Distributed Systems Abstract Model

Mohammed S. Al-mahfoudh, Ganesh Gopaakrishnan, and Ryan Stutsman

Last updated: April 19, 2016

Chapter 1

The Model & Operational Semantics

1.1 Intro

Distributed systems are notorious to develop, debug and make guarantees about. More over, developers are completely exposed to all difficulties of Distributes Systems (DS) simultaneously, yet they have to make tradeoffs between consistency, network partitioning and/or availability based on application in hand [1]. That makes them distracted about implementation details (the how to) rather than focusing on achieving the final goal of the system (the what to). Past experience of leading industry corporate shows that solutions for same/similar DS problems are re-occurring. e.g. Amazon's Eventually Consistent [2], and Microsoft[3, 4], . . . etc.

This informs us that many things are automate-able and Distributed Systems in general conform to certain model. Hence, we decided to explore the possibility of synthesizing distributed systems. Starting small from event-based fine grained distributed systems, and leaving domain specific tweaking to later explorations (however our model is flexible enough to address the latter with minor additions and modification). Latter maybe concerned with streaming services, hence back pressure should be taken care of, our event-based synthesis addresses control and autonomous distributed systems. However, that generic parsimonious model that is easy to understand, that we could possibly rely on for reasoning and that is extensible, is not existent.

As a result, this technical report presents our Operational Semantics of distributed systems generic enough to adopt for most use cases. That is inspired mostly by Akka [5] actors models and several others like P-language [6], and several distributed systems implementations in actors or in message-passing-based frameworks. In addition, we present the operational semantics interpreter to support our experimentation and validation of both the Operational Semantics model and to automate reasoning about distributed systems reliably.

Our effort shows that distributed systems operational semantics are parsimonious and are not that complicated when isolated properly from programs' logic. Actually, it becomes a generic, yet easy to follow, understand and reason about model that, also, can model a wide variety of concurrency models as well as weak and strong memory consistency models.

1.2 Motivating Example

To illustrate why fault tolerant, reliable, verified and performant distributed systems are crucial, we look at the following motivation example. There are many but we picked the following since it is commonly talked about nowadays with the advent of autonomous vehicles by Audi, Google, . . . etc.

The example:

- If an autonomous car is remotely guided by satellite
- Happens to go behind a mountain that blocks the satellite signals
- This will cause a fault.
  - Faults are very common in such systems, they are the norm.
  - And this situation could be a safety hazard.
- Another monitoring process running on that car detects there is a loss of the important control-signals/connectivity from the satellite.
- The monitoring process, should start a lesser process on the autonomous vehicle that can mitigate or lessen the potential damage.
- The forked process' functions may include the following:
  - Collision detection and avoidance
  - Keeping the speed under some safe limit
  - Steering according to computer vision feedback
- After the satellite connection is restored, the monitoring process communicates the data and state to the satellite process that was absent in order to bring it up to date. It also involves the following:
  - The currently lesser control process should seamlessly hand out the control to the satellite process after the satellite one was brought up to date.
  - The lesser process should shut down cleanly and update the monitoring process
  - The monitoring process should change to the old behavior of monitoring for connectivity of satellite process, instead of monitoring if there is a running lesser process and if not it launches it.

The above is a contrived and is overly simplified example. However, it illustrates how complex quickly a distributed system can become to satisfy the simplest needs of embedded systems. More over, it illustrates the importance and need for rigorously verified, reliable, reactive, fault tolerant and performant distributed systems in an age full of connected things.

As stated in the introduction, this is not an easy feat to achieve by the ordinary means available for engineers and developers alike. There must be an easier, formally specified operational semantics that specify the set of behaviors a distributed system may exhibit and then synthesize what is needed to prevent the erroneous states based on properties specified by the developer. This way engineers get to focus on what to achieve rather than how to achieve it.

1.3 The Operational Semantics States/data-structures

This formulation is similar to the Internet's Autonomous Systems (AS's) where each machine (the distributed system is itself a machine) acts on its locality while communicating with other machines to act on other machines' localities. At the same time, entities and constructs are heavily inspired by the Akka Actors documentation. We, also, think MPI communicating processes model conforms to this model. Figure 2.1 shows a an overview of how data structures compose a distributed system and how the scheduler is connected to it.

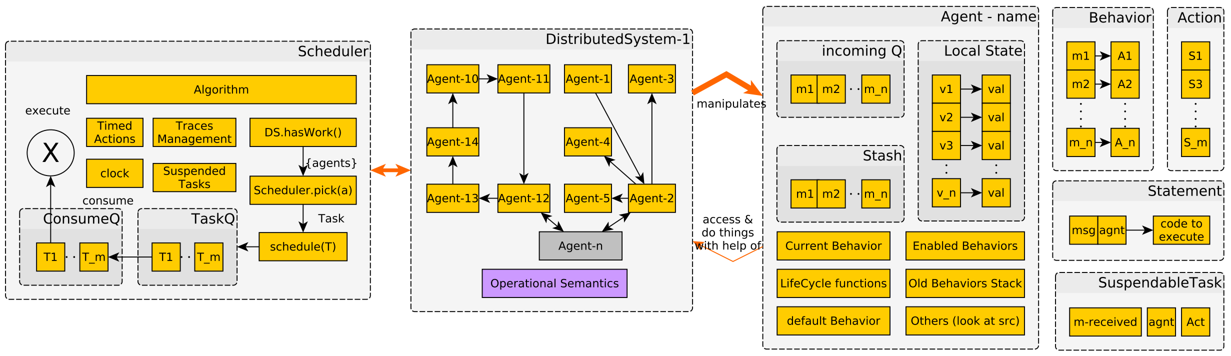

Figure 1.1: Simplified view of data structures and how they relate to each other

[Distributed System]

a set of communicating agents, a set of messages/events passing between them, a set of actions performed upon receiving certain message(s) by agents, and a set of behavior names/IDs mapped to behaviors. It is a tuple of $(\mathcal{A}, \mathcal{M}, \Gamma, \Delta, \Sigma)$, where:

- $\mathcal{A}$: The set of ALL **agents** that comprise the distributed system.
- $\mathcal{M}$: The set of ALL **messages** passed between agents.
- Γ: A set of ALL **actions** to be executed as a reaction to some message arriving at one end (agent).
- Δ: A map of ALL **behaviors**' names to behaviors $f : Ids \rightarrow \mathcal{M} \times \Gamma$.
- Σ: A scheduler attached to this distributed system, the runtime.

[Message]

A named tuple whose payload is a variable number of arguments of at least length 2 (name of message and the sender agent). It is $(\mu, \alpha \in \mathcal{A}, p_1, \cdots, p_n)$, where:

- μ: the name of the message. Not to confuse it when we say $\mu \in \mathcal{M}$ in such case it is the whole message (the whole tuple) reference.
- α: the sender of the message.
- p_1 through p_n: is called the payload and it is a set of immutable data values to be acted upon by the action $\gamma \in \Gamma$ associated with this message type in the destination agent's reactions $\mathcal{R}$.

[Agent]

A self contained communicating process, whose internal state except its incoming buffer is not reachable to other entities in the distributed system. It is a tuple $(q, \theta, \mathcal{S}, \mathcal{R}, \sigma, \tau, \beta, \mathcal{L})$ where:

- q: the thread-safe buffer from which the agent is receiving communication from outer world. Regardless of implementation details(e.g. can be priority queue, concurrent queue . . . etc), it is treated as a FIFO. This queue is modeling a link in reality which can be either reliable or unreliable (e.g. copper wire on same machine or Ethernet cable, respectively).

- θ: The default behavior for the agent.

- $\mathcal{S}$: an ordered set of deferred messages/events called "stash". From which one or more actions are unstashed in the same order they were stashed (a FIFO). Unstash means put back in front of the q, but after all system messages such as RF message instances explained later. Note though, even system messages can be stashed, if chosen by user, however once that is done then the user has to figure out a way to unstash it as we our unstash methods doesn't support unstashing system messages.

- $\mathcal{R}$: Initially, the set of **reactions** in the default behavior θ of an agent, can be swapped with another set of reactions. $\mathcal{R}$ is a non-empty partial function $\mathcal{R} : \mathcal{M} \rightarrow \Gamma = \{(\mu, \gamma)\} \neq \epsilon$, where:

  - $\mu \in \mathcal{M}$: Message/event name
  - $\gamma \in \Gamma$: an **action** executed/performed upon detecting/receiving a message/event. See [Action] below.

- $\mathcal{B}$: a set of **behavior names** the agent exhibits at different times and contexts. That is $\mathcal{B} \subseteq$ Domain(Δ).

- σ: a map of user-override-able special reactions (initially empty, except for σ(s)) when one of the following events happen:

  - σ(s): on-start action, initally implemented to "unlock" the receiving agent's q so that it can receive messages.
  - σ(j): on-join action
  - σ(rj): on-re-join action
  - σ(l): on-leave action
  - σ(d): on-demise action
  - σ(lf): on-link-failure action

- τ: a set of tuples of **timed-actions** to be done by an agent every period/range of time (logical or concrete time can be used, though we implemented only logical time in our interpretter). Tuples in this set takes different forms:

  - $(t, \gamma \in \Gamma)$, where t is the count down time to trigger action γ once it reaches zero, then it fires the action as a task and resets and repeats as long as the agent is not stopped or killed.

  - $(t_1, t_2, \gamma \in \Gamma)$, where t_1 and t_2 are either logical or concrete time. This tuple form specifies an action performed every "range" of time, that is at any instant that falls in time range. We need this to provide time-difference tolerance (error in time sync between communicating agents running on different physical machines).

  Concrete time: time specified in hh:mm:ss:ms. Most likely won't be implemented.

  Logical time: time specified as number of ticks of the scheduler (not the processor), e.g. Lamport clock.

- β: a stack to keep track of old behaviors in case **become(...)** was called with the third argument set to **true**. To use the behavior that sits at the top of the stack (previous behavior) an action must use **unbecome()**

- $\mathcal{L}$: local to agent state (store/memory). Which is a map $\mathcal{L} : Ids \to Values$ where:
  - $Values$: is the set of any value to be stored in memory, local and accessible ONLY to the agent and its actions.
  - Ids: the set of valid id's that can point to memory locations, labels/variable-names.

[Action]

It is a a tuple $(m, a, stmts, xq, dq)$, where:

- **m**: the message received that triggered the action.
- **a**: the agent owning this action, i.e. on which the action and its statements will act on its local state $a[\mathcal{L}]$.
- **stmts**: the sequence of statements composing this action. So, once an action γ is picked for scheduling, it is *reset* so that $\gamma[xq \mapsto stmts, dq \mapsto \epsilon]$.
- **xq**: "to execute queue" of statements that remain to be executed in this action.
- **dq**: the "done queue", a queue of statements that has beed executed.

Statement at the head of the `xq` is executed then removed from it and appended to `dq`. A function call is a statement by itself. Each statement act only on local state of the agent executing it unless it is one of the following, in which case it triggers a rule:

- **send(α_{src},μ,α_{dst})**: send a message μ from α_{src} to α_{dst}, by setting the sender of μ to α_{src} first; then en-queuing μ at the end of α_{dst}'s queue. However, if μ is an RF message then it is put right after the last RF message in the q of $\alpha_d st$. This, in turn, means it may enqueue it at the front of the destination agent's q if there isn't another RF message there.

- **ask(α_{src},μ,α_{dst})**: same as send, but returns a handle f of type future one can block on selectively using $get(f)$ that returns the future's value. A future is a place holder for a value, like a variable but to be populated later in time. It is unresolved as long as it's "resolved" field is $false$ and if resolved, its value field becomes read-only:
  - $get(f)$: blocks as long as the value of f is not resolved, then returns that value once resolved. Otherwise, it blocks forever.

  - *get*(f, t): same as *get*(f) but it blocks until a value is resolved or a timeout t happens. If a timeout happens, though, the value returned is $\bot$.

- **create(**α_p, α_c**)**: agent α_p creates agent α_c. Note that a distributed system has to have at least one agent that may eventually create all other agents.

- **start(**$\alpha_{starter}, \alpha$**)**: start agent α's operation. By default this is implemented to "unlock" α's q.

- **stop(**$\alpha_{stopper}, \alpha$**)**: stop agent operation cleanly, always by another agent or the containing distributed system boot agent.

- **kill(**$\alpha_{killer}, \alpha_{victim}$**)**: agent α_{killer} terminates another agent α_{victim} by sending a special message, i.e. *PoisonPill*. This leads to an abrupt termination of the killed agent that may lead to an ERROR state of the whole distributed system.

- **lock(**$\alpha \in \mathcal{A}$**)**: Makes the queue q of agent α inaccessible (read and write) to all other agents except the owner agent. Makes q not modifiable by all agents except agent α. To revert this effect, use **unlock(q)**.

- **unlock(**$\alpha \in \mathcal{A}$**)**: makes the queue q of agent α accessible (can be written) to **ALL** agents $\alpha \in \mathcal{A}$. However, the owner agent and scheduler are the only two entities that need to read and possibly write to q, while all other agents can only write to queue.

- **resumeConsuming(**$a \in \mathcal{A}$**)**: opens the q of that agent for consuming tasks by the scheduler. That is, when pick() is called it returns a task from that queue in order for the scheduler to consume that task (first by scheduling it then when a thread of its pool is available, it executes that task).

- **stopConsuming(**$a \in \mathcal{A}$**)**: closes the q of that agent for consuming tasks by the scheduler.

- **become(**$\alpha \in \mathcal{A}, b \in \mathcal{B}$**, remember** $= false$**)**: A function that takes the argument b and calls $\Delta(b)$ call it η, it <u>replaces</u> the current agent behavior $\mathcal{R}$ with η. In case the argument *remember* was set to true, the current behavior is added to the top of the stack β.

- **unbecome(**$\alpha \in \mathcal{A}$**)**: pops the behavior b at the top of α's stack β, and replaces the current behavior $\mathcal{R}$ of the agent α with it. If unbecome is called while the stack is empty, the agent restores the default behavior.

- **stash(**$\alpha \in \mathcal{A}, \mu \in \mathcal{M}$**)**: stashes away a message from the front of α's queue and adds it to the end of α's stash $\mathcal{S}$.

- **unstash(**$\alpha \in \mathcal{A}$**)**: removes one message from the front of α's stash $\mathcal{S}$ and puts it in the *front* of α's incoming queue q, but after all messages that are instances of *RF*, or other system messages if existed.

- **unstashAll(**$\alpha \in \mathcal{A}$**)**: same as stash but unstashes all deferred messages from α's stash $\mathcal{S}$ and puts it *in the same order* they appear in $\mathcal{S}$ in the front of α's incoming queue q, and after all system messages in the front of the queue.

Let the above special set of statements/functions be δ.

[Scheduler]

Its a thread pool $\mathcal{P} = (tq, cq, \{t\}, c)$ where:

- tq: is a thread safe queue
- cq: is the thread safe consume queue. Where tasks are moved into from the tq, after calling $consume()$, and then eventually executed by a thread.
- $\{t\}$: a set of at least ONE thread (called single threaded) or more threads (multithreaded) that are de-queuing tasks to execute from tq and performing them. When tq is ϵ threads wait for it to be populated.
- c: a scheduler's counter, counts the number of ticks. Each tick is a statement done executing, can also be adjusted. This is used to formulate logical time. In a multithreaded scheduler its a vector clock, each component is mapped to a thread-id. That is ONE vector clock to track ALL threads progress in time.

NOTE: since it is hard to interpret concrete real time in an abstract way, we will focus on using logical time only.

[Future]

A future is a tuple $future = (resolved \in \{true, false\}, value \in Values, \alpha_{asker} \in \mathcal{A}, \alpha_{replyer} \in \mathcal{A})$

- If $resolved = false$, the future doesn't hold a meaningful value and calling $get(f)$ will block the calling action (in turn agent) running on a thread (the thread running it doesn't get blocked) before until $resolved = true$.
- $\alpha_{replyer}$ is the agent that returned the future and will *potentially* resolve it.
- If $get(f, timeout)$ was called, the thread (And action running on it) will block till either one of the following happen first:
  - f gets resolved.
  - or, timeout happens.

Futures are considered values too, i.e. they can be stored in memory as a variable but their value must be accessed using the $get()$ method(s).

- **Futures**: the set of futures $\mathcal{F} \subseteq Values$

[Special Values]

- $\perp$: indicates no value, e.g. used in place of $null$.
- ϵ: used to indicate a data structure is empty, e.g. queue $= \epsilon$.
- $PoisonPill \in \mathcal{M}$: a special message sent to an agent to be killed.

- $RF(f \in \mathcal{F}) \in \mathcal{M}$: the future resolving message, the future it resolves is encapsulated inside RF as payload f. Important to mention is that this kind of message is only sent by methods and not directly by user's code. It is however important to differentiate it from regular messages to explain the difference how it is handeled by the "send" operation internally to implement the "resolve" semantics.
- ∇: a special state to indicate that a command execution never returns (never ends), and hence freezes in the state in which it was executed.

[Predefined Configurations]

- START: the distributed system is at its start, no messages sent (all queues are empty), and the scheduler has nothing to do.
- ERROR: The system is in error state and needs a special treatment to fix it. Possible error states:
  - Network partitioned: an agent can't send/receive to/from another.
  - Deadlock: at least one agent is waiting on something not being available indefinitely (time out, non-progress of another agent that will free a resource for the agent to proceed).
  - Live lock: at least two processes keep exchanging messages without making progress, indefinitely.
- TERMINATED: same as start but after consuming some/all agent's queues content (in-flight messages). A system may terminate before consuming all, in which case it may/not end up in an error state.

1.4 Conventions and implicits

- **subscripts**: refer to the entity labeled with the same subscript where the currently referred structure comes from/belong-to. For example: if we have agent α_p, then γ_p means the action of agent p that is currently running on a thread.

1.5 Auxiliary functions and predicates

- **hasWork()**: Returns a set of agents $\mathcal{A}' \subseteq \mathcal{A}$ such that $\alpha = \langle q, \cdots \rangle \wedge \mathcal{A}' = \{\forall \alpha \in \mathcal{A} | (q \neq \epsilon \wedge \alpha[consuming \neq false]) \vee RF \in q\}$.
- **pick(α)**: deterministically pick the next message μ in agent α queue and return a tuple (μ,α).
- **pick()**: non-deterministically pick $\alpha \in \mathcal{A}$ and return a pair (μ,α). That is, α is chosen randomly from $hasWork()$. That is, α is the randomly picked entity.
- **schedule($\mu \in \mathcal{M}, \alpha \in \mathcal{A}$)**: synonym for en-queuing a task into a thread pool's queue. The scheduler invokes **pick()** function followed by this function implicitly till all agents queues are empty. Then it waits till one of their queues is populated again to resume scheduling.

- **consume($t \in \{t\}$, $task = (\mu, \alpha)$):** Thread t in scheduler's available threads $\{t\}$ de-queues a *task* from the scheduler's incoming queue *tq* and deposites it in that thread's "consume queue" (cq). A task is a tuple $(\mu \in \mathcal{M}, \gamma = \mathcal{R}(\mu))$ where $\alpha = (\cdots, \mathcal{R}, \cdots)$. We assume that the payload of the message is added within thread t's scope (in its heap) so that everything will be accessible to both the thread and its **exec()**. Everything is: the agent α reacting to the message, the message μ and the message payload (including the sender agent inside of it).

- **the compute operator $\Downarrow$:** used to execute the statement on its left and potentially changes the state to another state are shown to its right after executing the command. Syntax of this command: *command-or-stmt* $\Downarrow$ *change-in-state*. This is the standard big-step semantics and syntax of this compute operator, its meaning is exactly the same as $\rightarrow$ applied many times in small step semantics.

- **dom($f : A \rightarrow B$):** domain of function f, A.

- **rng($f : A \rightarrow B$):** range of function f, B.

- **canSend(α_s, α_d):** a predicate that is *true* iff agent α_s can send to agent α_d, that is iff α_d's queue is unlocked.

- **canConsume(α):** a predicate that is *true* iff the scheduler can consume tasks from Agent α's queue, e.g. after calling $resumeConsuming(\alpha)$. Otherwise, returns $false$.

- **isIdle(t):** returns *true* iff thread t isn't doing any work any more, i.e. isn't executing a task and all tasks executed by it are completely and correctly finished. Otherwise, it returns $false$.

- **execute($\gamma \in \Gamma$):** Executes all statements in action γ as a sequential function call or until it blocks. If it blocks, the thread can switch to another task and the current blocked task will be re-scheduled when unblocked. It is important to mention that this is where timed actions' execution also may interleave with the currently executing action. However, once timed actions are triggered, they take priority to completion (as if they were inlined inside the point of the currently running action) then the current action completes execution (without the current action actually being blocked, it only gets frozen while timed actions execute). This function is a loop that calls, repeatedly, **executeOne(γ)** till the task's action γ has empty `xq` (that is no more statemnts to execute) and content of its **dq** is the same as its **stmts**.

- **executeOne($\gamma \in \Gamma$):** executes ONE statement from the action γ. If $\gamma = (m, a, stmts, s.xq, dq)$, the effect of executing *executeOne*(γ) on the action itself will be $\gamma[s.xq \mapsto xq, dq \mapsto dq.s]$. However, the precise effect of the statement executed on the agent's local state or the distributed system's state depends on the statement kind and attributes. This will be detailed in the following sections.

Overloaded functions. For the sake of keeping rules simpler, we override both **execute(...)** and **executeOne(...)** so that they can take a task as a parameter. Then, internally, these functions act on the action associated with that task.

1.6 Agents' operational semantics rules

For simplicity we will only consider a single-threaded scheduler (i.e. the scheduler has a thread pool, that thread pool has a single thread that processes an event at a time). Also, while reading rules, best is to read them column-wise starting from left most column.

[SEND]

Send to self

$$\frac{\begin{array}{l} s = send(\alpha_s, \mu, \alpha_s) \\ \hdashline executeOne(\gamma) \Downarrow (\gamma[s.xq \mapsto xq, dq \mapsto dq.s] \land \alpha_s[q \mapsto q.\mu] \land \Sigma[c \leftarrow c+1]) \end{array}}{\langle \mathcal{A}[\alpha_s[q, l = false]], \mathcal{M}[\mu], \Gamma[\gamma], \Delta, \Sigma[t[\gamma[s.xq]]]\rangle \rightarrow \langle \mathcal{A}[\alpha_s[q.\mu]], \mathcal{M}[\mu], \Gamma[\gamma], \Delta, \Sigma[t[\gamma[xq, dq.s]], c']\rangle}$$

The above rule describes how an agent sends a message to itself. It first checks if the destination agent is itself, and if that is true, it en-queues the message at the end of its incoming queue. The *send*() statement is obtained from the task by the thread executing it. Then the send operation is executed. If the returned value from executing the statement is *true* then this means the statement was executed successfully and the message μ is delivered, en-queued into the agent's buffer.

Send to other

$$\frac{\begin{array}{l} s = send(\alpha_s, \mu, \alpha_d) \\ \hdashline executeOne(\gamma) \Downarrow (\gamma[s.xq \mapsto xq, dq \mapsto dq.s] \land \alpha_d[q \mapsto q.\mu] \land \Sigma[c \leftarrow c+1]) \end{array}}{\begin{array}{c}\langle \mathcal{A}[\alpha_s, \alpha_d[q, l = false]], \mathcal{M}[\mu], \Gamma[\gamma], \Delta, \Sigma[t[\gamma[s.xq]]]\rangle \rightarrow \langle \mathcal{A}[\alpha_s, \alpha_d[q.\mu]], \mathcal{M}[\mu], \Gamma[\gamma], \\ \Delta, \Sigma[t[\gamma[xq, dq.s]], c']\rangle\end{array}}$$

Describes how an agent sends a message to another agent. That is by en-queuing a message at the back of the other agent's α_d incoming messages queue.

[ASK]

Asking

$$\frac{\begin{array}{l} s = ask(\alpha_s, \mu, \alpha_d) \\ \hdashline executeOne(\gamma) \Downarrow (\gamma[s.xq \mapsto xq, dq \mapsto dq.s] \land fresh(f) \in \mathcal{F} \land \alpha_d[q \mapsto q.\mu] \land \\ f[resolved \leftarrow false, \alpha_{asker} \leftarrow \alpha_s, \alpha_{replyer} \leftarrow \alpha_d] \land \alpha_s[\mathcal{L}(f[id]) \leftarrow f] \land \Sigma[c \leftarrow c+1]) \end{array}}{\begin{array}{c}\langle \mathcal{A}[\alpha_s, \alpha_d[q, l = false]], \mathcal{M}[\mu], \Gamma[\gamma], \Delta, \Sigma[t[\gamma[s.xq]]]\rangle \rightarrow \langle \mathcal{A}[\alpha_s[\mathcal{L}'], \alpha_d[q.\mu]], \mathcal{M}[\mu], \\ \Gamma[\gamma], \Delta, \Sigma[t[\gamma[xq, dq.s]], c']\rangle\end{array}}$$

Asking an agent something means sending it a message, getting a future on which the source agent of the first message can block any time to get the value of the future if that has been resolved. A future is like a place holder or a container for some value. It is locked (i.e. can't be read) as long as that value is not written to it, and unlocked ever after it is written.

Details: Above, in the operational semantics, an ask statement is detailed as a send with a handle for an object to be populated by a receiver of the first send. This is done in purpose so that implementation details don't bloat or restricts the abstract view of the operational semantics. However, we treat it, in our interpretter as an elastic (in time) round trip transction. That is, one send with a handle returned to the sender, followed some time in the future by another send of the original receiver (i.e. the reply which is a send in the opposit direction) that populates the handle object with a value. It is clear at this point how many things could go wrong in the transaction and that this model captures that without shadowing these potential problems.

[GET-FUTURE]

Blocking on a future and it resolves

$$\frac{\begin{array}{c} (s = get(\alpha_s, f) \vee s = get(\alpha_s, f, timeout)) \wedge f \in \mathcal{F} \\ \hdashline executeOne(\gamma) \Downarrow (\gamma[s.xq \mapsto xq, dq \mapsto dq.s] \wedge \alpha_d[q \mapsto q.\mu] \wedge \Sigma[c \leftarrow c+1]) \end{array}}{\begin{array}{c} \langle \mathcal{A}[\alpha_s[l = false, \mathcal{L}[f[resolved = true]]]], \mathcal{M}, \Gamma[\gamma], \Delta, \Sigma[t[\gamma[s.xq]]]\rangle \rightarrow \\ \langle \mathcal{A}[\alpha_s[\mathcal{L}(f[id]) \leftarrow f]], \mathcal{M}[\mu], \Gamma[\gamma], \Delta, \Sigma[t[\gamma[xq, dq.s]], c']\rangle \end{array}}$$

$$\frac{\begin{array}{c:c} \alpha_s, \alpha_d \in \mathcal{A} & \gamma_s[\mathcal{PC}_s] = st_s \\ \hdashline f_s \in \mathcal{F} & st_s = get(\alpha_s, f_s) \vee st_s = get(\alpha_s, f_s, timeout) \\ \hdashline f_s \in dom(\mathcal{L}_s) & st_s \Downarrow \alpha_s[\mathcal{L} \mapsto \mathcal{L}[f_s] = v] \wedge \mathcal{PC}_s \mapsto \mathcal{PC}_s + 1 \\ \hdashline \gamma_s \in \{rng(\mathcal{R}_s) \cup rng(\sigma_s)\} & \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta\rangle \rightarrow \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta\rangle}$$

Trying to $get()$ the value of a resolved future updates the local state of the agent and does *not* block the agent.

Blocking on a future and it times out

$$\frac{\begin{array}{c:c} \alpha_s, \alpha_d \in \mathcal{A} & \gamma_s[\mathcal{PC}_s] = st_s \\ \hdashline f_s \in \mathcal{F} & st_s = get(\alpha_s, f_s, timeout) \\ \hdashline f_s \in dom(\mathcal{L}_s) & timeout = 0 \\ \hdashline \gamma_s \in \{rng(\mathcal{R}_s) \cup rng(\sigma_s)\} & st_s \Downarrow \alpha_s \wedge \mathcal{PC}_s \mapsto \mathcal{PC}_s + 1 \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta\rangle \rightarrow \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta\rangle}$$

Trying to $get()$ the value of un-resolved future then timing out, won't change the memory state of the agent. However, it will change the execution state of the agent, by advancing the program counter to next statement. A timeout here is a countdown to ZERO.

Blocking on a future indefinitely

$$\frac{\begin{array}{c:c} \alpha_s, \alpha_d \in \mathcal{A} & \gamma_s[\mathcal{PC}_s] = st_s \\ \hdashline f_s \in \mathcal{F} & st_s = get(\alpha_s, f_s) \\ \hdashline f_s \in dom(\mathcal{L}_s) & \alpha_s[\mathcal{L}(f_s) = (false, \perp, \alpha_s, \alpha_d)] \\ \hdashline \gamma_s \in \{rng(\mathcal{R}_s) \cup rng(\sigma_s)\} & st_s \Downarrow \nabla \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta\rangle \rightarrow ERROR}$$

Trying to $get()$ the value of un-resolved future using the non-timing-out blocking $get()$, won't change the memory state of the agent nor will advance the program counter. Potentially never advancing, the agent's state becomes "blocked" (i.e. program counter executing agent's action $\mathcal{PC}_s$ isn't advancing) as long as the future is not resolved. After it is resolved, then the agent triggers the first rule of "get-future". We can easily see that a potential deadlock can happen if it was never resolved.

[LOCK]

Agent locks its queue from receiving messages

$$\frac{\begin{array}{c:c} \alpha_1 \in \mathcal{A} & \gamma_1[\mathcal{PC}_1] = st_1 \\ \hdashline \forall \alpha' \in \mathcal{A} \setminus \{\alpha_1\} : canSend(\alpha', \alpha_1) & st_1 = lock(\alpha_1) \\ \hdashline \gamma_1 \in \{rng(\mathcal{R}_1) \cup rng(\sigma_1)\} & st_1 \Downarrow \alpha_1[q \mapsto q'] \wedge \forall \alpha' : \neg canSend(\alpha', \alpha_1) \wedge \mathcal{PC}_1 \mapsto \mathcal{PC}_1 + 1 \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \rightarrow \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta \rangle}$$

Locking an agent's queue, when it was unlocked and when all agents could write to it, makes it not writable to all agents except the owner agent. The owner agent always can read and write from/to q.

Agent locking its already locked queue changes nothing

$$\frac{\begin{array}{c:c} \alpha_1 \in \mathcal{A} & \gamma_1[\mathcal{PC}_1] = st_1 \\ \hdashline \forall \alpha' \in \mathcal{A} \setminus \{\alpha_1\} : \neg canSend(\alpha', \alpha_1) & st_1 = lock(\alpha_1) \\ \hdashline \gamma_1 \in \{rng(\mathcal{R}_1) \cup rng(\sigma_1)\} & st_1 \Downarrow \alpha_1 \wedge \mathcal{PC}_1 \mapsto \mathcal{PC}_1 + 1 \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle}$$

Locking an already locked data structure doesn't change a thing, i.e. it stays locked and no error is thrown. The owner agent can read and write to it all the time, whether it is locked or unlocked.

NOTE: It is important to note that changing the program counter doesn't count as changing the state of the distributed system, it is merely a change in the scheduler (the runtime), which models an algorithm for exploring schedules and this is what we need to achieve.

[UNLOCK]

Agent unlocks its queue for receiving messages

$$\frac{\begin{array}{c:c} \alpha_1 \in \mathcal{A} & \gamma_1[\mathcal{PC}_1] = st_1 \\ \hdashline \forall \alpha' \in \mathcal{A} \setminus \{\alpha_1\} : \neg canSend(\alpha', \alpha_1) & st_1 = unlock(\alpha_1) \\ \hdashline \gamma_1 \in \{rng(\mathcal{R}_1) \cup rng(\sigma_1)\} & st_1 \Downarrow \alpha_1[q \mapsto q'] \wedge \forall \alpha' : canSend(\alpha', \alpha_1) \wedge \mathcal{PC}_1 \mapsto \mathcal{PC}_1 + 1 \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \rightarrow \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta \rangle}$$

Unlocking an agent's queue, when it was not writable nor readable by any agent except its owner, makes it writable by all agents.

Agent unlocking its already unlocked queue changes nothing

$$\frac{\begin{array}{c:c} \alpha_1 \in \mathcal{A} & \gamma_1[\mathcal{PC}_1] = st_1 \\ \hdashline \forall \alpha' \in \mathcal{A} \setminus \{\alpha_1\} : canSend(\alpha', \alpha_1) & st_1 = unlock(\alpha_1) \\ \hdashline \gamma_1 \in \{rng(\mathcal{R}_1) \cup rng(\sigma_1)\} & st_1 \Downarrow \alpha_1 \land \mathcal{PC}_1 \mapsto \mathcal{PC}_1 + 1 \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle}$$

Unlocking an already unlocked a data structure doesn't change a thing, i.e. it stays unlocked and every agent can write to it and its owner can also read from it. It doesn't throw an error.

[CREATE]

Agent p creates a new agent c

$$\frac{\begin{array}{c:c} \alpha_p \in \mathcal{A} & st_1 = create(\alpha_p, \alpha_c) \\ \hdashline \gamma_p \in \{rng(\mathcal{R}_p) \cup rng(\sigma_p)\} & \begin{array}{c} st_1 \Downarrow \alpha_c[\theta \mapsto \theta_p, \mathcal{R} \mapsto \theta_p, \mathcal{B} \mapsto \mathcal{B}_p, \sigma \mapsto \sigma_p] \land \mathcal{A} \mapsto \mathcal{A} \cup \{\alpha_c\} \land \mathcal{PC}_p \mapsto \\ \mathcal{PC}_p + 1 \end{array} \\ \hdashline \gamma_p[\mathcal{PC}_p] = st_1 & \end{array}}{\begin{array}{c} \langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \to \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta \rangle \land \\ (\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \to \langle \mathcal{A}, \mathcal{M}', \Gamma, \Delta \rangle \lor \\ \langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \to \langle \mathcal{A}, \mathcal{M}, \Gamma', \Delta \rangle \lor \\ \langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \to \langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta' \rangle) \end{array}}$$

If an agent α_p is creating an agent α_c and it succeeded, agent α_c will inherit few attributes from agent α_p. These few attributes are: $\theta, \mathcal{B}, \sigma$. Other attributes will be fresh objects. Then, the created agent will be added to the universal agents set. After that, the creator agent will start the new agent α_c, triggering "start" and "join" rules consecutively. "start" implicitly triggers "join".

[START]

Starting an agent

$$\frac{\begin{array}{c:c} \alpha_p, \alpha_c \in \mathcal{A} & st_1 = start(\alpha_p, \alpha_c) \\ \hdashline \gamma_p \in \{rng(\mathcal{R}_p) \cup rng(\sigma_p)\} & st_1 \Downarrow \alpha_c[q \mapsto q.s] \land \mathcal{PC}_p \mapsto \mathcal{PC}_p + 1 \\ \hdashline \gamma_p[\mathcal{PC}_p] = st_1 & \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \to \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta \rangle}$$

After creating an agent, it has a locked incoming queue. It sends it self two messages before unlocking its queue q to other agents. These two messages are, in order, s and j. After that, it unlocks its queue for other agents to be able to communicate with it.

[STOP]

Stopping an agent

$$\frac{\begin{array}{c:c} \alpha_1, \alpha_2 \in \mathcal{A} & st_1 = stop(\alpha_1, \alpha_2) \\ \hdashline \gamma_1 \in \{rng(\mathcal{R}_1) \cup rng(\sigma_1)\} & st_1 \Downarrow \alpha_2[q \mapsto q.l] \wedge \mathcal{PC}_1 \mapsto \mathcal{PC}_1 + 1 \\ \hdashline \gamma_1[\mathcal{PC}_1] = st_1 & \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \to \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta \rangle}$$

For an agent to be stopped, another agent must send 'l' special message to it. The reaction to this message to be done by the scheduler on the agent involves calling the *onLeave*() of that specific agent to do the necessary. That behavior is implemented by the users. However, a safe implementation will involve locking the queue of the agent, scheduling all previously scheduled tasks, and then executing all of them by the scheduler. After that, it is safe to assume the agent is completely and safely stopped, i.e. onDemise can be called safely to free resources of the already stopped agent.

[KILL]

Killing an agent an agent

$$\frac{\begin{array}{c:c} \alpha_1, \alpha_2 \in \mathcal{A} & st_1 = kill(\alpha_1, \alpha_2) \\ \hdashline \gamma_1 \in \{rng(\mathcal{R}_1) \cup rng(\sigma_1)\} & st_1 \Downarrow \alpha_2[q \mapsto q.PoisonPill] \wedge \mathcal{PC}_1 \mapsto \mathcal{PC}_1 + 1 \\ \hdashline \gamma_1[\mathcal{PC}_1] = st_1 & \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \to ERROR}$$

If agent α_1 kills an agent α_2, it sends it the special message $PoisonPill$. Once that message is consumed by α from the front of its queue, and a task $= (PoisonPill, \alpha_2)$ has been scheduled to be executed by the thread pool, it may get executed too soon. Then, all resources of the killed agent is freed by calling the *onDemise*() special reaction on that agent, leading to possible loss of scheduled tasks by α_2'. This possibly leads to an erroneous state of the distributed system. The reason is that further interactions possibly invoked by still running tasks of this or other agent(s), however with the killed agent not being able to participate anymore leads to a mess.

[BECOME]

Changing the current behavior to another

$$\frac{\begin{array}{c:c} \alpha_1 \in \mathcal{A} & st_1 = become(\alpha_1, b, false) \\ \hdashline \gamma_1 \in \{rng(\mathcal{R}_1) \cup rng(\sigma_1)\} & b \in \{\mathcal{B}_1 \cap dom(\Delta)\} \\ \hdashline \gamma_1[\mathcal{PC}_1] = st_1 & st_1 \Downarrow \alpha_1[\mathcal{R}_1 \mapsto \Delta[b]] \wedge \mathcal{PC}_1 \mapsto \mathcal{PC}_1 + 1 \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \to \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta \rangle}$$

Changing an agent's behavior to a new one using *become*() and setting its last argument to the default ($false$), results in forgetting the old behavior for ever and getting replaced by the new one. Except if the old behavior is the default one, then it can be restored since it is stored as θ.

Changing the current behavior and remembering the old behavior

$$\frac{\begin{array}{c|c} \alpha_1 \in \mathcal{A} & st_1 = become(\alpha_1, b, true) \\ \hdashline \gamma_1 \in \{rng(\mathcal{R}_1) \cup rng(\sigma_1)\} & b \in \{\mathcal{B}_1 \cap dom(\Delta)\} \\ \hdashline \gamma_1[\mathcal{PC}_1] = st_1 & st_1 \Downarrow \alpha_1[\beta \mapsto \mathcal{R}_1.\beta, \mathcal{R}_1 \mapsto \Delta[b]] \wedge \mathcal{PC}_1 \mapsto \mathcal{PC}_1 + 1 \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \rightarrow \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta \rangle}$$

Changing an agent's behavior while setting the last argument to *true*, will put the old behavior at the top of the agent's stack of behaviors β.

[UNBECOME]

Changing the current behavior to an old one

$$\frac{\begin{array}{c|c} \alpha_1 \in \mathcal{A} & st_1 = unbecome(\alpha_1) \\ \hdashline \gamma_1 \in \{rng(\mathcal{R}_1) \cup rng(\sigma_1)\} & st_1 \Downarrow \alpha_1[R \mapsto \Delta[b], b.\beta \mapsto \beta] \wedge \mathcal{PC}_1 \mapsto \mathcal{PC}_1 + 1 \\ \hdashline \gamma_1[\mathcal{PC}_1] = st_1 & \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \rightarrow \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta \rangle}$$

If an agent calls unbecome, the agent's stack of behaviors is investigated. If it is not empty, then the behavior at the top of the stack is used to replace the current behavior of the agent $\mathcal{R}$.

Changing the current behavior to the default

$$\frac{\begin{array}{c|c} \alpha_1 \in \mathcal{A} & st_1 = unbecome(\alpha_1) \\ \hdashline \gamma_1 \in \{rng(\mathcal{R}_1) \cup rng(\sigma_1)\} & \beta = \{\epsilon\} \\ \hdashline \gamma_1[\mathcal{PC}_1] = st_1 & st_1 \Downarrow \alpha_1[R \mapsto \theta] \wedge \mathcal{PC}_1 \mapsto \mathcal{PC}_1 + 1 \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \rightarrow \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta \rangle}$$

If an agent unbecomes while its stack of behavior is empty, it restores its default behavior.

[STASH]

Delay processing a message for later time/different-behavior

$$\frac{\begin{array}{c|c} \alpha_1 \in \mathcal{A} & st_1 = stash(\alpha_1, \mu) \\ \hdashline \gamma_1 \in \{rng(\mathcal{R}_1) \cup rng(\sigma_1)\} & st_1 \Downarrow \alpha_1[\mathcal{S} \mapsto \mathcal{S}.\mu] \wedge \mathcal{PC}_1 \mapsto \mathcal{PC}_1 + 1 \\ \hdashline \gamma_1[\mathcal{PC}_1] = st_1 & \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \rightarrow \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta \rangle}$$

If an agent's action called $stash(\mu)$, μ is the current message to be processed. Then, that message is *en-queued* in its stash $\mathcal{S}$, at the end of that stash. A stash is just a FIFO that doesn't need to be thread safe as it is modified internally by the agent itself (even if scheduled as a task, which it is).

[UNSTASH]

Resuming processing a message that was delayed

$$\frac{\begin{array}{c:c} \alpha_1 \in \mathcal{A} & st_1 = unstash(\alpha_1) \\ \hdashline \gamma_1 \in \{rng(\mathcal{R}_1) \cup rng(\sigma_1)\} & \mu \in \mathcal{M} \setminus \{RF\} \\ \hdashline \gamma_1[\mathcal{PC}_1] = st_1 & st_1 \Downarrow \alpha_1[q \mapsto \mu.q, \mu.\mathcal{S} \mapsto \mathcal{S}] \wedge \mathcal{PC}_1 \mapsto \mathcal{PC}_1 + 1 \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta\rangle \rightarrow \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta\rangle}$$

Unstashing a single message means removing a message from the front of the stash (its a FIFO), and putting it back at the front of the agents queue. Why front? because it arrived before all messages in the queue and hence it gets the precedence.

Unstashing a message to a queue containing system messages

$$\frac{\begin{array}{c:c} \alpha_1 \in \mathcal{A} & st_1 = unstash(\alpha_1) \\ \hdashline \gamma_1 \in \{rng(\mathcal{R}_1) \cup rng(\sigma_1)\} & \mu \in \mathcal{M} \setminus \{RF\} \\ \hdashline \gamma_1[\mathcal{PC}_1] = st_1 & st_1 \Downarrow \alpha_1[RF^+.q \mapsto RF^+.\mu.q, \mu.\mathcal{S} \mapsto \mathcal{S}] \wedge \mathcal{PC}_1 \mapsto \mathcal{PC}_1 + 1 \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta\rangle \rightarrow \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta\rangle}$$

Unstashing a message when there are system messages (such as RF) in the front of the queue, resultes in unstashing that message to the front most of the queue but after all system messages in the front of it.

Unstashing a system message

$$\frac{\begin{array}{c:c} \alpha_1 \in \mathcal{A} & st_1 = unstash(\alpha_1) \\ \hdashline \gamma_1 \in \{rng(\mathcal{R}_1) \cup rng(\sigma_1)\} & \mu \in \{RF\} \\ \hdashline \gamma_1[\mathcal{PC}_1] = st_1 & st_1 \Downarrow \alpha_1[q, \mu.\mathcal{S}] \wedge \mathcal{PC}_1 \mapsto \mathcal{PC}_1 + 1 \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta\rangle}$$

Unstashing from a stash containing system messages leads to leaving those system messages in the stash. We don't support stashing (although our implementation allows doing that using the same stash method) and unstashing of system messages as that doesn't play nicely with the operational semantics. We leave that to be decided by scheduler's implementors to decide how to handel that manually.

Resuming processing ALL messages that were delayed

$$\frac{\begin{array}{c:c} \alpha_1 \in \mathcal{A} & st_1 = unstashAll(\alpha_1) \\ \hdashline \gamma_1 \in \{rng(\mathcal{R}_1) \cup rng(\sigma_1)\} & st_1 \Downarrow \alpha_1[q \mapsto \mu^+.q, \mu^+.\mathcal{S} \mapsto \mathcal{S}] \wedge \mathcal{PC}_1 \mapsto \mathcal{PC}_1 + 1 \\ \hdashline \gamma_1[\mathcal{PC}_1] = st_1 & \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta\rangle \rightarrow \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta\rangle}$$

Like unstashing one message but with a subtle but important difference. Unstashing all messages means prepending the entirety of the stash into the front of the agents incoming queue. Then, removing all the messages from the stash (emptying it not to cause duplication when unstashing again). Now, if we used unstash-one-message multiple times, it will put all messages in front of the agent's queue *but* in reverse order, this is a semantic error. This is why it is important to differentiate.

Resuming All messages but with system messages in the queue

$$\frac{\begin{array}{c|c} \alpha_1 \in \mathcal{A} & st_1 = unstashAll(\alpha_1) \\ \hdashline \gamma_1 \in \{rng(\mathcal{R}_1) \cup rng(\sigma_1)\} & \forall \mu \in \mathcal{S} : \mu \in \mathcal{M} \setminus \{RF\} \\ \hdashline \gamma_1[\mathcal{PC}_1] = st_1 & st_1 \Downarrow \alpha_1[RF^{+}.q \mapsto RF^{+}.\mu^{+}.q, \mu^{+}.\mathcal{S} \mapsto \mathcal{S}] \wedge \mathcal{PC}_1 \mapsto \mathcal{PC}_1 + 1 \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \rightarrow \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta \rangle}$$

Unstashing all messages from the stash, given that they don't have a system message, into a queue containing system messages leaves the stash empty and appends those messages behind all system messages in the queue but before all normal messages.

Resuming All messages but with system messages in the stash

$$\frac{\begin{array}{c|c} \alpha_1 \in \mathcal{A} & st_1 = unstashAll(\alpha_1) \\ \hdashline \gamma_1 \in \{rng(\mathcal{R}_1) \cup rng(\sigma_1)\} & \forall \mu \in \mathcal{S} : \mu \in \mathcal{M} \setminus \{RF\} \\ \hdashline \gamma_1[\mathcal{PC}_1] = st_1 & st_1 \Downarrow \alpha_1[q \mapsto \mu^{+}.q, RF^{+}.\mu^{+}.\mathcal{S} \mapsto RF^{+}.\mathcal{S}] \wedge \mathcal{PC}_1 \mapsto \mathcal{PC}_1 + 1 \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \rightarrow \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta \rangle}$$

Unstashing all messages from the stash while having some stashed system messages, leads to leaving those system messages in the stash while unstashing normal ones in the same order to the queue.

[BOOTSTRAP]

Boot strapping an agent means starting it

$$\frac{\begin{array}{c|c} \alpha_{boot}, \alpha_{booted} \in \mathcal{A} & st_1 = bootStrap(\alpha_{boot}, \alpha_{booted}) \\ \hdashline \gamma_p \in \{rng(\mathcal{R}_p) \cup rng(\sigma_p)\} & st_1 \Downarrow \alpha_{booted}[q \mapsto q.s] \wedge \mathcal{PC}_{boot} \mapsto \mathcal{PC}_{boot} + 1 \\ \hdashline \gamma_p[\mathcal{PC}_p] = st_1 & \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \rightarrow \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta \rangle}$$

Boot strapping an agent means sending the 's' special message to it. The scheduler handels executing what is necessary from here on. Initially, the *onStart()* of all agents is implemented to *unlock* their own q for receiving messages. As this may be overriden, it is important to take note of what this default implementation does and not to forget to *unlock* the queue inside the *onStart()* hook user-defined implementation.

[HAS-WORK]

The critical has-work function

$hasWork()$ is a critical function implemented by the dustributed system in order to return a set of agents that satisfy a set of predicates. These predicates are specified in our implementation as the following:

$\forall \alpha \in \mathcal{A} : size(\alpha[q]) > 0 \wedge \alpha[consuming] = true \wedge (\alpha[blocked] = false \vee \alpha[\mu.q] \implies \mu \in \{RF\})$

Its clear that system messages are processed regardless weather the agent is blocked or not. A scenario where this is important is that the agent is blocked on a future, but leter in time there is an *RF* message that resolves this future and then it is unblocked. However, this can be overrident by calling $stopConsuming()$ on that agent.

1.7 Scheduler (runtime) operational semantics rules

[PICK]

Deterministically pick a task

$$\frac{\begin{array}{l} \alpha \in hasWork() \wedge canConsume(\alpha) \\ \hdashline \mu \in \mathcal{M} \\ \hdashline Scheduler.pick(\alpha) \Downarrow \alpha[\mu.q \mapsto q] \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \rightarrow \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta \rangle}$$

The function $hasWork()$ checks all agents queues and if one/some of them has a message in the queue, it adds that agent to a set. It finally returns that set of candidate agents for scheduling. We deterministically choose to schedule one agent α out of that set to be scheduled using the function call $pick(\alpha)$ that, in turn, returns a tuple called $task = (\mu, \alpha)$.

Deterministically pick an idle agent

$$\frac{\begin{array}{c} \alpha \notin hasWork() \wedge canConsume(\alpha) \\ \hdashline \mu \in \mathcal{M} \\ \hdashline Scheduler.pick(\alpha) \Downarrow \bot \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \rightarrow ERROR}$$

Same as above except that this time we try to pick an agent whose queue is *not* populated by any message (i.e. empty) and try to get the *task* to be scheduled. Except that there isn't any message in the queue so when we try to de-queue from an empty queue, it is an ERROR.

Randomly pick a task

$$\frac{\begin{array}{c} \{\alpha_1, \cdots, \alpha_n\} = hasWork() \wedge canConsume(\alpha_1) \wedge \cdots \wedge canConsume(\alpha_n) \\ \hdashline \mu \in \mathcal{M} \\ \hdashline Scheduler.pick() \Downarrow \alpha_1[\mu.q \mapsto q] \vee \cdots \vee \alpha_n[\mu.q \mapsto q] \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \rightarrow \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta \rangle}$$

This is exactly the same as Deterministic pick that doesn't throw an ERROR above, except that it picks *randomly* from the set of agents returned by $hasWork()$. Then, it returns a $task = (\mu, \alpha)$ ready to be scheduled by the scheduler.

[TERMINATE]

The distributed system terminates in an orderly manner

$$\frac{\begin{array}{c:c} scheduler = \langle tq, \{t\}, c\rangle & \forall t \in \tau : \gamma_t \in \{rng(t.action)\} \wedge \forall st \in \gamma_t : st \notin \delta \\ \hdashline tq = \epsilon & \forall st' \in \gamma_t : st' \Downarrow \mathcal{A} \\ \hdashline hasWork() = \epsilon & \forall th \in \{t\} : isIdle(th) \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta\rangle \rightarrow TERMINATED}$$

An agent can broadcast a special message s or multiple agents can collaborate to deliver that message to all other agents, including selves. This will stop all agents cleanly and then the system will end up in a clean terminated state.

The distributed system terminates with a potential error

$$\frac{\begin{array}{c:c} \exists \alpha_1 \in \mathcal{A} & \gamma_1 \in \{rng(\mathcal{R}_1) \cup rng(\sigma_1)\} \\ \hdashline scheduler = \langle (PoisonPill, \alpha_1).tq, \{t\}, c\rangle & a \in \{\delta \cap \gamma_1\} \\ \hdashline scheduler.execute(PoisonPill, \alpha_1) \Downarrow \mathcal{A} \mapsto \mathcal{A} \setminus \{\alpha_1\} & tq \neq \epsilon \vee \exists th \in \{t\} : \neg isIdle(th) \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta\rangle \rightarrow TERMINATED \wedge ERROR}$$

If there is at least one thread that is *not* idle and/or all agents shutdown correctly but one, then the system may terminate with ERROR, i.e. not cleanly. This may lead to some data corruption by a still running thread but not able to complete the task it is executing due to un available, .e.g, a destination agent for one of its send/ask operations. More over, if one agent sent a future, for example, in response to one of the other agents' asks, and yet it didn't manage to resolve it, then that is a missing value and may lead to incorrect overall system state.

[SCHEDULE]

Scheduling a task picked for execution

$$\frac{\begin{array}{c} \alpha \in hasWork() \subseteq \mathcal{A} \wedge \mu \in \mathcal{M} \\ \hdashline scheduler = \langle tq, \{t\}, c\rangle \\ \hdashline Scheduler.schedule(\mu, \alpha) \Downarrow scheduler[tq \mapsto tq.(\mu, \alpha)] \wedge \alpha[\mu.q \mapsto q] \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta\rangle \rightarrow \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta\rangle}$$

Scheduling a task after picking it is just en-queuing that pair of message-agent at the end of the scheduler's task queue. Later to be consumed by threads in its thread pool, executing a task at a time.

[CONSUME]

Consuming a task

$$\frac{\begin{array}{c} scheduler[t.tq] \\ \hdashline Scheduler.consume() \Downarrow scheduler[t.tq \mapsto tq, cq \mapsto cq.t] \end{array}}{\langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta\rangle \rightarrow \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta\rangle}$$

Consuming is just moving the task from the front of the task queue tq to the end of the consume queue cq of the scheduler.

[EXECUTE]

Executing an already scheduled task

$$\begin{array}{c} scheduler[t.cq] \\ \hdashline Scheduler.execute(t) \Downarrow scheduler[t.cq \mapsto cq, \{t\} \setminus t_1, c \mapsto c+1] \wedge \neg isIdle(t_1) \\ \hline \langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \to \langle \mathcal{A}'[\alpha[q \mapsto q.\mu]], \mathcal{M}, \Gamma, \Delta \rangle \vee \\ \langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \to \langle \mathcal{A}'[\alpha[\mathcal{L} \mapsto \mathcal{L}]], \mathcal{M}, \Gamma, \Delta \rangle \vee \\ \langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \to \langle \mathcal{A}' \cup \{\alpha\}, \mathcal{M}, \Gamma, \Delta \rangle \vee \\ \langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \to TERMINATED \vee ERROR \vee \\ \langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \end{array}$$

Executing an action on a message reaching an agent, means executing an action on both the message, its payload, and the agents internal state. As long as the action contains an action from δ (i.e. send(). ask(), . . . , etc) then a rule should be triggered for each of those, otherwise the other statements on the action will act on the agents internal state implicitly (i.e. there is no rule to be triggered for them).

[SPECIAL-EVENTS]

executing a task with message 's'

$$\begin{array}{c} \alpha \in \mathcal{A} \wedge s \in \mathcal{M} \\ \hdashline scheduler = \langle (s, \alpha).tq, \{t\}, c \rangle \\ \hdashline Scheduler.execute(onStart(s, \alpha)) \Downarrow scheduler[(s, \alpha).tq \mapsto tq] \wedge \alpha \mapsto \alpha' \\ \hline \langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \to \langle \mathcal{A}', \mathcal{M}, \Gamma, \Delta \rangle \end{array}$$

When the scheduler is to execute one of the special tasks (those task that involve processing messages: s, j, l,rj, d, ,lf), then the scheduler executes the corresponding user overridable functions defined at that agent passing the message as parameter to those special functions. Each agent defines its own special functions: onStart(), onJoin, onLeave(), onReJoin(), onDemise(), and onLinkFailure().

Those special message to special function correspondence is as the following:

- When task = (s,α) is to be executed, $onStart(s, \alpha)$ is called by the scheduler.
- When task = (j,α) is to be executed, $onJoin(j, \alpha)$ is called by the scheduler.
- When task = (l,α) is to be executed, $onLeave(l, \alpha)$ is called by the scheduler.
- When task = (rj,α) is to be executed, $onReJoin(rj, \alpha)$ is called by the scheduler.
- When task = (d,α) is to be executed, $onDemise(d, \alpha)$ is called by the scheduler.
- When task = (lf,α) is to be executed, $onLinkFailure(lf, \alpha)$ is called by the scheduler.

These are user override-able actions and by default they are implemented to do nothing. Only the developer of a distributed system knows exactly what should happen there. However, the ability to inspect, instrument and analyze these user-provided reactions are as important as analyzing other actions done by all reactions in the behaviors defined for the distributed system.

[TIMED]

timed actions triggering

$$\frac{\begin{array}{c} t \in \tau \\ \hdashline \mathit{scheduler.tick}() \Downarrow \mathit{scheduler}[c \mapsto c+1] \wedge t[\mathit{scheduler.c} - \mathit{submitted} \geq t.\mathit{startLimit}] \\ \hdashline \mathit{scheduler.execute}(t.a, t.\alpha) \Downarrow \mathit{scheduler}[c \mapsto c+n] \wedge n \in \mathit{Int} \end{array}}{\begin{array}{c} \langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \rightarrow \langle \mathcal{A}'[\alpha[q \mapsto q.\mu]], \mathcal{M}, \Gamma, \Delta \rangle \vee \\ \langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \rightarrow \langle \mathcal{A}'[\alpha[\mathcal{L} \mapsto \mathcal{L}]], \mathcal{M}, \Gamma, \Delta \rangle \vee \\ \langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \rightarrow \langle \mathcal{A}' \cup \{\alpha\}, \mathcal{M}, \Gamma, \Delta \rangle \vee \\ \langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \rightarrow \mathit{TERMINATED} \vee \mathit{ERROR} \vee \\ \langle \mathcal{A}, \mathcal{M}, \Gamma, \Delta \rangle \end{array}}$$

Note that timed actions don't block. Timed means strictly conforming to time, in contrast to blocking which means regardless of time don't make progress.

1.7.1 Normal Use of a scheduler

Normally, a scheduler will do the following as long as there is work in the distributed system:

1. Check if there is work (using hasWork) in the distributed system
   - Possibly doing some analyses here.
2. Pick an agent (who has work to do)
   - Possibly doing some analyses here.
3. Schedule that task into the task queue
   - Possibly doing some analyses here.
4. Consume a task from the task queue
   - Possibly doing some analyses on the consume queue before and/or after it execute a task.
5. Execute a task
   - Possibly doing some analyses here.
6. Loop till there is no more work/discoveries anymore (reaching a fixed point of the context object, that is the distributed system).

It is clear now how much analysis the model allows in both the distributed system and in runtime (scheduler) stages in order to achieve the discovery of how the distributed system implementation satisfies/violates specified properties.

In our implementation, we provide more facilities in the scheduler that are already implemented and are ready for use in both analyses and implementing other schedulers. However, we keep the exact same operational semantics explained in this document through out our implementation.

1.7.2 Multithreaded scheduler

Although the operational semantics are made without regard to the multithreaded thread pool backing the scheduler (multithreaded scheduler), they are correct for multithreaded schedulers as well. Only there is some difference in details that has to be cleared here.

When the multithreaded scheduler executes multiple tasks at once, actions from these tasks *need not to* share data while at least one of them modifying the shared-memory state. This applies for agents trying to modify a shared store between them rather than relying on communication protocols to resolve what action to execute first.

We are not considering the above data races. Also, the good news, actions of agents are run as one atomic transaction, rather they interleave in a coarse grained manner instead of a statement level manner (fine grained interleaving). The bad news, however, is that if the timed actions get triggered, they interleave in a "semi-fine grained" manner with those actions. That is, those actions are no more atomic. This is also the case when a task is blocked (and hence its respective agent). This is to some extent true also for a single threaded scheduler, but the timed actions slide before or after a statement from the agent's action being executed. Contrasting it from the finer (yet still semi fine) interleaving in a multithreaded scheduler that operates with a vector-clock.

However, there are still message-induced data races when there are two or more agents with at least one of them modifying the destination agent's internal state by sending messages from these two or more agents to that destination agent.

Also, some actions may run faster than earlier actions scheduled before them for execution. This may lead to another level of data race. We are not considering these data races. There is one case where we consider a bit of interleaving, when an agent gets blocked waiting on a future then tasks coming after it from other agents certainly will execute while it is blocking.

One prominent difference, also, between a single threaded vs. multithreaded scheduler is that time is no longer a single logical clock. That clock becomes a vector clock from there on in the multithreaded scheduler.

The good news, with regards to the vector-clocks in multithreaded schedulers, is that a single threaded scheduler is easily extended to have a vector clock and then it can "simulate" multithreaded schedulers.

IMPORTANT NOTE: As a result, we opted for a single threaded scheduler, and implemnented one. We are working on extending it to simulate multithreaded scheduelrs to handel "eventual/weak consistency" models in distributed systems.

1.8 Scheduler and Agents cooperation for get and get-timeout

1.8.1 Intro

Futures and promises are better known to be conducive to efficient and elegant distributed programming. [7, 8] Implementations of these constructs in the presence of time and blocking semantics is believed to be a challenging issue, hence the need to explore how these constructs are implemented arises. However, we have first time experienced that implementing this in a purely distributed system style is notoriously hard. In this document, we describe our experience on how we implemented it.

1.8.2 Background

- Futures, promises and delay's are synchronization constructs. They act as a proxy for a result that is currently unknown but may eventually be. [8]
- They are often used interchangeably though some differences are there. [8]
  - Future is a single assignment container. [8] It also needs not know which promise will set its value, many can. [8] Future is a readonly. [8]
  - Promise is writable, single assignment container that, in turn, sets the value of a future.
- In other cases, a promise and a future are created together, the future is the value while the promise is the computation that sets the value. [8] This is similar to our implementation where a future is the return value of the promise function.
- One small difference is that the computation is NOT encoded inside the "promise" function, it is completely left to the agent promising the future to resolve it later. This gives the freedom to the agent to delegate, do the work, or delay it. That is can be as eager or as lazy as it wants resolving the future. [8]
- Setting the value of the future is called: resolving, fulfilling, or binding it. [8] We use the term "resolve" to refer to that in our work.
- The essence of Futures is that they allow programmers to write asynchronouse programs in a direct style (rather than continuation-passing-style), decoupling value (future) from how it is computed (promise).[8]
- There are a lot can be acheived with futures like chaining (reducing round trip times), composing, and exception handling. [8] Our implementation may deliver exceptions but we leave the rest of features to be implemented in a strictly distributed manner (that is in both the communicating parties loginc instead of complicating Future's implementation)
- Our implementation, also, is both explicit and blocking. That is calling future.get method will block the process if the future wasn't resolved yet, unblocked as soon as it is resolved. Also, we implement a timing-out get for our future.

1.8.3 Our Experience

During the development of our operational semantics interpreter for our generic distributed systems model, we faced several difficulties. One of the most difficult pieces of code was dealing with tracking time, blocking processes, pre-emption of executing tasks for those processes, resuming these tasks, and creating processes to act on behalf of blocked agents in certain occasions. More over, these are all centered around a purely distributed model of future creation and resolution.

To make this discussion clear, we provide some implementation details for the model we are implementing.

- What is an action: Seq of statements
- Statement: function from message received and agent receiving it to code.

First implementaiton

Setup:

- StandardExecutionThread, Agent.blocked code
- When Agent.blocked == true, block, the action maintains its latest state and used in resuming the processing once agent.blocked == false.

What is wrong? The agent stays blocked even if the get(..) times out. **Reason** The Standard-ExecutionThread checks again if the agent is blocked and even the timed action task gets blocked too! this leads to a system freez and accumulation of tasks in the blocking manager.

Second implementation with surrogates

Setup

- Same as before except that a separate processes called agent.name-timed-surrogates that marely executes a timed action after the time out is reached.
- The action to execute has the logic for both checking if the future the agent is blocking on is resolved and if not yet the wait is over, it unblocks the original agent and then disappears.
- The surrogate serves one purpose, it can't be blocked by the Standard execution thread so it executes the timed action on behalf of the blocked agent.
- The timed action is scheduled only once and then is auto-removed by the timed-actions tracker.

What is wrong? Other timed actions will be blocked, and will be blocked periodically which is worse than just being blocked. **Reason** If the original agent schedules/scheduled other timed actions with its name, all those timed actions will keep beeing triggered and immediately blocked. They also will accumulate in the timed actions tracker leading to memory leak and overall system malfunction when they get rescheduled (too many timed actions happening at once). **

Third implementation with non-contradicting timed vs blocked semantics

Setup

- Same as above but additionally, mark the timed actions with a flag so that the standard execution thread checks it as part of its blocking logic.
- Why we kept the **surrogate**, we need to defferentiate between runtime-created timed actions and agent-created timed actions when we investigate the distributed systems runtime state.
- If the agent is NOT blocked OR the task is timed, continue executing the action. Otherwise, block the task, agent and action being executed.

Seems to work Reasoning Blocking and timed-actions are contradictory properties of distributed systems. If something is timed, it needs to conform to time in the most strict possible way. If that action is blocked, it means regardless of time passing by, it won't progress. So if an action is timed, it can't be at the same time blocked, and vice versa. They are mutually execlusive and an action is either timed or blockable. Allowing timed actions to be blocked is against the semantics of timed actions in the first place, so we coded logic to ignore any blocking on a timed action.

1.8.4 Related work and comparison

- Promises [7] are used to both make different processes collaborate in achieving the work (the deferreds) and as a result to resolve the future.
- Deferreds [7] also allow to know what work has been done and if the future is resolved that future is returned instead of doing repetitive work. This is not to be mixed that multiple workers may retry (in case an error occured) to process deferreds simultaneously nor with workers may start to work in parallel to do the same job. They allow both.
- They require a central authority to track who is doing what and if the deferreds has been completed. After the central locking authority has locked those deferreds to a worker, the promise broadcast a completion message to the peers observing (And waiting) for the future to be resolved. [7] Essencially composing queiries from partial queiries and enabling additional queiries to hook into them in order to reduce work done by the Mongo DB [7]. As a result re-using these partial queiries and their results.
- Ours is completely distributed (peer-to-peer) future implementation where deferreds are completely left to the worker and/or its delegates.
- Ours can compose as well and can deliver exceptions as a value of the future.
- Ours however doesn't implement logic to handle exceptions and retry in the future, it leaves that to communicating parties to decide what to do in each case.
- Our future is merely the basic synchronization construct with value contained and that is completely independent from any central authority.

1.9 Operational Semantics Interpreter (runtime)

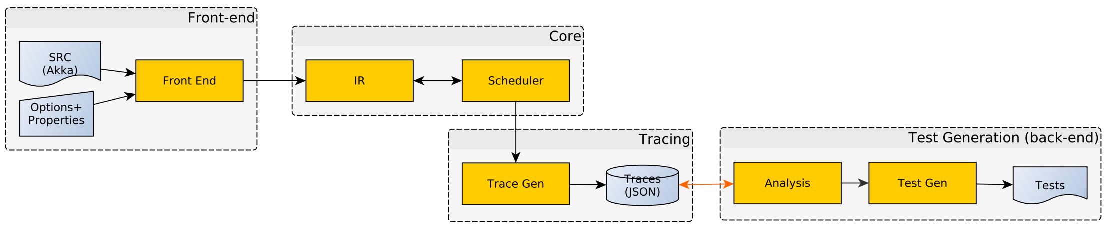

Figure 1.2: Architecture of our tool

In this section, we will talk about the core of our analysis tool that is composed of the following (also shown in figure 1.2):

- **Front end:** takes Akka actors code in byte code level and extracts the model in our IR.
- **Intermediate Representation:** All structures talked about in the operational semantics regarding Distributed Systems and Agents.
- **Schedulers:** A basic scheduler along with other more advanced ones to explore traces generated from the model. Smart schedulers try to maximize hitting an erroneous situation and generating a test case that replays that, as well as outputting the trace that lead to that test case.Later we run queries and property checking on those traces.
- **Tracing**: it is used to generate traces by the scheduler on the data structures of the core.
- **Test Generation**: Tests generator, that generates tests on detecting a trace that violates a property, in order to facilitate auto-testing of distributed systems.

1.9.1 Benchmarks

We developed 3 benchmarks to form the basis of our analysis (the input to our front end). These benchmarks are as the following:

- **OpenChord:** [9] a distributed hash table originally called Chord, later an open source implementation called Open Chord was developed. We ported it to Akka Actors. We have 2 different implementation, one of them is purely developed in Scala and another in mixed scala/java.
- **MultiPaxos:** [10, 11, 12] the consensus protocol proposed by Lamport. We developed it in Akka Actors.
- **ZAB:** [13] the primary server replication protocol (atomic broadcast) developed for Apache's Zookeeper [14].We ported it to Akka Actors.
- **Raft:** [15] a simpler more modular equivalent to Paxos. We found an implementation on github in Akka Actors. We try to run our tool through out-of-house implementation to see how it performs.

All of which are available in Github as of today. They are free to use with their respective original licenses restrictions.

1.9.2 Completion to date April 19, 2016

- **Front end:** early stage development.
- **IR:** implemented.
- **Scheduler:** implemented
  - **Deadlock Detection Scheduler**: implementing soon
- **Tracing:** implemented.
- **Test Generation:** not implemented yet.
- **snapshot runtime state and resume:** testing and debugging
- **snapshot runtime state and resume operational semantics:** ready, need be typeset

Chapter 2

Distributed Systems Runtime Snaphsot and Restore

2.1 Intro

This document describes the operational semantics of the *copy* and *link* pair of operations for each data structure. These operations are crucial in making backtracking algorithms as well as saving then restoring the *whole runtime state.* These operations are *not* intended for persisting the runtime and/or distributed system state although it can simply be done by serializing the acyclic state copy then de-serializing and linking it back on resume. However, the copy/link operation is intended, beyond persistence, to snapshot and restore the whole runtime state to an earlier execution state then resume executing. That is, it is for restarting from some point of time.

2.2 How it works in high level

At any point of time during execution of a distributed system, a runtime (including both the scheduler and the distributed system) state is snapshot. The snapshot is done by calling a method of the scheduler used to run the distributed system. The scheduler proceeds in execution after taking that snapshot doing any kind of work and/or analysis.

At the time the scheduler decides to backtrack to a previous state, without losing the traces stored in the trace manager, it calls its own method for restoring the state.

The whole runtime goes back in time as if nothing has happened except that traces stored in the trace manager of the scheduler are maintained as is. That is, the traces are maintained because that's the point of an exploring (analyzing) scheduler.

The process over the whole runtime system is two-split. The scheduler first snapshots the distributed system it is running, then it snapshots its own state. Finally it stores everything in a `SchedulerState` object to be restored later.

The copy is used to generate a snapshot of each data structure (de referencing cycle-causing data structures of the distributed system, yet storing their identifiers). The time the distributed system is to be used (after restoring the state to the incomplete snapshot), the `link` operation is called on the distributed system and scheduler in order to restore these data structure from the snapshot. Rather than older statements for keeping referring to the old distributed system's agents, the statements get updated to refer to the agents of the NEW (i.e. snapshot's) distributed system agents. We called this runtime state snapshot/restore, basically `copy` followed by `link`, two-phase copy.

To make this clear, copying and linking first copies and links, respectively, the runtime in two-splits (starting with distributed system, then with scheduler's state). The whole process is called two-phase because there is a copy phase, followed by a link phase.

2.2.1 The snapshot/copy

A snapshot is taken from the runtime, leaving a lot of cyclic references to some data structures out from the snapshot, and only maintaining an identifier of that structure in the corresponding data structure being copied. For example, a message that refers to agent 'a1' when copied, its `copy` operation copies all "terminal" fields of the message, e.g. its name as a string, and stores only an identifier of the agent 'a1', which is its name as a string. Leaving the responsibility of copying the acyclic state of the agent to agent 'a1' itself, that is to a1.copy().

The purpose, hence, of the copy operation: at the time of restoring a state, instead of that message keeping referring to an agent from an old distributed system copy, the call to link can use the identifier to refer to the new system's agent copy. This propagates through out the whole distributed system agents, actions, ...etc down to each *instrumented* statement. An instrumented statement is any statement not of kind "NONE".

2.2.2 The linking

Completing our example from the copy operation above, a message is now to be linked with a new distributed system. The link operation of that message when called, should update the field of that message according to the identifier stored in the copy, yet by making it refer to the new distributed system's agent with the same identifier (since we know already that this agent exists in both the original and the copy of the distributed system).

The same thing happens to all data structures starting from the containing `DistributedSystem` down to smallest contained object in the "cyclic graph" of interconnected objects. Note that we say graph since it is NOT a tree, with respect to which object refers to which another.

The purpose of the link operation: To update all references of all data structures to refer to the new distributed system's objects (e.g. agents). Also, to link a scheduler's suspended tasks, timed tasks, ...etc to the new distributed system's objects.

2.3 Motivating Example

Suppose we have a statement that is supposed to send a message (in distributed system 'ds1') from agent "src" to agent "dst". In the original distributed system (i.e. before any copy operation), it will work just fine.

Now suppose we copied this distributed system and lets call the copy “ds2”. If we are to copy the statement, its “code” field is a function whose body is the send statement. We simply can’t copy those objects even by serializing then de-serializing to/from byte streams since this will create infinite loops (due to distributed system referred objects by the body of the function, according to our experience). So, we will just refer to the old code, assuming we don’t have implementation for any copy operation. This will result in so many problems clearest of them is that when restoring to the “ds2” state, the send operation will send nothing to “ds2.dst”, instead it will send to “ds1.dst”. More over the sender (assuming “ds2.dst” is to reply to the “ds2.src”), this will not work either. The reply will go to “ds1.src” instead of going to “ds2.src”.

The above clearly violates the semantics explained in our DS2 operational semantics TR. [REFERENCE HERE]

Worse yet, since DS2 is implemented in scala, nothing can be changed in the `statements.code` body by scala’s experimental reflections API, neither compile time nor runtime API’s since all we get is a reference to that function not the whole AST of it. [REFERENCE the discussion here]. It is said some libraries may provide this ability in the future but so far we came up with our own solution.[DISCUSS later the Statement how it wraps that code and generates the function body dynamically instead and yet we can change references]

2.4 Implementation specific data structures

In our DS2 TR we introduced the abstract operational semantics data structures. In this section, we introduce our implementation specific data structures. They are same data structures in the TR except with additional attributes that enable the discussed operations in this document.

To help keep this document short, we will not re-introduce the data structures already discussed in the TR. Instead, we will introduce the *additional attributes* of each data structure *that relates to our copy-link operations* and will assume the reader knows the remaining from the TR. If not, we refer the reader to first read the abstract DS2 operational semantics TR. We show Figure 2.1 to remind the reader of what’s there in the data structures and how they relate to each other before a lot of the additional attributes discussed in the next section. This figure will be referred to very frequently, so it is advisable for the readers to familiarize themselves with its content.

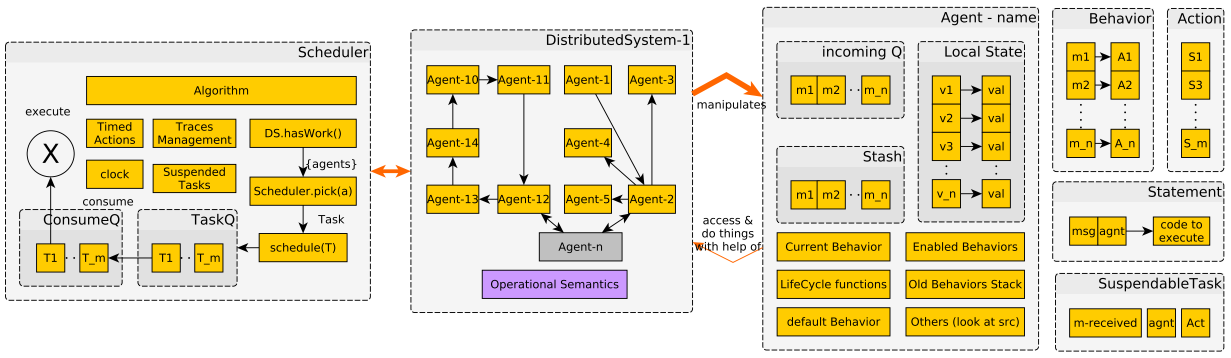

Figure 2.1: Simplified view of data structures and how they relate to each other

2.5 Types and Universes

$\mathcal{K}$: the statement's kind and all possible values are: {NONE, SEND, ASK, CREATE, START, STOP, KILL, LOCK, UNLOCK, STOP_CONSUME, RESUME_CONSUME BECOME, UNBECOME, STASH, UNSTASH, UNSTASH_ALL GET, GET_TIMED, BOOT_STRAP, BOOT_STRAP_ALL, MODIFY_STATE MODIFY_STATE_REF, IF_ELSE, WHILE, FUNCTION}

C : the "code" data type which is $\mathcal{M} \times \mathcal{A} \rightarrow \mathcal{K} \times B$

B : all executable code, that is a block of executable statements. In c-like languages, it is either a basic block or a single syntactic statement. In our case, it is a scala block of statements.

$\mathcal{D}$: the universe of all distributed systems.

$\mathcal{P}$: the universe of all programs consisting of executable statements. It includes both the host language statements (in our case Scala) as well as our IR data structures and schedulers statements. So, when we say $st \in \mathcal{P}$, it means that st is a generic statement, while if we say $st \in \mathcal{K} \setminus NONE$ it means that it is one of our distributed system's specific statements.

NOTE: NONE kind means it is "none of the other" kinds of statements that we care about in distributed systems analysis such as: SEND, ASK, ...etc. All other kinds meaning is clear from their names.

2.6 Auxiliary functions

- **link(ds, ir)**: the link function that links an "ir" data structure to a new distributed system "ds"
- **copy(ir)**: the copy operation that copies an "ir" data structure and returning an incomplete copy of it to be linked to another Distributed system. Incomplete means some of its fields is not a copy but a reference to the original data structure's fields e.g. the same agent from the old distributed system that owned the original of the data structure.

- **getSet(ds,agentName,ir,fieldName)**: Finds an agent $\alpha \in ds[\mathcal{A}]$ whose name is `agentName` and sets the field `fieldName` of the data structure `ir` to the found agent α.

- **generate(st)**: a function that checks the statement's "st"'s kind and generates code to do that kind of operation. e.g. if it is of kind SEND, it generates the code: that does a send operation according to parameters set inside the statement w.r.t. the containing distributed system context object.

- **executedLast($act \in \mathcal{A}$)**: returns the last statement executed in the action passed in the argument *act*.

- **saveState(sch)**: returns a copy of the scheduler `sch` state.

- **restoreState(sch,state)**: restores the scheduler's `sch` state to the saved state `state`. Scheduler's states contains the distributed system state. So, it is both saved and restored as part of the scheduler's state saving/restoration.

The actual behavior of each data structure's copy/link operation depends on the type of the "ir" argument.

In addition, all of the above auxiliary functions can happen any where in our IR, scheduler's behavior, or in $\mathcal{P}$. Once the function is executed then the correspondent rule of the IR is invoked according to the type of the IR.

2.7 Distributed System state

Agent

- **q**: Incoming messages queue.

- θ: the default behavior of the agent

- $\mathcal{S}$: delayed messages buffer.

- $\mathcal{R}$: the current behavior of the agent.

- $\mathcal{B}$: a set of enabled behavior names mapped to behaviors this agent can become.

- σ: the behavior mapping the default special messages (e.g. start, join, rejoin, ...etc) to their actions.

- τ: the set of timed actions associated with this agent, that is scheduled periodically by this agent.

- β: a stack containing the names of older exhibited behaviors. This is used by *unbecome* to revert back to latest old behavior.

- **lo**: indicates if the incoming queue is locked, i.e. doesn't allow incoming messages from other agents. By default set to `true`.

- **b**: indicates if this agent is blocked (i.e. has a blocking task in the blocking manager) so it can't schedule any more tasks. By default it is set to `false`.

- **bo**: If `blocked` field is set to `true`, this field *must* point to the future this agent is blocking on.
- **c**: indicates that the scheduler is allowed to schedule tasks from the agent's incoming messages queue. By default it is set to `true`
- $\mathcal{L}$: a map from variable names to the most generic type (Any in scala) values representing the local state of the agent (i.e. variables declared, used, modified, ...etc).
- **fp**: the set of futures promised by this agent.
- **fw**: the set of futures other agents promised this agent.

Behavior

- **bn**: behavior name.
- **a**: The agent this behavior is associated with, a reference to the original agent.
- **an**: holds the owning agent name of this behavior (the one currently using this behavior object).
- $\mathcal{R}$: a set of pairs $f : \mathcal{M} \rightarrow \Gamma$ with each pair mapping a message to an action.

Action

- **m**: The message received, and reacted upon, by the agent to apply this action as a reaction.
- **a**: The agent owning this action, and whose state maybe modified by statements of this action.
- **an**: The name of the agent to be used in the "link" operation to the new distributed system.
- **stmts**: Statements composing this action.

Timed Action Note that there are more than the following fields. But since they are just copied and not linked, they are left out. All of the left out fields are simply (big) integers. We will replace all of those with a single component called `t`, stands for time, and another called `c` for the count of times the timed action will be triggered.

- **a**: The agent that scheduled the timed action
- **an**: the name of the agent that scheduled the timed action, to be used in the "link" operation to the new distributed system.
- **act**: The action to be triggered once the countdown reaches zero.
- **t**: The time after which this timed action will be triggered periodically.
- **c**: the number of times this timed action must be triggered, that is after how many time-periods passed.

Statement A statement is a wrapper for executable code, its kind, and many other instrumentation info (shown below) that we used to copy and then link statements. It also has auxiliary functions to re-generate code that is of kind $\mathcal{K} \setminus NONE$.

The following are the instrumentation info dealt with during the copy-link operation:

- **code**: The code to be executed by the `apply` method of Statement. It is a lambda of the form $\mathcal{M} \times \mathcal{A} \rightarrow \mathcal{K} \times B$
- **m**: The message received by the agent and set by the action containing this statement to act upon.
- **a**: The agent who owns this statement, in turn the action housing this statement, and whose state maybe modified by this statement (in case its kind is MODIFY).
- **an**: The name of the agent to be used in the "link" operation to the new distributed system.
- **k**: The kind of a statement. By default it is NONE, unless an instrumentation statement constructor(s) was used.
- **mo**: The message *possibly* sent by this statement in case it is a SEND or ask for example.
- **sa**: The src agent in case for example the statement is a SEND .
- **san**: The name of the src agent to be used in the "link" operation to the new distributed system. In case the statement kind is SEND, ASK, or LOCK for example.
- **da**: The dst agent in case for example the statement is a SEND .
- **dan**: The name of the dst agent to be used in the "link" operation to the new distributed system. In case the statement kind is SEND, or ASK for example.
- **f**: A reference to a future in case the statement is of kind ASK, GET, or GET_TIMED.
- **to**: timeout, Used in GET_TIMED instrumented statement.
- **bn**: the name of behavior in case the statement is BECOME, or UNBECOME.
- **r**: true or false to enable/disable BECOME statements to/from remembering the old behavior.
- **bs**: A set of names of agents that has been boot-strapped in case the statement is either BOOTSTRAP or BOOTSTRAP_ALL.
- **var**: The variable to be modified in case the statement is of kind MODIFY.
- **val**: The to be held by the variable indicated in "variable" field.
- **act**: In case of a GET, GET_TIMED the action being executed maybe suspended, this reference is used to hold that suspend-able action.
- **ds**: The distributed system this statement is associated with. This is necessary after copying a statement and if its kind is not NONE, its code will be regenerated and will point to actors and entities from the new distributed system. This field is referenced at the time of "copy" and is updated to the new distributed system at the time of "link" operation.
- **ins**: `false` if the statement kind is NONE, true otherwise. stands for "instrumented".

Message

- **s**: The agent that send this message (sender)
- **sn**: The name of the agent that sent this message, this field is used to re-point to a new distributed system's agent whose name is the same as the "sender". This style of updating the agents references is used throughout the IR data-structures.
- **pl**: the payload of the message, a sequence of objects/values/mix.

Future

- **pb**: The agent that promised this future (promised-by)
- **pbn**: the name of the agent that promised this future, used in the "link" operation.
- **wf**: The agent that *may* wait, i.e. potentially blocking temporarily or permanently, for this future to be resolved. Statements that normally wait for such future objects to be resolved are either GET or GET_TIMED statements.
- **wfn**: The name of the agent that may wait for this future to be resolved, this field is used in the "link" operation.
- **r**: `true` if the future was resolved (indicating that the value stored in the `val` field is valid. `false` otherwise.
- **val**: the value resolved, can be any value/object.

Task

- **a**: The agent that scheduled this task for execution.
- **an**: The name of the agent that scheduled this task for execution.
- **act**: The action to be executed by this task.

2.8 Scheduler State

Blocking Tasks Manager

- **ds**: The distributed system whose tasks are being blocked by this blocking manager.
- **sch**: The scheduler associated with the distributed system whose tasks are being blocked by the blocking manager.
- **bt**: A list of tasks blocking in the distributed system, always on a future to be resolved. It is good to remind that all of such blocking tasks are not timed actions as this violates the definition of a "timed" action, which means strictly conforming to time.

Timed Actions Tracker

- **ta**: the set of timed actions to be triggered on their countdown reaching Zero.

Task Queue

- **tQ**: The queue of the tasks scheduled by agents into the scheduler.

Consume Queue

- **cQ**: The tasks pulled from the `taskQ` and are ready for execution by the thread(s) backing the scheduler.

Note that we have a construct called "SchedulerState" that we are detailing next. We are having it as a separate construct although we have it in the scheduler because a scheduler's saved state stays constant, while a scheduler's live state keeps evolving. Think of it as a boxed and sealed state object to keep the snapshot of the scheduler's state untouched and unchanged.

SchedulerState Since non of the above scheduler state can exist by itself, it must be encapsulated in a data structure, called `SchedulerState` to be stored and restored. So this construct has all the following fields that encapsulates the whole runtime state:

- **ds**: the distributed system this scheduler is associated, or to be associated, with.
- **clk**: The `clk` scheduler field copy, basically the clock of the scheduler is a big int.
- **tQ**: The task queue of the scheduler, whose tasks must be linked to this scheduler state's `ds` field.
- **cQ**: The consume queue of the scheduler, whose tasks must be linked to this scheduler state's `ds` field.
- **bMgr**: The blocking manager of the scheduler, whose tasks and futures they are blocking on must be linked with `ds` and its agents.
- **tat**: The timed actions tracker of the scheduler. It contains all timed actions that are scheduler by agents and are to be triggered at least once according to some time interval, based on the scheduler's `clk` field and some internal counters.

2.9 Conventions

- **Original**: we use normal abbreviations such as s to indicate the original *SchedulerState* object.
- **Non-linked copies**: we use s' to indicate the non-linked copy of the original state s.
- **Linked copy**: we use s'' to indicate the linked copy of the state copy s'. That is in turn a copy of the original state s.

The above conventions applies to any other data structure that we can copy and then link such as timed actions tracker, timed action, agent, . . . etc.

Also, We say $s \in SchedulerState$ to say *s is a scheduler state data structure/type*, and similarly to other types/data-structures.

2.10 Runtime State Copying in detail

While copying a distributed system's and runtime state, we need to do a two split two phase copy. The two phases of rutnime copy are copy, then link. The copy operation makes incomplete copies, while the link operation completes that copy. Two split means the distributed system is first copied, and internally linked. Then the Scheduler state is copied then linked with the already linked distributed system copy, mentioned previously.

In this section, and its two subsections, we will detail the operational semantics of the two split copy, then in the next section we will detail the two-split link operation. Each subsection deals with copying a distributed system, then copying a scheduler's state. Similarly, the link in the next section deals with linking the distributed system, then linking the scheduler to that distributed system.

2.10.1 IR Data-structures State

[Distributed System]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ ds \in \mathcal{D} \\ st = copy(ds) \\ st \Downarrow \forall \alpha \in ds[\mathcal{A}], \mu \in ds[\mathcal{M}], \gamma \in ds[\Gamma], \delta \in ds[\Delta] : \\ copy(\alpha) \Downarrow ds'[\mathcal{A} \mapsto \mathcal{A} \cup \{\alpha'\}] \\ copy(\mu) \Downarrow ds'[\mathcal{M} \mapsto \mathcal{M} \cup \{\mu'\}] \\ copy(\gamma) \Downarrow ds'[\Gamma \mapsto \Gamma \cup \{\gamma'\}] \\ copy(\delta) \Downarrow ds'[\Delta \mapsto \Delta \cup \{\delta'\}] \end{array}}{\mathcal{D} \mapsto \mathcal{D} \cup \{ds'\}}$$

Description: If a copy call is encountered whose argument is a distributed system `ds`, and then executed. The effect of it is producing a new copy of that distributed system, call it ds', by copying all of the earlier agents, messages, actions, and behaviors.

[Agent]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ ds \in \mathcal{D} \\ \alpha \in ds[\mathcal{A}] \\ st = copy(\alpha) \\ st \Downarrow copy(\alpha[b]) \Downarrow \alpha'[b \mapsto b'] \wedge \\ copy(\alpha[lo]) \Downarrow \alpha'[lo \mapsto lo'] \wedge \\ copy(\alpha[c]) \Downarrow \alpha'[c \mapsto c'] \wedge \\ \forall \mu \in \alpha[q] : copy(\mu) \Downarrow \alpha'[q \mapsto q.\mu'] \wedge \\ copy(\alpha[\theta]) \Downarrow \alpha'[\theta \mapsto \theta'] \wedge \\ \forall \mu \in \alpha[\mathcal{S}] : copy(\mu) \Downarrow \alpha'[\mathcal{S} \mapsto \mathcal{S}.\mu'] \wedge \\ copy(\alpha[\mathcal{R}]) \Downarrow \alpha'[\mathcal{R} \mapsto \mathcal{R}'] \wedge \\ copy(\alpha[\mathcal{B}]) \Downarrow \alpha'[\mathcal{B} \mapsto \mathcal{B}'] \wedge \\ copy(\alpha[\sigma]) \Downarrow \alpha'[\sigma \mapsto \sigma'] \wedge \\ \forall b \in \alpha[\beta] : \alpha'[\beta \mapsto \beta.b] \wedge \\ copy(\alpha[bo]) \Downarrow \alpha'[bo \mapsto bo'] \wedge \\ \forall t \in \alpha[\tau] : copy(t) \Downarrow \alpha'[\tau \mapsto \tau \cup \{t'\}] \wedge \\ copy(\alpha[\mathcal{L}]) \Downarrow \alpha'[\mathcal{L} \mapsto \mathcal{L}'] \wedge \\ \forall f \in \alpha[fp] : copy(f) \Downarrow \alpha'[fp \mapsto fp.f'] \wedge \\ \forall f \in \alpha[fw] : copy(f) \Downarrow \alpha'[fw \mapsto fw.f'] \end{array}}{ds'[\mathcal{A}'] \to ds'[\mathcal{A}' \cup \{\alpha'\}]}$$

Description: To copy an agent α and produce another α', later still needs to be linked, we need to *custom-copy* each of the non-terminal constructs introduced by our distributed systems semantics that are in α. Other, terminal, constructs such as language specific booleans, integers, strings, . . . etc are copied by the runtime. There is one possibly confusing part of the latter examples, the local state map of the agent $\mathcal{L}$. Since there isn't a copy constructor nor a rule in our operational semantics, this is serialized and de-serialized to make a copy. We assume no one will use but *strings* as keys, and strings are immutable, and only *terminal values* for values such as *name of the agent* instead of a direct reference to that agent. This way we avoid infinite copy cycles of copying two non-terminal constructs referring to each other. Note that futures promised by the agent or the agent is waiting for are handled in a different storage, namely in fp and fw and are copied using the *custom-copy* constructor for futures.

Note: note that specific *custom-copy* constructors are called according to the *type-of* the argument to copy. For example, the copy constructor of agents is called when the argument to the *copy(...)* is of type $\mathcal{A}$ and so on.

[Behavior]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ b = (a, an, bn, \mathcal{R}) \in rng(\Delta) \\ st = copy(b) \\ st \Downarrow b'[a \mapsto b[a], an \mapsto b[an], bn \mapsto b[bn]] \wedge \\ copy(b[\mathcal{R}]) \Downarrow \forall r = (\mu, \gamma) \in b[\mathcal{R}] : \\ copy(\mu) \Downarrow r'[\mu \mapsto \mu'] \wedge copy(\gamma) \Downarrow r'[\gamma \mapsto \gamma'] \wedge b'[\mathcal{R} \mapsto \mathcal{R} \cup \{r'\}] \end{array}}{ds' \rightarrow ds'[\Delta \mapsto \Delta \cup \{(b'[bn], b')\}]}$$

Description: Copying a behavior b to produce another b', requires the copying of its *reaction pairs* $\mathcal{R}$, and referencing the remaining fields: `a`, `an`, and `bn`. Where `an` and `bn` are strings standing for *agent name* and *behavior name*, respectively, and the field `a` is the *agent* owning this behavior. Since the agent can't be copied but by its own copy method, the field `an` indicates which agent from the new distributed system copy ds' can be referenced upon linking.

[Action]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ \gamma = (m, a, an, stmts) \in ds[\Gamma] \\ st = copy(\gamma) \\ st \Downarrow \gamma'[a \mapsto \gamma[a], an \mapsto \gamma[an]] \wedge \\ copy(\gamma[m]) \Downarrow \gamma'[m \mapsto m'] \wedge \\ \forall s \in \gamma[stmts] : copy(s) \Downarrow \gamma'[stmts \mapsto stmts.s'] \end{array}}{ds' \rightarrow ds'[\Gamma \mapsto \Gamma \cup \{\gamma'[stmts \mapsto stmts']\}]}$$

Description: In order to copy an action, the fields `a`, and `an` has to be linked just like in copying the behavior. The message by which this action was triggered is also copied. However, `stmts` need to be copied one by one in the same order to the fresh copy of an action. Copying statements comes next after copying timed actions.

[Timed-Action]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ \alpha \in ds[\mathcal{A}] \\ ta \in \alpha[\tau] \\ ta = (m, a, act, t, c) \\ st = copy(ta) \\ st \Downarrow ta'[a \mapsto ta[a], an \mapsto ta[an], t \mapsto ta[t], c \mapsto ta[c]] \wedge \\ copy(ta[m]) \Downarrow ta'[m \mapsto m'] \wedge \\ copy(ta[act]) \Downarrow ta'[act \mapsto act'] \end{array}}{ds'[\mathcal{A}[\alpha'[\tau \mapsto \tau \cup \{ta'\}]]]}$$

Description: Just like an action, but with few additional fields as a wrapper object called `TimedAction` binding them together, `a` and `an` are referenced while `m` and `act` are copied using their respective copy constructor.

[Statement]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ \gamma \in ds[\Gamma] \\ s \in \gamma[stmts] \\ st = copy(s) \\ st \Downarrow ((s[k] = NONE \wedge s'[code \mapsto s[code]]) \vee (s[k] \in \mathcal{K} \setminus \{NONE\} \wedge generate(s) \Downarrow s'[code \mapsto code']))\wedge \\ s'[a \mapsto s[a], an \mapsto s[an], k \mapsto s[k], sa \mapsto s[sa], san \mapsto s[san], da \mapsto s[da], dan \mapsto s[dan], to \mapsto s[to], \\ bn \mapsto s[bn], var \mapsto s[var], ds \mapsto s[ds]]\wedge \\ copy(s[r]) \Downarrow s'[r \mapsto r']\wedge \\ copy(s[val]) \Downarrow s'[val \mapsto val']\wedge \\ copy(s[ins]) \Downarrow s'[ins \mapsto ins']\wedge \\ copy(s[mo]) \Downarrow s'[mo \mapsto mo']\wedge \\ copy(s[f]) \Downarrow s'[f \mapsto f']\wedge \\ copy(act) \Downarrow s'[act \mapsto act']\wedge \\ copy(s[m]) \Downarrow s'[m \mapsto m']\wedge \\ \forall n \in s[bs] : s'[bs \mapsto bs.n] \end{array}}{ds' \rightarrow ds'[\Gamma[\gamma'[stmts \mapsto stmts.s']]]}$$

Description: Copuing a statement leaves most of the fields referencing the old statement's fields except for some. The custom copied ones include: `mo`, `f`, `act`, `m`, and `code`. The remaining are taken care of in the link operation. Note that there are two partitions of statements kinds, there are the `NONE` kind. Those are referenced in the copy as is without touching them. While others that have their kind as not `NONE`, that is $k \in \mathcal{K} \setminus NONE$, get regenerated to reference entities encapsulated inside of it from the new distributed system copy, namely those from ds'. The way those statements get regenerated is discussed in §2.11.

Note: that we are talking about some action in the distributed system's actions set $ds[\Gamma]$. However, in reality, these actions can live inside behaviors, actions inside an agent reactions, ...etc. The reason why we are only referring to distributed system's actions is to simplify the rule and make it more readable since copying a statement doesn't differ no matter where it was since it will end up in the corresponding distributed system's copy. That is if it was in an action inside an agent's default reaction, it will end up in that agent copy's default reaction's exact place (which is by the way an action too). Our statements never live but inside an action's sequence of statements.

[Message]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ \mu \in ds[\mathcal{M}] \\ st = copy(\mu) \\ st \Downarrow \mu'[s \mapsto \mu[s], sn \mapsto \mu[sn]] \wedge \\ copy(pl) \Downarrow \mu'[pl \mapsto pl'] \end{array}}{ds' \rightarrow ds'[\mathcal{M} \mapsto \mathcal{M} \cup \{\mu'\}]}$$

Description: A copy μ' of the message μ is produced by linking the same agent a, copying its name an and copying the payload pl. The latter is a sequence of terminal values that can not contain any DS2 specific data structures, just like the agent's local state map $\mathcal{L}$. So, pl is serialized then de-serialized to/from a byte stream to make a copy, since it is not allowed to contain cyclic references that may cause infinite loop during serialization.

Again, for simplicity we show what will change only in a distributed system's collection of declared messaging. A message, just like statements, can live any where inside the agent's entities: Agent, Behavior, Action, TimedAction, Task, ...etc.

[Future]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ f \in \mathcal{F} \\ st = copy(f) \\ st \Downarrow f'[pb \mapsto f[pb], pbn \mapsto f[pbn], wf \mapsto f[wf], wfn \mapsto f[wfn]] \wedge \\ copy(f[r]) \Downarrow f'[r \mapsto r'] \wedge \\ copy(f[val]) \Downarrow f'[val \mapsto val'] \end{array}}{\mathcal{F}' \mapsto \mathcal{F}' \cup \{f'\}}$$

Description: In order to copy a future f to produce another f', all fields are referenced from the old future except for terminal values (`val`, and `r`), since the latters change over the life time of a future.

Note: We are referring to the set of futures in a distributed system $\mathcal{F}$ instead of where exactly that future is copied from and to. This is becaue it may live in many different places at the same time: in blocking manager, in `fp` field of agent, and in `fw` field of agent.

[Task]

$$\frac{\begin{array}{c} t \in \{(a, an, act)\} \\ ss \in SchedulerState \\ st \in \mathcal{P} \\ st = copy(t) \\ st \Downarrow t'[a \mapsto t[a], an \mapsto t[an]] \wedge \\ copy(act) \Downarrow t'[act \mapsto act'] \end{array}}{ss \rightarrow ss[tQ \mapsto tQ.t'] \vee ss[cQ \mapsto cQ.t'] \vee ss[btm[bt \mapsto bt \cup \{t'\}]]}$$

Description: A task can only exist in the context of a scheduker state. When copied, it will as well reside in a scheduler's state. It will either be in a task queue tQ, consume queue tQ, or blocking tasks bt in a blocking tasks manager.

2.10.2 Scheduler State

In this section, we will firest detail the copying of each component if a scheduler's state. Then, we will put all of that togetehr in the "Scheduler State" copy rule. This helps keeping the rules appear more readable, clearer and simpler to understand. However, in each of the component's copying rules, we will still assume that component exists inside of a scheduler state `ss` as they can't exist standalone otherwise.

[Blocking-Tasks-Manager]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ ss \in SchedulerState \\ btm = ss[btm] \\ btm = (ds, sch, bt) \\ st = copy(btm) \\ st \Downarrow btm'[ds \mapsto btm[ds], sch \mapsto btm[sch]] \wedge \forall r \in btm[bt] : copy(t) \Downarrow btm'[bt \mapsto bt \cup t'] \end{array}}{ss' \rightarrow ss'[btm \mapsto btm']}$$

Description: To produce a copy of a blockint tasks manager, we need to reference the old ds (for now) and reference the old scheduler and to copy each blocking task in the blocking tasks set `bt`. Copying each blocking task triggeres the "Task" copying rule explained above.

[Timed-Actions-Tracker]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ ss \in SchedulerState \\ tat = ss[tat] \\ tat = (ta) \\ st = copy(tat) \\ st \Downarrow \forall t \in tat[ta] : copy(t) \Downarrow ss'[tat[ta \mapsto ta \cup t']] \end{array}}{ss' \rightarrow ss'[tat \mapsto tat']}$$

Description: Copying a timed actions tracter, which doesn't live but inside a scheduler's state, requires us to copy all timed actions stored in its ta.

[Task Queue]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ ss \in SchedulerState \\ st = copy(ss[tQ]) \\ st \Downarrow \forall t \in ss[tQ] : copy(t) \Downarrow ss'[tQ \mapsto tQ.t'] \end{array}}{ss' \rightarrow ss'[tQ \mapsto tQ']}$$

Description: To copy a task queue, each task that resides in that queue needs to be copied individually from the original task queue to the scheduler state copy's task queue $ss'[tQ]$.

[Consume Queue]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ ss \in SchedulerState \\ st = copy(ss[cQ]) \\ st \Downarrow \forall t \in ss[cQ] : copy(t) \Downarrow ss'[cQ \mapsto cQ.t'] \end{array}}{ss' \rightarrow ss'[cQ \mapsto cQ']}$$

Description: Like a task queue copy, except for a consume queue copy. Each task needs to be copied from the original to the scheduler state copy's consume queue $ss'[cQ]$.

[Scheduler State]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ ss \in SchedulerState \\ ss = (ds, clk, tQ, cQ, cQ, bMgr, tat) \\ st = copy(ss) \\ st \Downarrow copy(ss[clk]) \Downarrow ss'[clk \mapsto clk'] \wedge \\ copy(ds) \Downarrow ss'[ds \mapsto ds'] \wedge \\ copy(tQ) \Downarrow ss'[tQ \mapsto tQ'] \wedge \\ copy(cQ) \Downarrow ss'[cQ \mapsto cQ'] \wedge \\ copy(bMgr) \Downarrow ss'[bMgr \mapsto bMgr'] \wedge \\ copy(tat) \Downarrow ss'[tat \mapsto tat'] \end{array}}{SchedulerState \mapsto SchedulerState \cup ss'}$$

Description: To copy a scheduler state ss to produce another ss', we need to copy all of its fields according to each component's copy rule, components that the scheduler state encapsulates.

2.11 (Re-) Generating Statement's Code

Copying the `code` field of a statement is, actually, a regeneration of that code in case the kind of the statement is $\mathcal{K} \setminus \{NONE\}$. So, the $copy(st)$ has a call to $generate(st)$. That generate statement is detailed in this section. Specifically, this is shown in table 2.2.

2.12 Runtime State linking in detail

In this section, we focus on linking the Distributed System and Scheduler's states. First, we detail the distributed system linking, then we detail the scheduler state linking. The reason behind this is that without first having a linked distributed system copy ds'', we can't link a scheduler's state. This, in turn, is due to the fact that there must be a distributed system in existence so that a scheduler state makes sense e.g. blocking tasks manager may contain blocking tasks who w/o distributed system won't make any sense. Other examples are scheduler's consume queue, task

Kind	Generated `code` body	Fields in Statement wrapper
SEND	ds.send(sa,mo,da)	`ds` distributed system (applies to all kinds) `sa` as source agent `mo` as the message sent `da` as destination agent
ASK	f = ds.ask(sa, mo, da)	`f` the future to be returned rest is the same as SEND
CREATE	ds.create(sa, da)	`sa` source agent, parent/creator `da` destination agent, created one
START	ds.start(sa, da)	`sa` starter agent `da` started agent
STOP	ds.stop(sa, da)	`sa` stopper agent `da` stopped agent
KILL	ds.kill(sa, da)	`sa` killer agent `da` killed agent
LOCK	ds.lock(sa)	`sa` locked agent
UNLOCK	ds.unlock(sa)	`sa` unlocked agent
STOP_CONSUME	ds.stopConsuming(sa)	`sa` agent stopped consuming
RESUME_CONSUME	ds.resumeConsuming(sa)	`sa` agent resumed consuming
BECOME	ds.become(sa, bn, r)	`sa` agent becoming another behavior `bn` behavior name to become `r` remember old behavior or not
UNBECOME	ds.unbecome(sa)	`sa` unbecoming agent
STASH	ds.stash(sa, mo)	`sa` agent stashing the message `mo` stashed message
UNSTASH	ds.unstash(sa)	`sa` agent unstashing a message
UNSTASH_ALL	ds.unstashAll(sa)	`sa` agent stashing all messages
GET	ds.get(sa, f, act)	`sa` agent potentially blocking on future `f` future to block on `act` if blocked, the actual `action` blocked
GET_TIMED	ds.get(sa, f, to, act)	`to` time out for blocking get rest is the same as GET
BOOT_STRAP	ds.bootStrap(da)	`da` boot strapped agent

BOOT_STRAP_ALL	val agents = bs map {
	x ⇒ ds.get(x) }.toSet;
	ds.bootStrap(agents)
		`bs` names of boot strapped agents
		`agents` agents to boot strap

MODIFY_STATE	da.$\mathcal{L}$(var) = value	`da` agent whose local state to modify `var` variable name and type to modify `val` the value stored in that var
MODIFY_STATE_REF		
IF_ELSE		
WHILE		
FUNCTION		
NONE	nothing, referenced already	not a field changes
everything else	thorw an error	not a field changes

Table 2.2: Statement code generation and what fields are used in each case

queue, and timed actions tracker all of which content won't make sense if there wasn't a distributed system to link against.

2.12.1 IR Data-structures State

[Distributed System]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ ds' \in \mathcal{D} \\ st = link(ds') \\ st \Downarrow \forall \alpha' \in ds'[\mathcal{A}'], \mu' \in ds'[\mathcal{M}'], \gamma' \in\in ds'[\Gamma'], \delta' \in ds'[\Delta'] : \\ link(ds', \alpha') \Downarrow ds'[\mathcal{A}' \mapsto \mathcal{A}' \cup \alpha'']\wedge \\ link(ds', \mu') \Downarrow ds'[\mathcal{M}' \mapsto \mathcal{M}' \cup \mu'']\wedge \\ link(ds', \gamma') \Downarrow ds'[\Gamma' \mapsto \Gamma' \cup \gamma'']\wedge \\ link(ds', \delta') \Downarrow ds'[\Delta' \mapsto \Delta' \cup \delta''] \end{array}}{\begin{array}{c} ds' \rightarrow ds''[\mathcal{A}' \mapsto \mathcal{A}'', \mathcal{M}' \mapsto \mathcal{M}'', \Gamma' \mapsto \Gamma'', \Delta' \mapsto \Delta'']\wedge \\ ds'' = ds \end{array}}$$

Description: In order to link a distributed system copy ds' and produce a linked copy ds'' that is identical to but separate from ds, all its agents, messages, actions, and behaviors need to be linked.

[Agent]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ \alpha' \in ds'[\mathcal{A}'] \\ st = link(ds', \alpha') \\ st \Downarrow \forall \mu' \in \alpha'[q'] : link(ds', \mu') \Downarrow \alpha'[q'[\mu' \mapsto \mu'']]\wedge \\ \forall \mu' \in \alpha'[\mathcal{S}'] : link(ds', \mu') \Downarrow \alpha'[\mathcal{S}'[\mu' \mapsto \mu'']]\wedge \\ link(ds', \alpha'[\theta']) \Downarrow \alpha'[\theta' \mapsto \theta'']\wedge \\ link(ds', \alpha'[\mathcal{R}']) \Downarrow \alpha'[\mathcal{R}' \mapsto \mathcal{R}'']\wedge \\ link(ds', \alpha'[\mathcal{B}']) \Downarrow \alpha'[\mathcal{B}' \mapsto \mathcal{B}'']\wedge \\ link(ds', \alpha'[\sigma']) \Downarrow \alpha'[\sigma' \mapsto \sigma'']\wedge \\ link(ds', \alpha'[bo']) \Downarrow \alpha'[bo' \mapsto bo'']\wedge \\ \forall ta' \in \tau' : link(ds', ta') \Downarrow \alpha'[\tau'[ta' \mapsto ta'']]\wedge \\ \forall f' \in fp' : link(ds', f') \Downarrow \alpha'[fp'[f' \mapsto f'']]\wedge \\ \forall f' \in fw' : link(ds', f') \Downarrow \alpha'[fw'[f' \mapsto f'']] \end{array}}{ds' \rightarrow ds'[\mathcal{A}'[\alpha' \mapsto \alpha'']] \wedge \alpha'' = ds[\mathcal{A}][\alpha]}$$

Description: Linking an agent requires linking all of its fields that:

1. have at least ONE field of Agent type $\mathcal{A}$
2. and/or is composed of fields with at least one that distributed system specific `ir` e.g. message, agent, ...etc.

All other fields should have been taken care of by the `copy(...)` operation. Those fields for example are just terminal values such as a string, a number, ...etc. Actually, all of the copy-link operation was due to the fact that there are cyclic references to agents between `ir` instances in the distributed system.

[Behavior]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ b' \in rng(ds'[\Delta']) \\ st = link(ds', b') \\ st \Downarrow (getSet(ds', b'[an], b', a) \Downarrow b'[a' \mapsto a''])\wedge \\ \forall r' = (\mu', act') \in b'[\mathcal{R}] : link(ds', r') \Downarrow r' \mapsto r'' = (\mu'', act'') \wedge b'[\mathcal{R}'[r' \mapsto r'']] \end{array}}{ds' \to rng(ds'[\Delta'])[b' \mapsto b''] \wedge b'' = ds[\Delta][b]}$$

Description: Linking a behavior requires its agent field to be set to the agent α' from the new copy of the distributed system ds'. Also, all reaction pairs $r' = (\mu', act')$ need to be linked to the new distributed system, pairs to become $r'' = (\mu'', act'')$. Only then the behavior is considered linked.

[Action]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ \gamma' \in ds'[\Gamma'] \\ st = link(ds', \gamma') \\ st \Downarrow (getSet(ds', \gamma'[an], \gamma', a) \Downarrow \gamma'[a \mapsto a'])\wedge \\ \forall s' \in \gamma'[stmts'] : link(ds', s') \Downarrow \gamma'[stmts'[s' \mapsto s'']] \end{array}}{ds' \to ds'[\Gamma'[\gamma' \mapsto \gamma''[stmts' \mapsto stmts'']]] \wedge \gamma'' = ds[\Gamma][\gamma]}$$

Description: Note that the agent field of the action $\gamma'[a]$ can still be a copy a'. This is correct behavior of the link operation because weather that agent was linked or not before we link the action γ', it is stil going to be linked anyways and the action will be referencing it anyways, so the action will eventually reference a linked copy a'' of a.

[Timed-Action]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ \alpha' \in ds'[\mathcal{A}'] \\ ta' \in \alpha'[\tau'] \\ ta' = (m', a, act', t, c) \\ st = link(ds', ta') \\ st \Downarrow (getSet(ds', ta'[an], ta', a) \Downarrow ta'[a \mapsto a'])\wedge \\ link(ds', act') \Downarrow ta'[act' \mapsto act''] \end{array}}{ds' \to ds'[\mathcal{A}'[\alpha'[\tau'[ta' \mapsto ta'']]]] \wedge ta'' = ds[\mathcal{A}][\alpha][\tau][ta]}$$

Description: Linking a timed action requires setting its agent field a to the new system's agent object of the same name specificed in $ta'[an]$, and calling the `link(...)` function on its act' action.

[Statement]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ \gamma' \in ds'[\Gamma'] \\ s' \in \gamma'[stmts'] \\ st = link(ds', s') \\ st \Downarrow (link(ds', s'[m']) \Downarrow s'[m' \mapsto m''])\wedge \\ (getSet(ds', s'[an], s', s'[a]) \Downarrow s'[a \mapsto a'])\wedge \\ (link(ds', s'[mo']) \Downarrow s'[mo' \mapsto mo''])\wedge \\ (getSet(ds', s'[san], s', s'[sa]) \Downarrow s'[sa \mapsto sa'])\wedge \\ (getSet(ds', s'[dan], s', s'[da]) \Downarrow s'[da \mapsto da'])\wedge \\ (link(ds', s'[f']) \Downarrow s'[f' \mapsto f''])\wedge \\ (link(ds', s'[act']) \Downarrow s'[act' \mapsto act''])\wedge \\ s'[ds \mapsto ds'] \end{array}}{ds' \rightarrow ds'[\Gamma'[\gamma'[stmts'[s' \mapsto s'']]]] \wedge s'' = ds[\Gamma][\gamma][stmts][s]}$$

Description: Linking a statement invokes a chain of other rules to link its sub-fields. For example, its agent fields `a`, `sa`, and `da` get set to the agents with the same name from the new distributed system copy ds'. The ds fields of the statement gets a reference to the new copy ds', and all other constructs are linked using their respective linking rule. After all that is done, the statement copy becomes identical to but separate from the original statement copied from the original distributed system ds.

[Linking Code of Statement]

By linking the statement itself, its code is automatically linked. Given that the code was generated using an instrumentation constructor like described in the code generation section §2.11. Otherwise, if the statement kind is `NONE`, its `code` just gets referenced.

[Message]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ \mu' \in ds'[\mathcal{M}'] \\ st = link(ds', \mu') \\ st \Downarrow (getSet(ds', \mu'[sn], s) \Downarrow \mu'[s \mapsto s']) \end{array}}{ds' \rightarrow ds'[\mathcal{M}'[\mu' \mapsto \mu'']] \wedge \mu'' = ds[\mathcal{M}][\mu]}$$

Description: To link a message, we only need to assign its `s` field an agent from the new distributed system copy's agents $ds'[\mathcal{A}']$ that has the same name specified by the message field $\mu'[sn]$.

Note: As you may have noticed, we are referring to messages in a distributed system's messages collection $ds'[\mathcal{M}']$. This is to keep the rule simple and readable. However, keep in mind that a message may exist in an agent's incoming queue, agent's stash, behaviors, . . . etc.

[Future]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ f' \in \mathcal{F} \\ st = link(ds', f') \\ st \Downarrow (getSet(ds', f'[pbn], f', pb) \Downarrow f'[pb \mapsto pb'])\wedge \\ (getSet(ds', f'[wfn], f', wf) \Downarrow f'[wf \mapsto wf']) \end{array}}{\mathcal{F} \rightarrow \mathcal{F}[f' \mapsto f''] \wedge f'' = \mathcal{F}[f]}$$

Description: To link a future, we need to set its two fields $f'[pb]$ and $f'[wf]$ to agents from the new distributed system copy's agents $ds'[\mathcal{A}']$ whose names are $f'[pbn]$ and $f'[wfn]$, respectively. Future's can exist anywhere in an agent, a suspendable task, and in blocking manager but again we simplify the rule making it more readable by refering to the universal set of futures $\mathcal{F}$.

[Task]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ t' \in \{(a, an, act')\} \\ ss' \in SchedulerState \\ st = link(ds', t') \\ st \Downarrow (getSet(ds', t'[an], t', a) \Downarrow t'[a \mapsto a'])\wedge \\ (link(ds', t'[act]) \Downarrow t'[act' \mapsto act'']) \end{array}}{\begin{array}{c} (ss' \mapsto ss'[tQ \mapsto tQ[t' \mapsto t'']] \vee ss'[cQ \mapsto cQ[t' \mapsto t'']] \vee ss'[btm' \mapsto btm'[bt[t' \mapsto t'']]])\wedge \\ (t'' = ss[tQ][t] \vee t'' = ss[cQ][t] \vee t'' = ss[btm][bt][t]) \end{array}}$$

Description: To link a task to a new distributed system ds', we need to set its $t'[a]$ to an agent from the new distributed system whose name is $t'[an]$, and then the action of that task $t'[act']$ needs to be linked by invoking its `link(...)` method/rule. The `OR` in the transitions is to give the choce of where that task lies and, hence, should lie after being linked.

2.12.2 Scheduler State

Like copying a scheculer's state, here we detail linking the components linking then we state the wholse scheduler state linking.

[Blocking-Tasks-Manager]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ ss' \in SchedulerState \\ btm' = ss'[btm'] \\ st = link(ds', btm') \\ st \Downarrow (\forall t' \in btm'[bt] : link(ds', t') \Downarrow btm'[bt[t' \mapsto t'']]) \wedge \\ \forall t' \in btm'[bt] : (\exists f \in t'[pb[fp]] : f[id] = t'[f'[id]] \Rightarrow t'[pb[fp[f \mapsto f']]]) \wedge \\ (\exists f \in t'[wf[fw]] : f[id] = t'[f'[id]] \Rightarrow t'[wf[fw[f \mapsto f']]]) \wedge \\ (t'[wf[bo' \mapsto t'[f]]]) \wedge \\ lastExecuted(t'[act]) \mapsto t'[act] \end{array}}{ss' \rightarrow ss'[btm' \mapsto btm''] \wedge btm'' = ss[btm]}$$

Description: Linking a blocking tasks manager is the most involved operation in the two phase copy-link operation. First, all tasks that are blocked, stored in $btm'[bt]$ are individually linked like any normal task. Second, for each task t' of them, each of the following must be done:

- Agent referenced in the future field of the task $f[pb]$, has futures promised collection fp. We update the future whose `id` is the same as the future on which the current task is blocking to reference the *blocking task's future*, i.e. $t'[pb[fp[f \mapsto f']]]$ where $f' = t'[f']$.
- We do the same for the waiting/blocked agent, however on its fw collection of futures. Also, the blocking agent `bo` field is made to reference the future of the current task.
- Then, we need to update the statement that blocked the current task, can be extracted using $lastExecuted(t'[act])$, so that its act field references the blocked task's act field. This one is for instrumentation purposes.

Only after doing all of the above, a blocking tasks managet is actually linked so that it is identical to but separate from the original blocking tasks manager $ss[btm]$.

[Timed-Actions-Tracker]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ ss' \in SchedulerState \\ tat' = ss'[tat'] \\ st = link(ds', tat') \\ st \Downarrow \forall t' \in tat'[ta'] : (link(ds', t') \Downarrow tat'[ta'[t' \mapsto t'']]) \end{array}}{ss' \rightarrow ss'[tat' \mapsto tat''] \wedge tat'' = ss[tat]}$$

Description: Linking a timed action tracker means linking every timed action it tracks that is stored in $ss'[tat']$. After that happens, then the linked tat'' is an identical to but separate copy from the original timed actions tracker $ss[tat]$.

[Task Queue]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ ss' \in SchedulerState \\ q' = ss'[tQ'] \\ st = link(ds', q') \\ st \Downarrow \forall t' \in q' : (link(ds', t') \Downarrow q'[\mu' \mapsto \mu'']) \end{array}}{ss' \to ss'[tQ' \mapsto q''] \wedge q'' = ss[tQ]}$$

Description: Linking a task queue constitutes linking all of the tasks inside of it. Only then it will be an equal copy q'' but separate from the original task queue $ss[tQ]$.

[Consume Queue]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ ss' \in SchedulerState \\ q' = ss'[cQ'] \\ st = link(ds', q') \\ st \Downarrow \forall t' \in q' : (link(ds', t') \Downarrow q'[\mu' \mapsto \mu'']) \end{array}}{ss' \to ss'[cQ' \mapsto q''] \wedge q'' = ss[cQ]}$$

Description: Linking a consume queue constitutes linking all of the tasks inside of it. Only then it will be an equal copy q'' but separate from the original task queue $ss[cQ]$.

[Scheduler State]

$$\frac{\begin{array}{c} st \in \mathcal{P} \\ ss' \in SchedulerState \\ st = link(ds', ss') \\ (link(ds', ss'[tQ']) \Downarrow ss'[tQ' \mapsto tQ''])\wedge \\ (link(ds', ss'[cQ']) \Downarrow ss'[cQ' \mapsto cQ''])\wedge \\ (link(ds', ss'[bMgr']) \Downarrow ss'[bMgr' \mapsto bMgr''])\wedge \\ (link(ds', ss'[tat']) \Downarrow ss'[tat' \mapsto tat'']) \end{array}}{ss' \mapsto ss'' \wedge ss'' = ss}$$

Description: Linking a scheduler state copy ss' requires linking all its fields each using their respective linking rule. Only then the scheduler state is an identical to but separate from the original scheduler state. Note that the scheduler state contains the minimal set of fields which constitutes a *basic* scheduler state, ommitting the history of exploration which is *trace manager* as restoring and resuming the state of a scheduler should NEVER reset the trace manager, otherwise it may lose exploration history. Never the less, this is easily overrideable by including one field `tMgr` in the scheduler state that doesn't need `custome-copying` nor `linking`, since no entities are to be used to snapshot and/or resume execution of the runtime. It can be copied using the normal serialization mechanism already implemented in it, namely: `toJson` and `fromJson` methods composed in this specific way: `fromJson(toJson(tMgr))`.

2.13 Snapshotting and Restoring runtime state

Snapshotting the whole runtime state and then restoring to that exact state from a schedulr becomes a breez from now on. In order to snapshot the system state into a scheduler state object call it `state`, is done as the following:

$$state = saveState(sch)$$

where sch is the scheduler. Restoring from that state later on is done by the following:

$$restoreState(sch, state)$$

In near future, sch need not be the same scheduler that saved the state. This is to enable different schedulers, for example, running in parallel to explore different paths in the distributed system's state.

Bibliography

[1] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual consistency," *Commun. ACM*, vol. 57, no. 5, pp. 61–68, 2014.

[2] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, pp. 40–44, Jan. 2009.

[3] S. Burckhardt, "Principles of eventual consistency," *Foundations and Trends in Programming Languages*, vol. 1, no. 1-2, pp. 1–150, 2014.

[4] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv, *Programming Languages and Systems: 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, ch. Eventually Consistent Transactions, pp. 67–86. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[5] "Typesafe." http://www.typesafe.com/activator/template/hello-akka,Retrieved Jan 27, 2016.

[6] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey, "P: Safe asynchronous event-driven programming," Tech. Rep. MSR-TR-2012-116, November 2012.

[7] S. Rathbun, "Parallel processing with promises," *Queue*, vol. 13, pp. 10:10–10:18, Mar. 2015.

[8] "Futures and promises." https://en.wikipedia.org/wiki/Futures_and_promises, Retrieved Jan 31, 2016.

[9] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, pp. 17–32, feb 2003.

[10] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos Made Live: An Engineering Perspective," in *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, (New York, NY, USA), pp. 398–407, ACM, 2007.

[11] L. Lamport, "Paxos Made Simple," *SIGACT News*, vol. 32, pp. 51–58, Dec. 2001.

[12] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, pp. 133–169, May 1998.

[13] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pp. 245–256, IEEE, 2011.

[14] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “ZooKeeper: Wait-free Coordination for Internet-scale Systems,” in *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIX ATC ’10, (Berkeley, CA, USA), pp. 11–11, USENIX Association, 2010.

[15] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, (Philadelphia, PA), pp. 305–319, USENIX Association, June 2014.