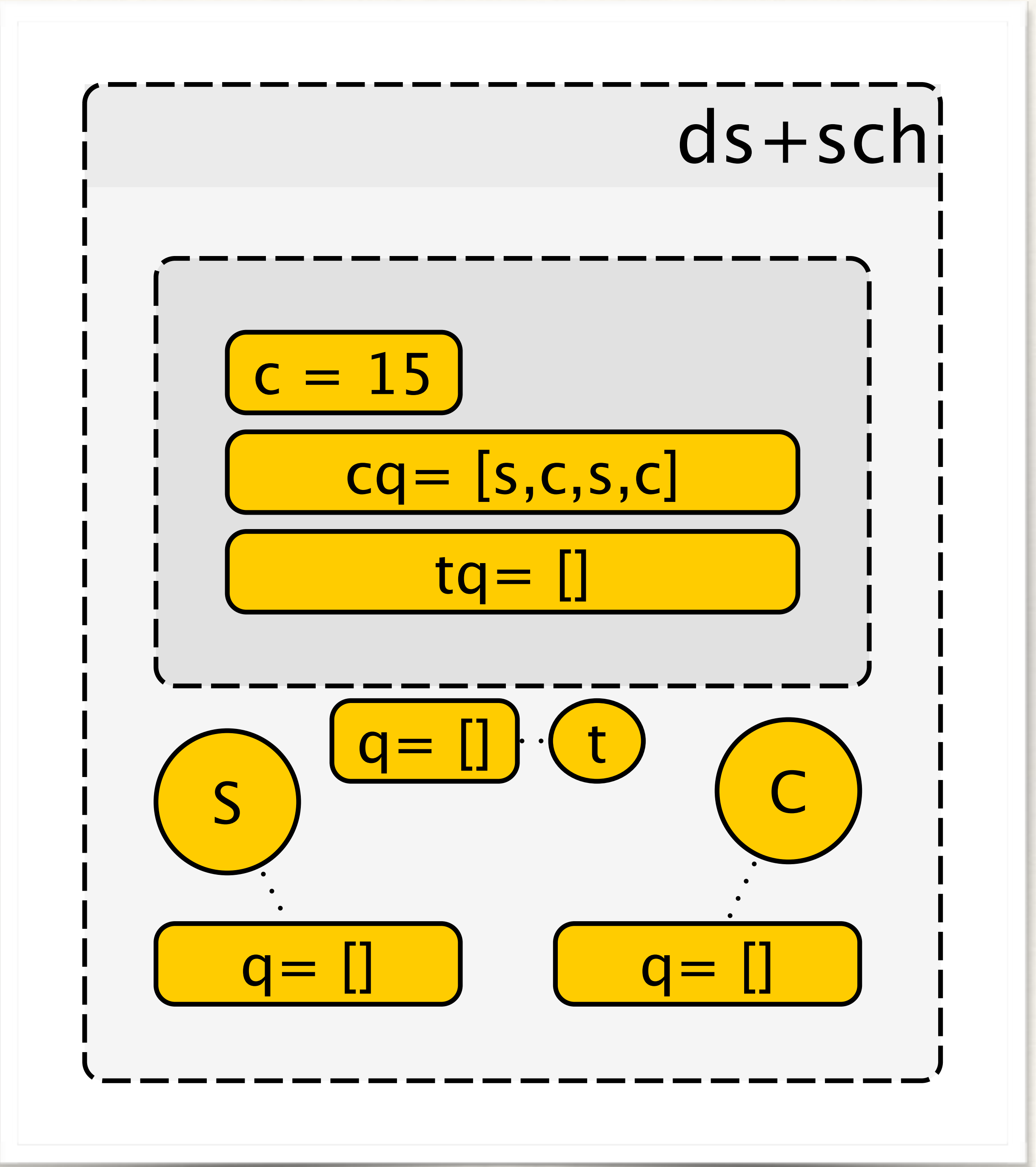
sch.consume(c) // consume GET

sch.consume(s) // consume r-send

sch.consume(c) // "happy"

<u>To Execute:</u>

sch.executeOne // s print("Hello")



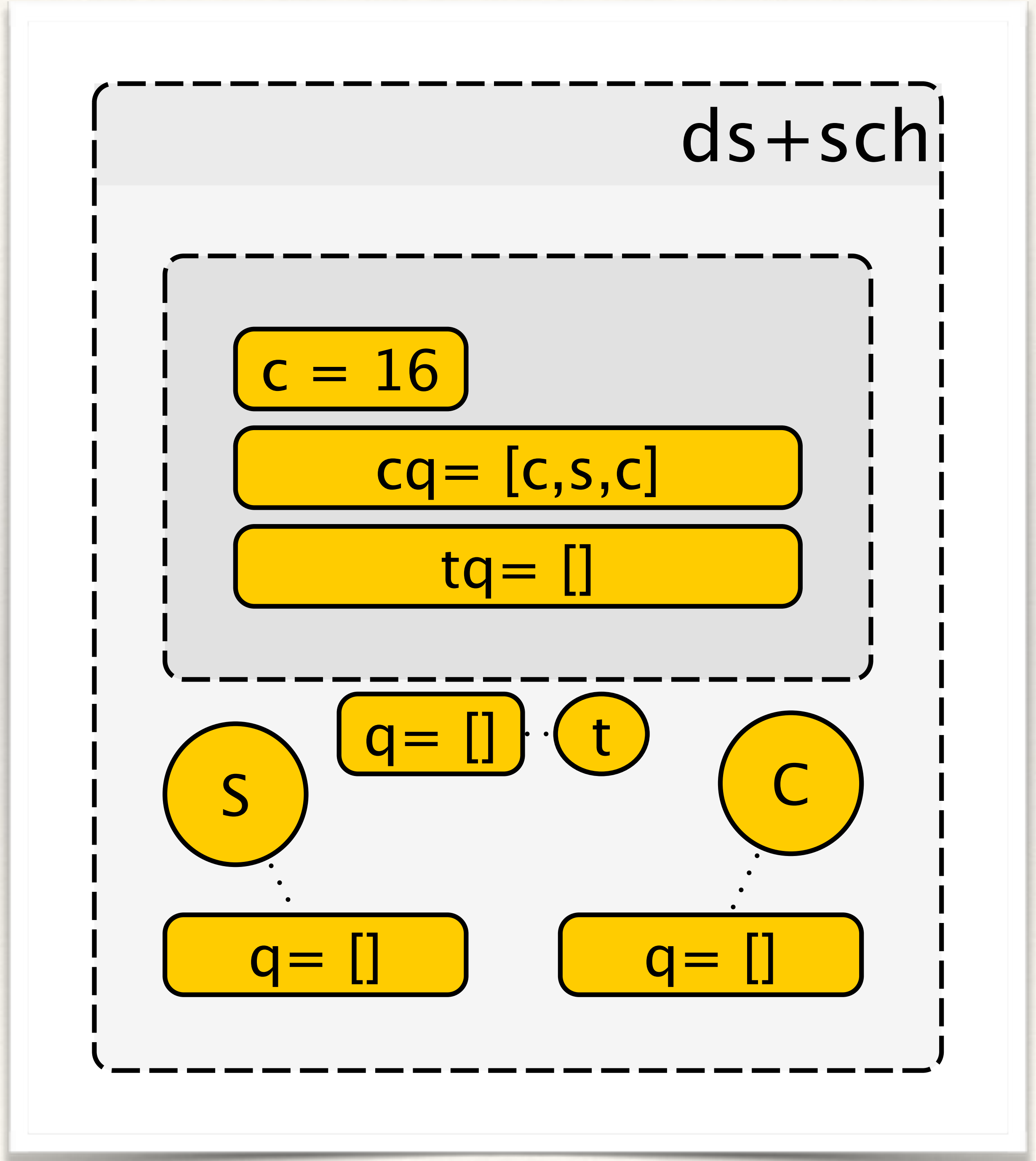
animated schedule

Executed:

sch.executeOne   // print("Hello")

To Execute:

sch.executeOne   // c blocks



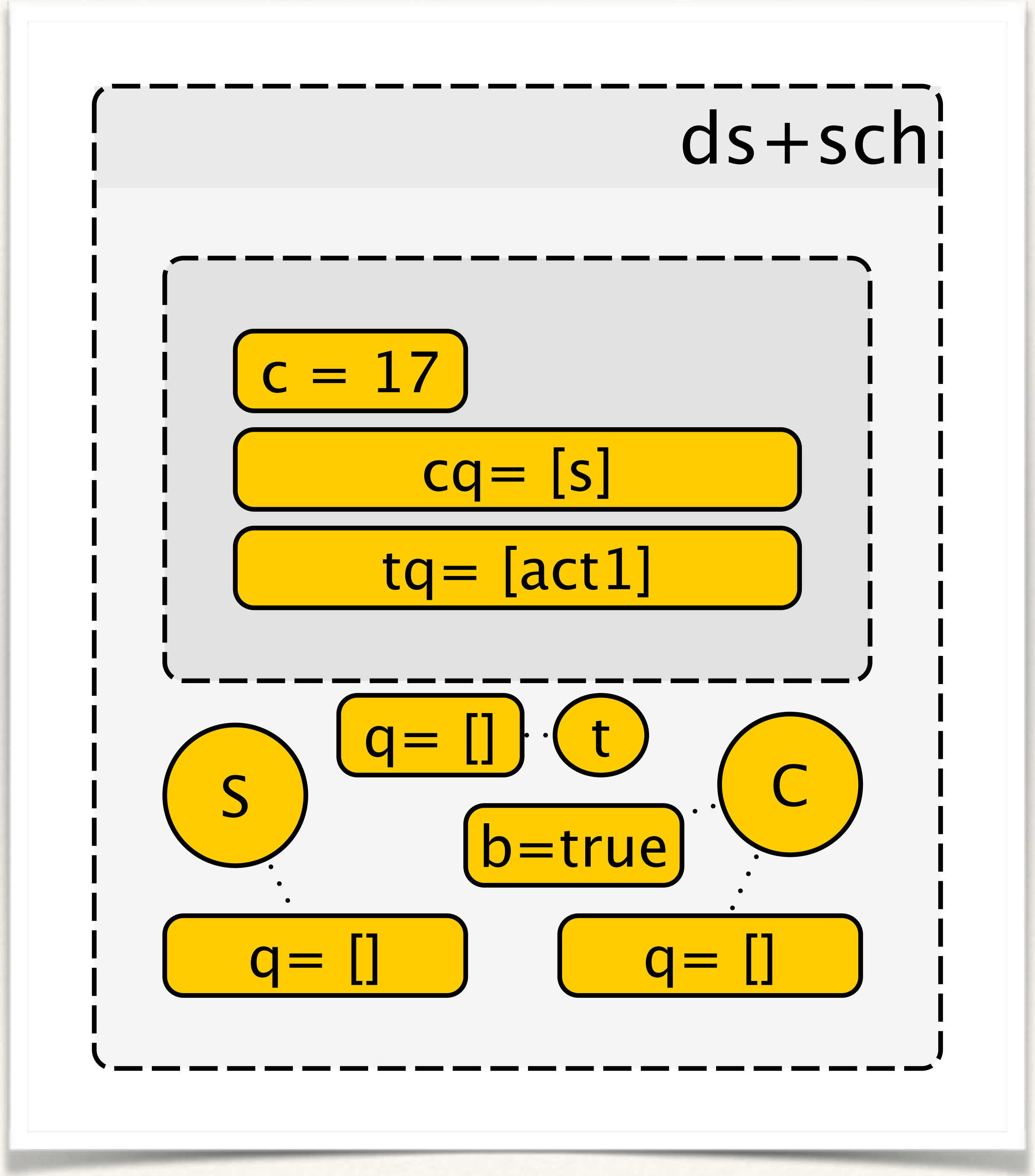
animated schedule

Executed:

sch.executeOne // c blocks

To Execute:

sch.executeOne // r-send(..)



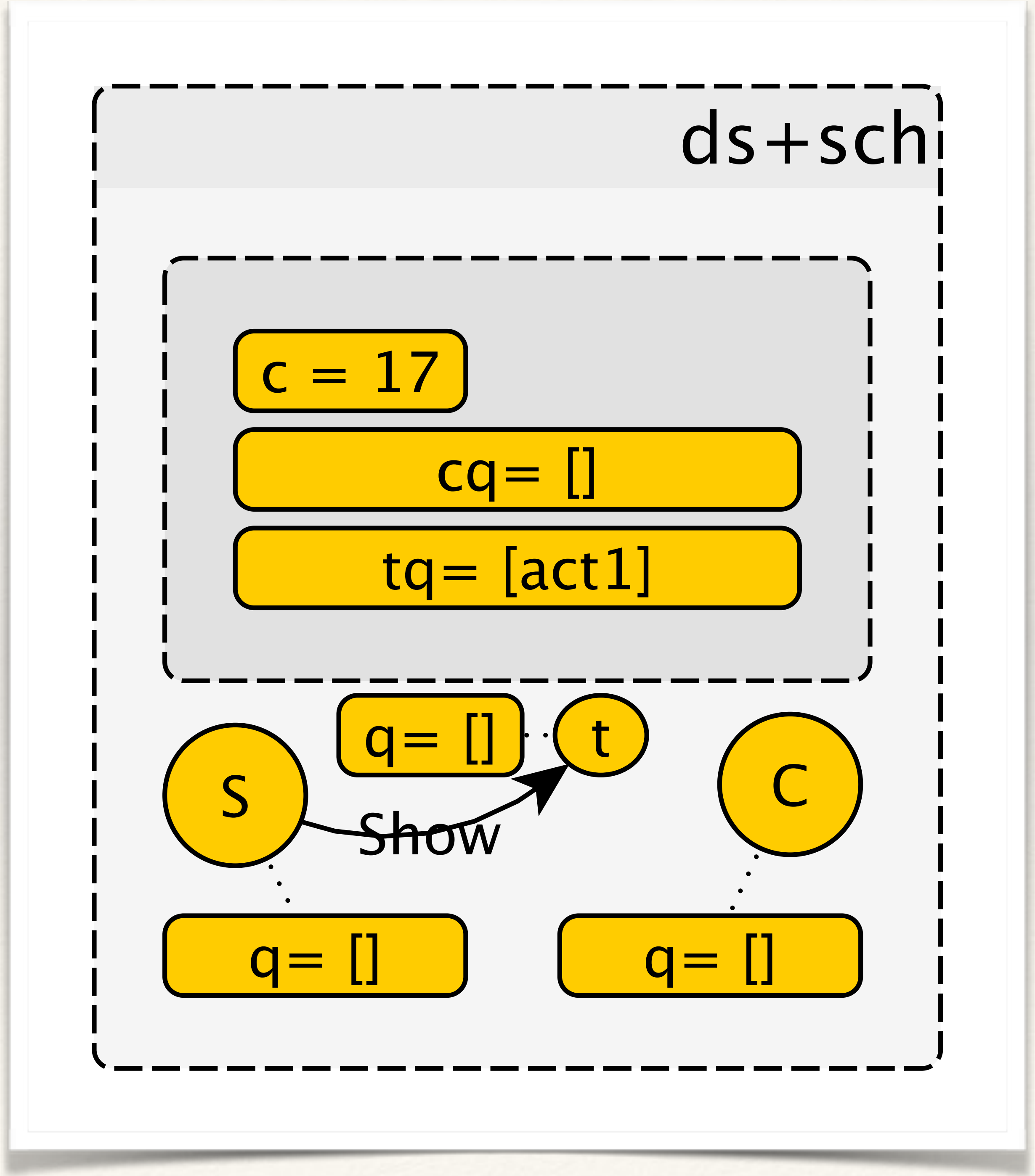
animated schedule

<u>Executed:</u>

sch.executeOne // c blocks

<u>To Execute:</u>

sch.executeOne // r-send(..)



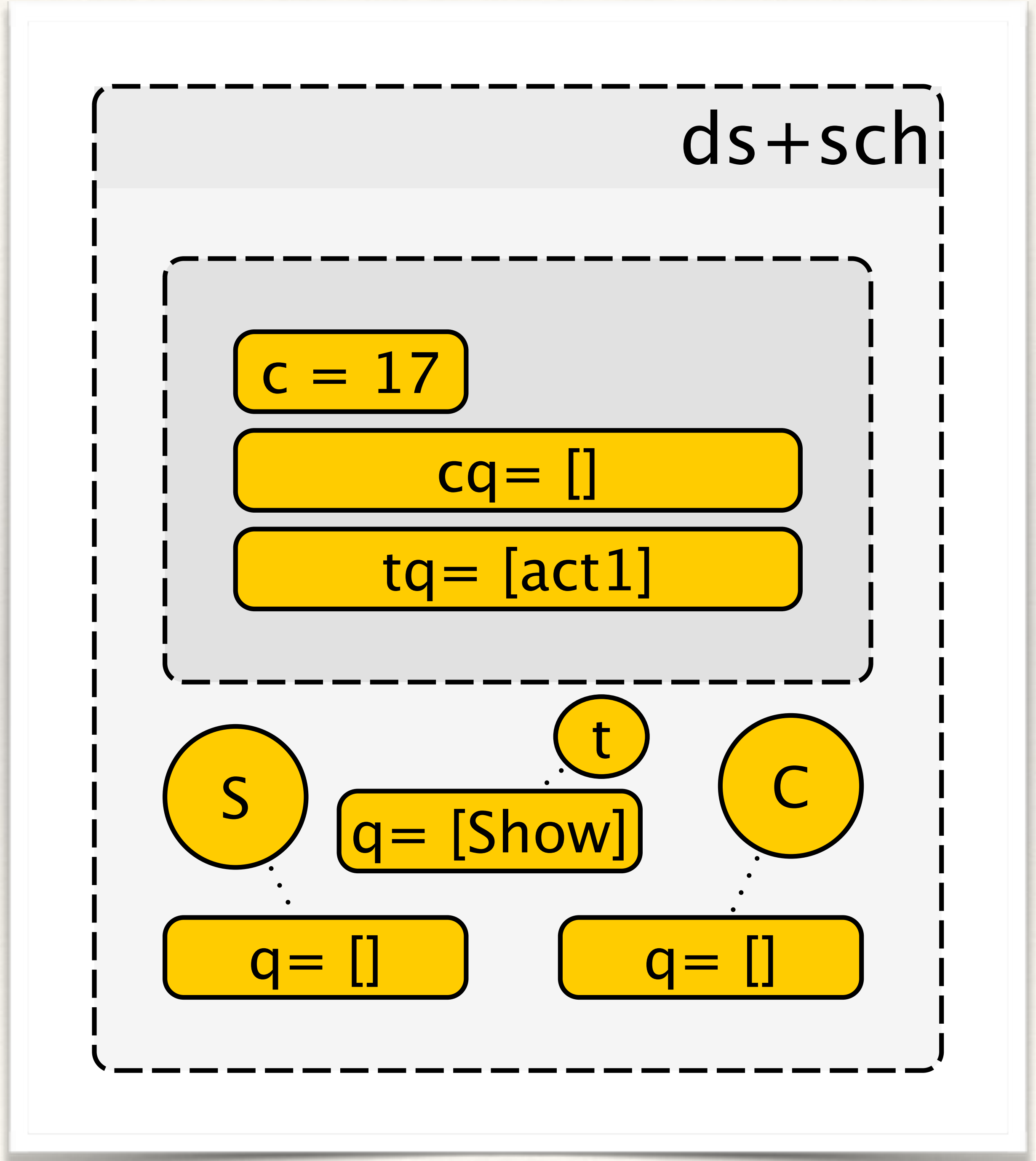
animated schedule

Executed:

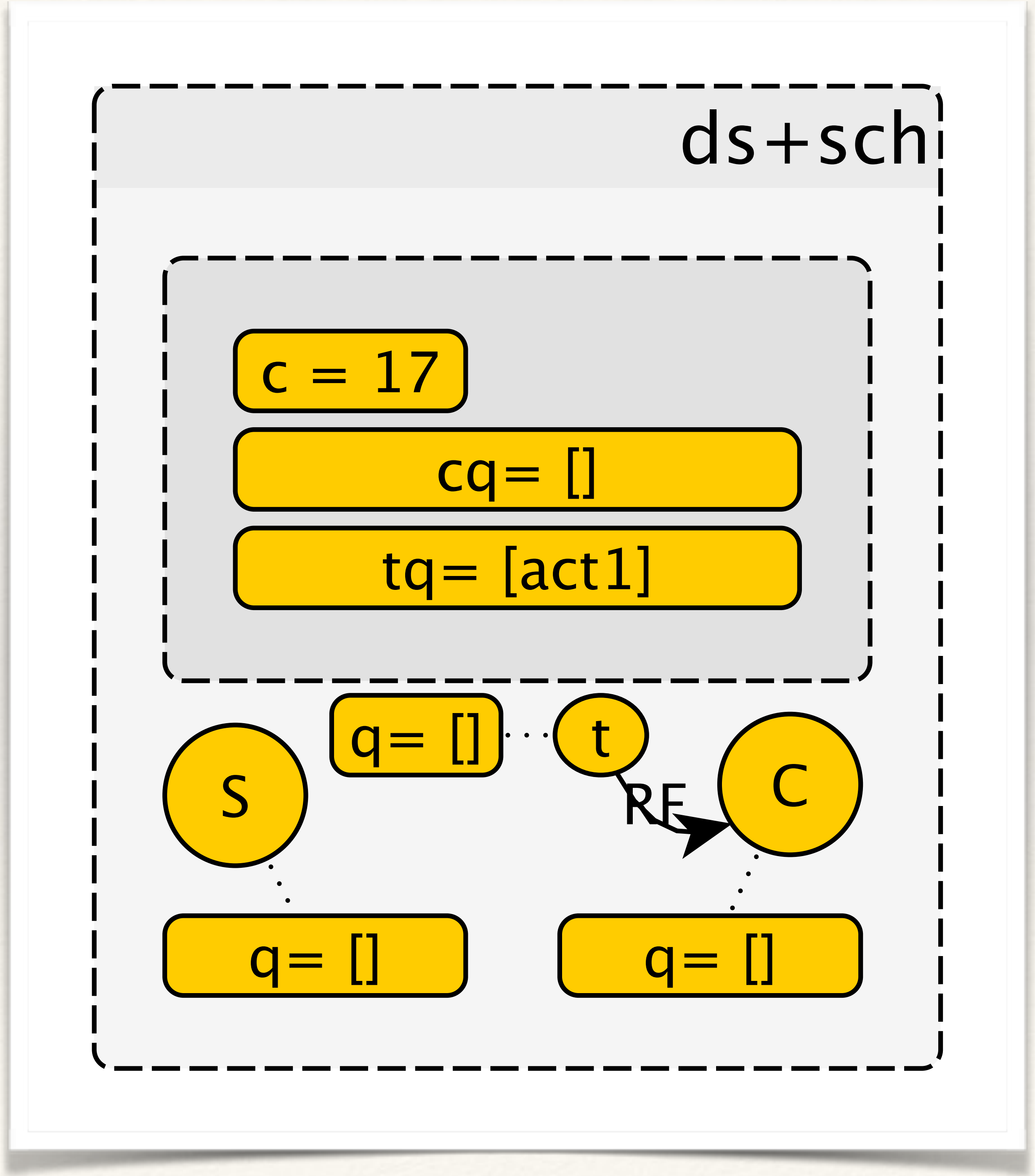
sch.executeOne   // c blocks

To Execute:

sch.executeOne   // r-send(..)



animated schedule



Executed:

sch.executeOne // c blocks

To Execute:

sch.executeOne // r-send(..)