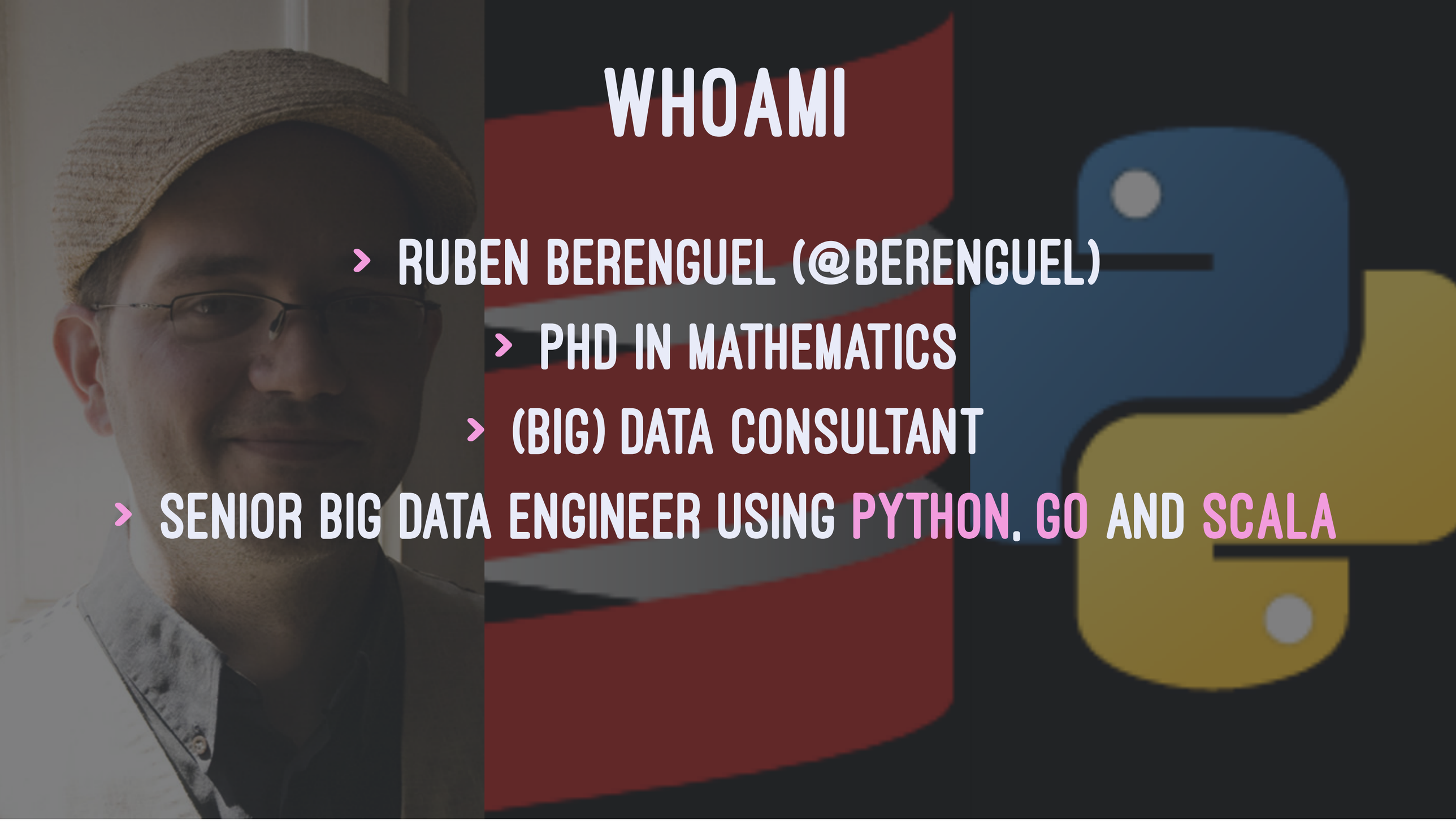
FORMALLY VERIFYING COMPLEX SYSTEMS USING TLA^+

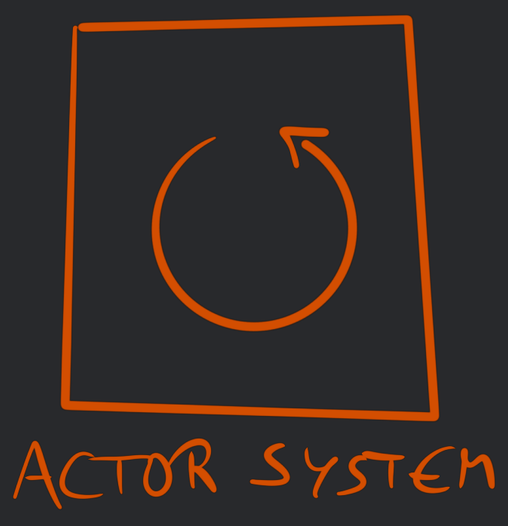


WHOAMI

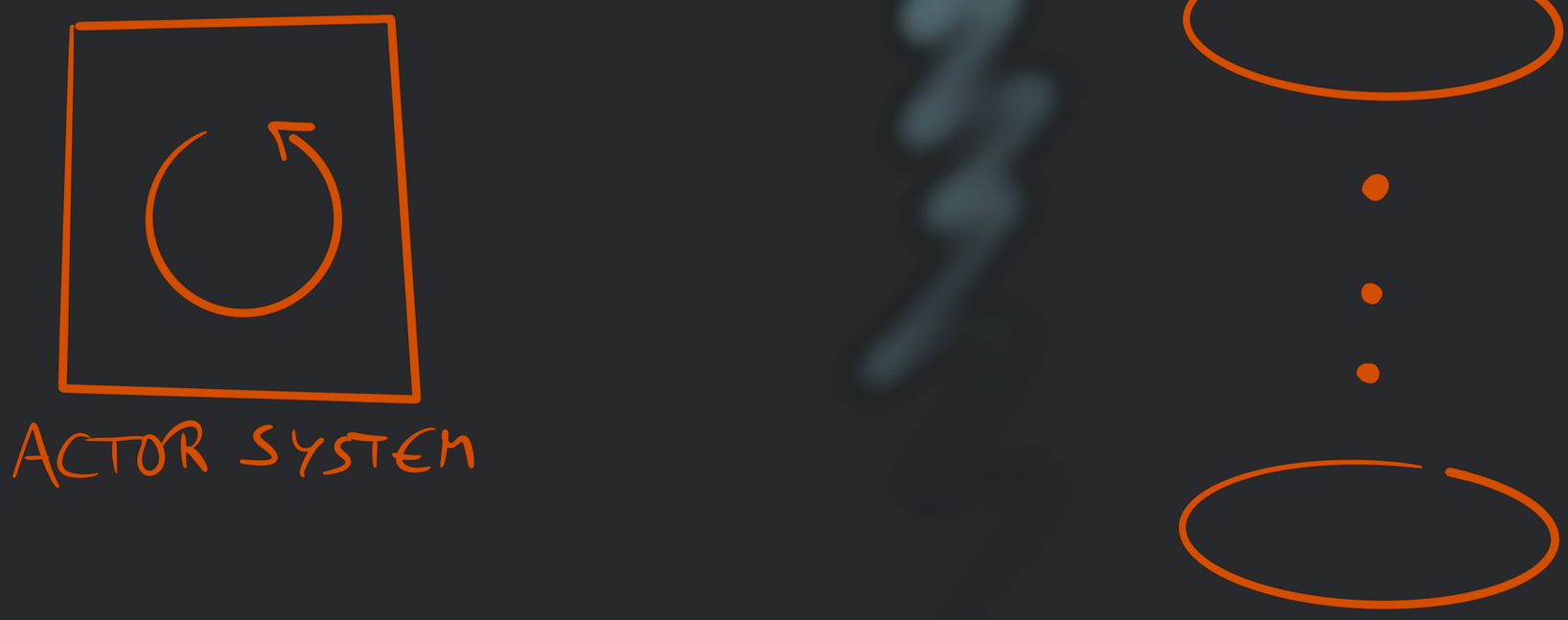
- RUBEN BERENGUEL (@BERENGUEL)
- PHD IN MATHEMATICS
- (BIG) DATA CONSULTANT
- SENIOR BIG DATA ENGINEER USING PYTHON, GO AND SCALA

FORMAL METHODS

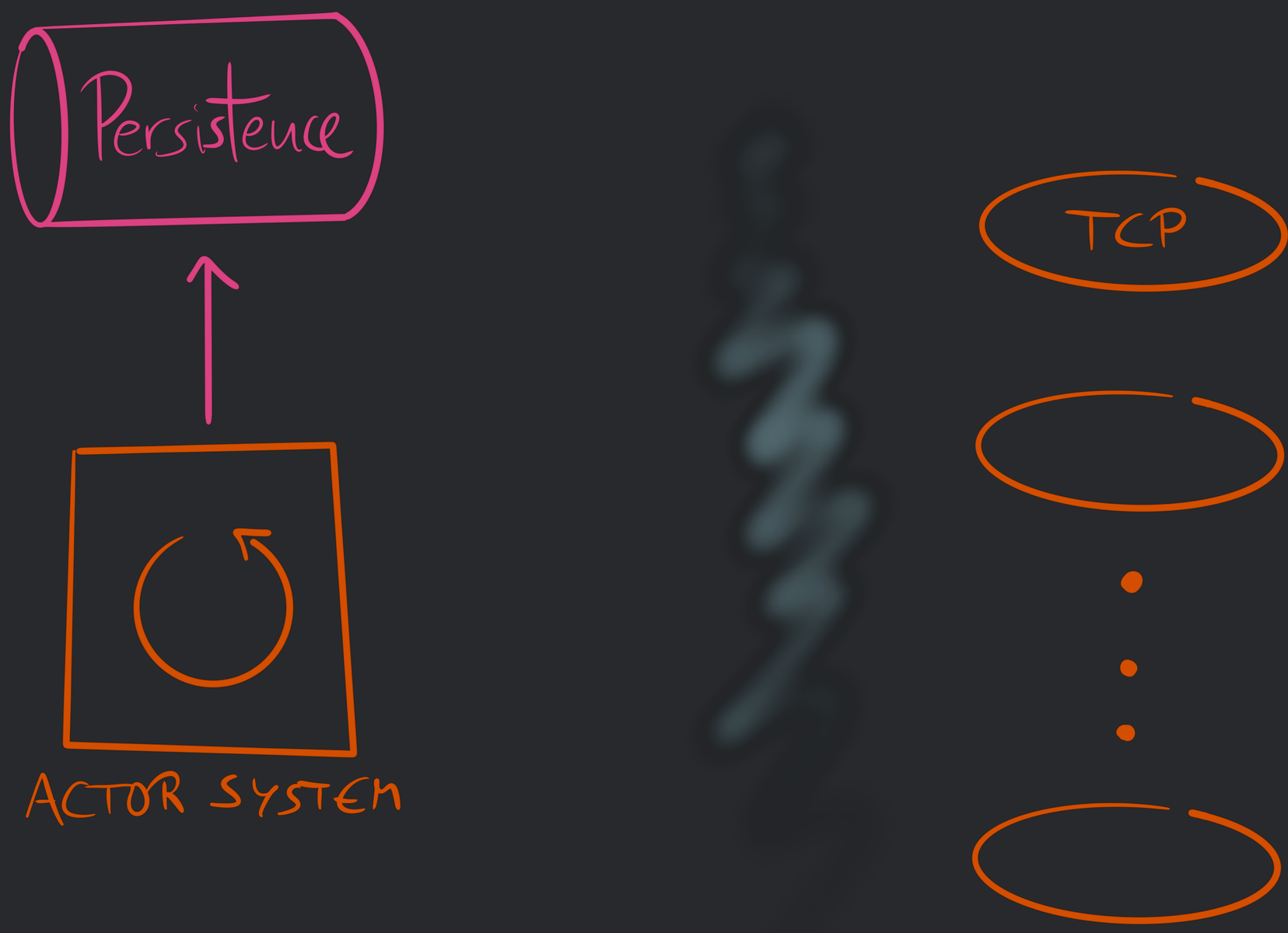
WHY?



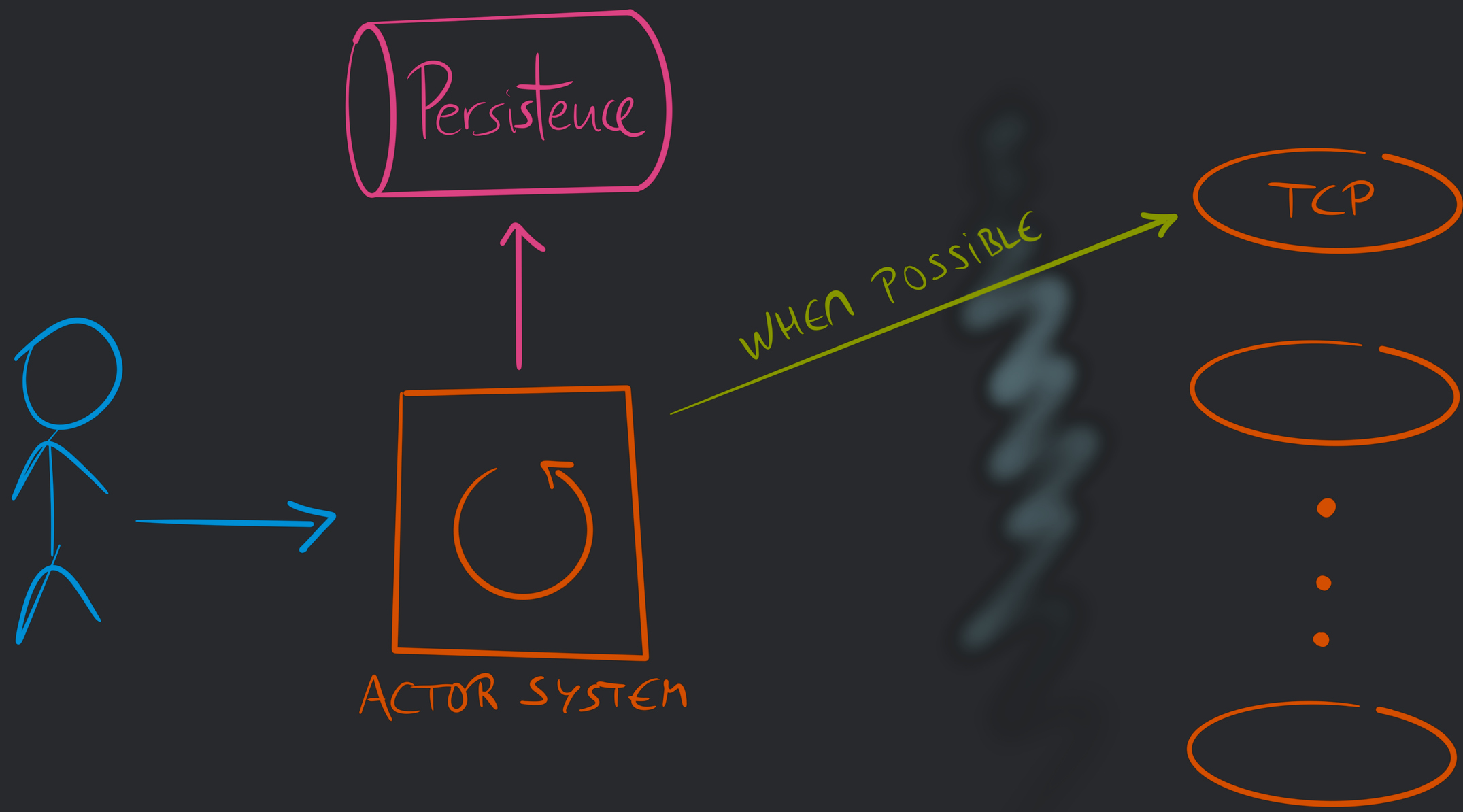
I had a system based in Akka



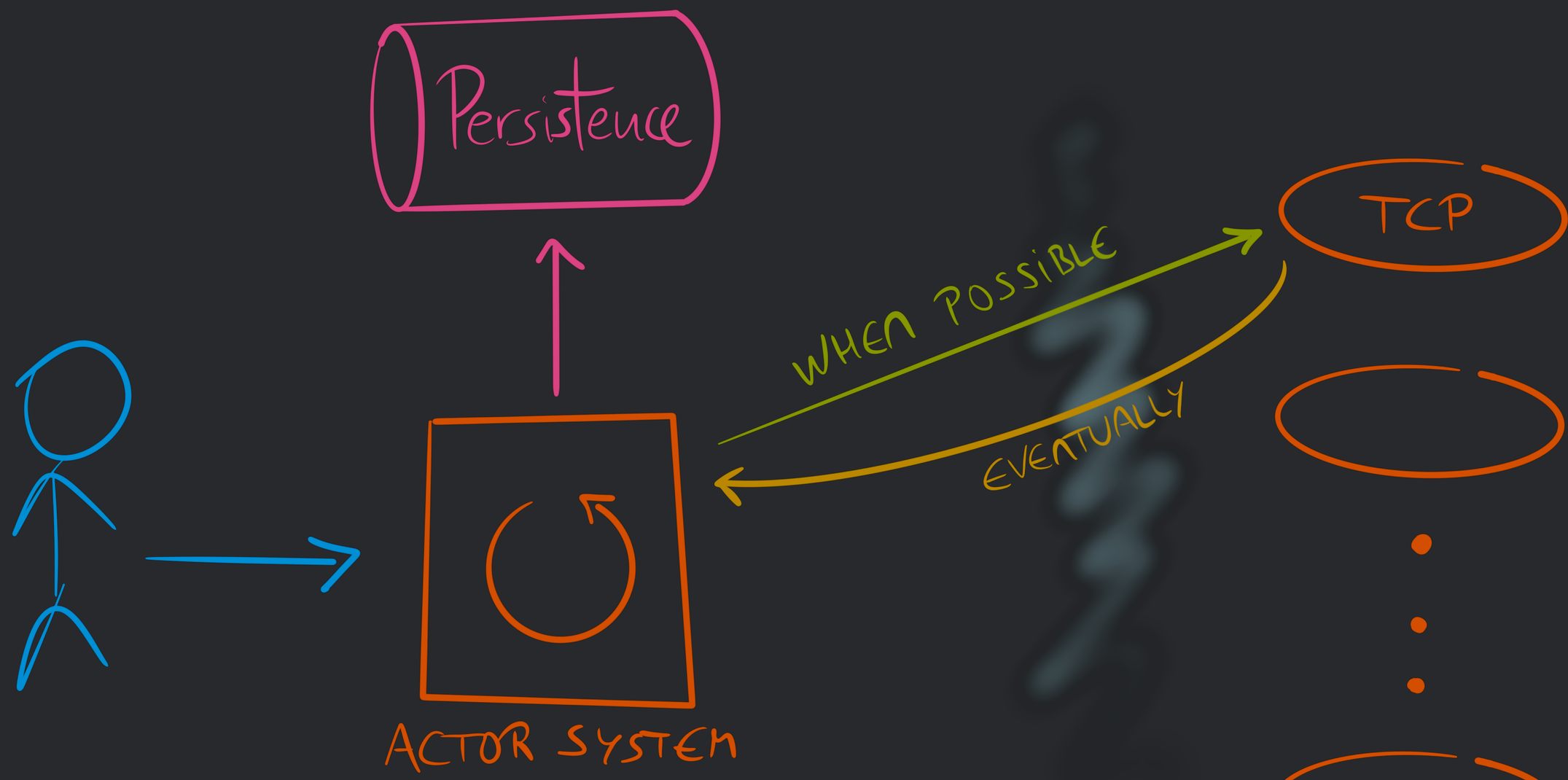
Requiring to communicate with many external systems (via TCP)



The system was using Akka persistence to store its internal state mutations



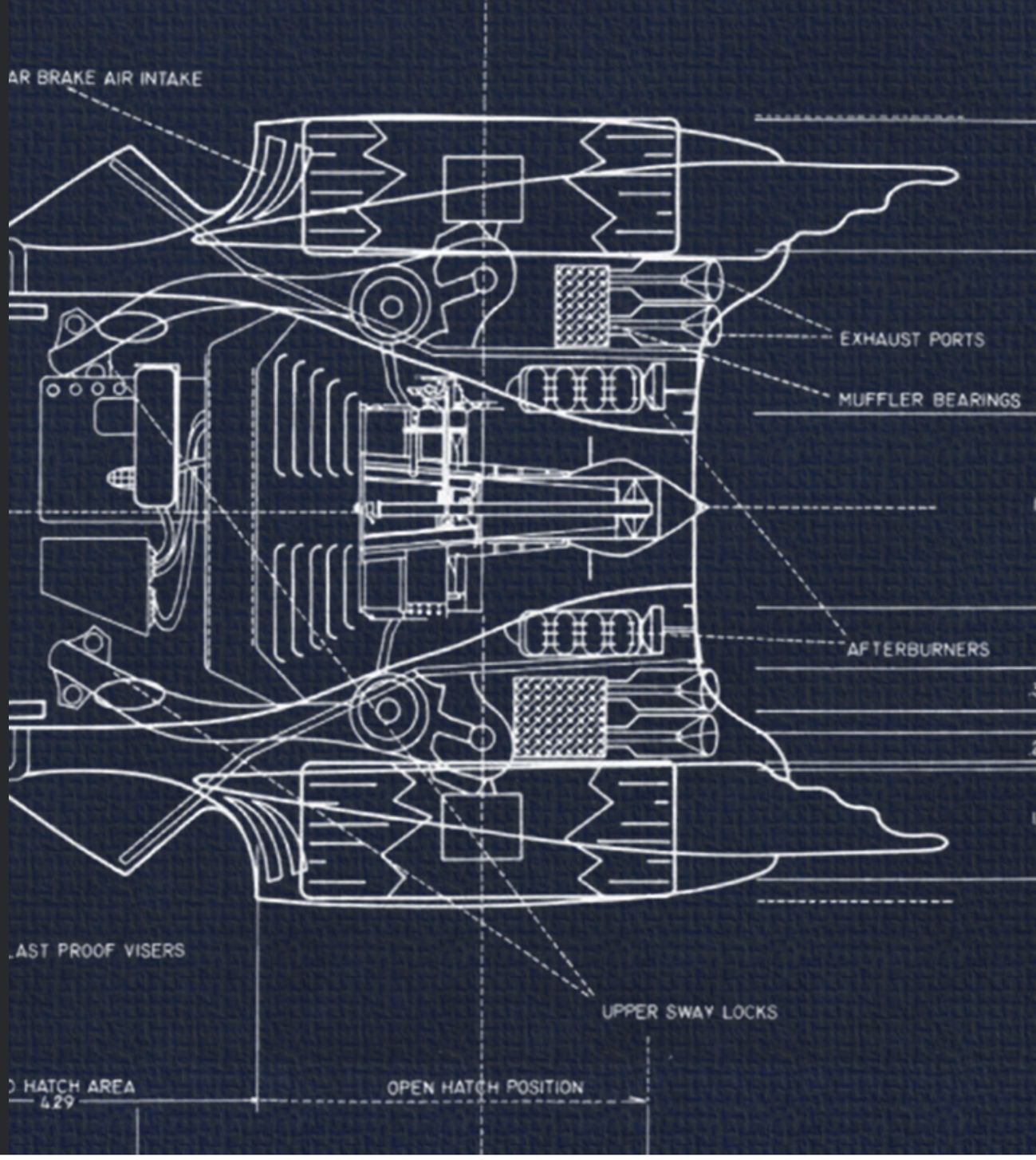
An external "source" (say, a user) would trigger a change request that would need to be forwarded to the external systems when they were available. The state of the request goes into the internal state of the actor.



And they should answer fulfilment status of the request when possible.

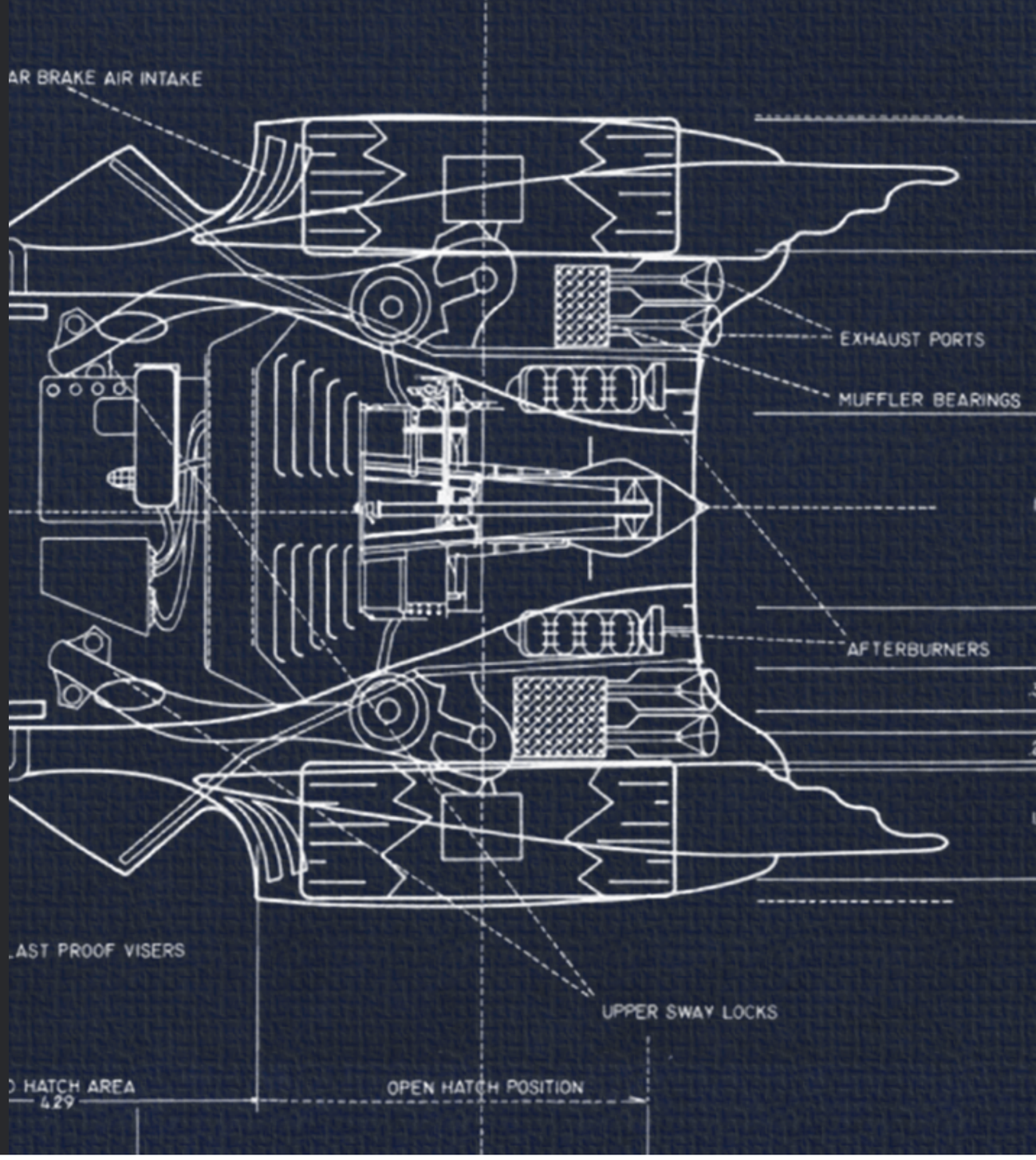
How can I be sure the external services get the request, and the actor system get the confirmation, given the large amount of possible states that persistence introduces?

WHAT ARE FORMAL METHODS**?**



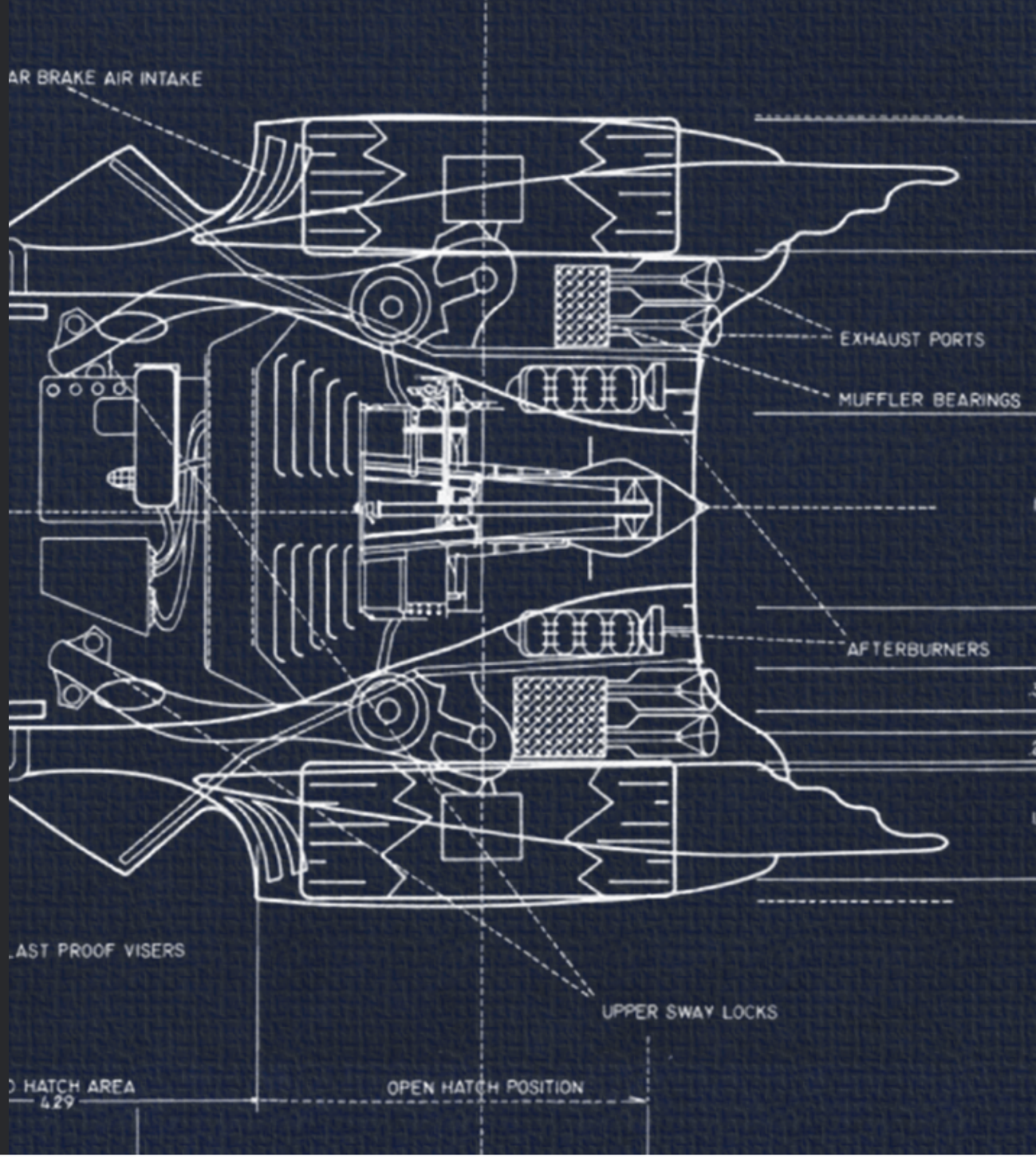
WHAT ARE FORMAL METHODS?

> TECHNIQUE FOR SPECIFYING THE BEHAVIOUR OF HARDWARE OR SOFTWARE



WHAT ARE FORMAL METHODS?

- TECHNIQUE FOR SPECIFYING THE BEHAVIOUR OF HARDWARE OR SOFTWARE
- BASED ON SOME MATHEMATICAL FORMALISM OR THEORY



EXAMPLES

EXAMPLES

- LAMBDA CALCULUS

EXAMPLES

- LAMBDA CALCULUS
- (POSSIBLY DEPENDENT) TYPE SYSTEMS

EXAMPLES

- LAMBDA CALCULUS
- (POSSIBLY DEPENDENT) TYPE SYSTEMS
- TLA (TEMPORAL LOGIC OF ACTIONS)

WHAT DOES FORMAL VERIFICATION REQUIRE?

A specification is an abstraction: it describes some aspects of the system and
ignores others. We want the specification to be as simple as possible (but, like
someone famous said, *not simpler*)

WHAT DOES FORMAL VERIFICATION REQUIRE?

- WRITE A SPECIFICATION OF WHAT THE SYSTEM IS SUPPOSED TO DO

A specification is an abstraction: it describes some aspects of the system and
ignores others. We want the specification to be as simple as possible (but, like
someone famous said, *not simpler*)

WHAT DOES FORMAL VERIFICATION REQUIRE?

- WRITE A SPECIFICATION OF WHAT THE SYSTEM IS SUPPOSED TO DO
- COMPUTATIONALLY VERIFY THE DESIRED PROPERTIES

A specification is an abstraction: it describes some aspects of the system and
ignores others. We want the specification to be as simple as possible (but, like
someone famous said, *not simpler*)

WHAT DOES FORMAL VERIFICATION REQUIRE?

- WRITE A SPECIFICATION OF WHAT THE SYSTEM IS SUPPOSED TO DO
  - COMPUTATIONALLY VERIFY THE DESIRED PROPERTIES
  - OPTIONALLY/ADDITIONALLY: PROVE THE DESIRED PROPERTIES OF THE SYSTEM

A specification is an abstraction: it describes some aspects of the system and ignores others. We want the specification to be as simple as possible (but, like someone famous said, *not simpler*)

WHAT IS TLA^+?

FORMAL SPECIFICATION LANGUAGE DEVELOPED ON TOP OF TLA BY LESLIE LAMPORT[1]

MODELS WRITTEN IN TLA^+ CAN BE CHECKED USING TLC

[1] THE LA FROM LaTeX AND THE, WELL, AUTHOR OF THE PAXOS) CONSENSUS ALGORITHM, AMONG MANY OTHER THINGS

Temporal Logic was introduced in the late 1950s, and LLs Temporal Logic of Actions was published in 1994. The language was introduced in 1999, late that year TLC was started

A PSEUDOCODE-LIKE LANGUAGE. PLUSCAL TRANSPILES INTO TLA^+. AND CAN BE USED FOR SEQUENTIAL ALGORITHMS

COMES IN TWO FLAVOURS, P[2] AND C[3]

[2] P OF PASCAL? `begin while do stuff end while;`

[3] C AS IN C. {CURLY CURLY}

PlusCal (formerly +Cal) has been available since 2009

AN ADDITIONAL PROOF SYSTEM TLAPS[4] CAN BE USED TO CHECK PROOFS

THIS IS HOW BIZANTYNE PAXOS[5] WAS PROVEN, AND HAS BEEN USED IN SEVERAL INDUSTRY PROJECTS

[4] CAN BE FOUND HERE, IT'S SEPARATE FROM THE TLA+ TOOLBOX

[5] MECHANICALLY CHECKED SAFETY PROOF OF A BYZANTINE PAXOS ALGORITHM

Has been used for proving soundness in Azure's CosmosDB, for instance

SYSTEM	COMPONENTS	LINES OF CODE	RESULT
S3	FAULT TOLERANT NETWORK ALGO	804 (PLUSCAL)	2 BUGS, FURTHER BUGS IN LATER OPTIMISATIONS
S3	**BACKGROUND REDISTRIBUTION**	**645 (PLUSCAL)**	**1 BUG, 1 BUG IN THE FIX**
DYNAMODB	REPLICATION & MEMBERSHIP	939 (PLUSCAL)	3 BUGS (35 STEP TRACE)
EBS	**VOLUME MANAGEMENT**	**223 (PLUSCAL)**	**BETTER CONFIDENCE**
INTERNAL LOCK MANAGER	REPLICATION	318 TLA+	1 BUG

Here we can see some examples of use from Amazon Web Services (from here)

HOW DOES TLA^+ LOOK LIKE?

I will show a simple infinite state machine in TLA+

```
-- MODULE ThemeSong --

EXTENDS Integers

VARIABLES s, count

vars == <<s, count>>

Sinit == s = "na" /\ count = 2

Na == s' = "na" /\ count' = count - 1
ToB == s' = "Batman" /\ count' = 0

Snext == IF count > 0 THEN /\ ( Na \/ ToB)
                      ELSE /\ ToB

Spec == Sinit /\ [][Snext]_vars /\ WF_vars(ToB)

Batman == s = "Batman" /\ count = 0

Liveness == <>Batman
```

```
-- MODULE ThemeSong --

EXTENDS Integers

VARIABLES s, count

vars == <<s, count>>

Sinit == s = "na" /\ count = 2

Na == s' = "na" /\ count' = count - 1
ToB == s' = "Batman" /\ count' = 0

Snext == IF count > 0 THEN /\ ( Na \/ ToB)
                      ELSE /\ ToB

Spec == Sinit /\ [][Snext]_vars /\ WF_vars(ToB)

Batman == s = "Batman" /\ count = 0

Liveness == <>Batman
```

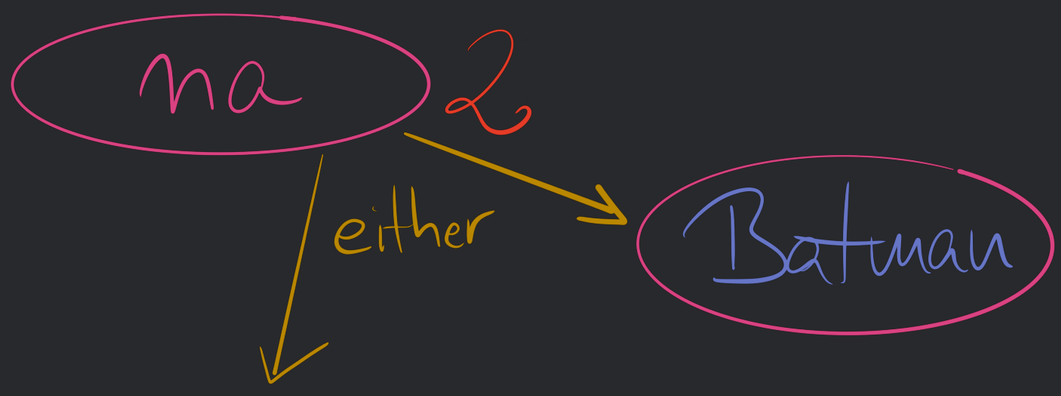
```
-- MODULE ThemeSong --

EXTENDS Integers

VARIABLES s, count

vars == <<s, count>>

Sinit == s = "na" /\ count = 2

Na == s' = "na" /\ count' = count - 1
ToB == s' = "Batman" /\ count' = 0

Snext == IF count > 0 THEN /\ ( Na \/ ToB)
                      ELSE /\ ToB

Spec == Sinit /\ [][Snext]_vars /\ WF_vars(ToB)

Batman == s = "Batman" /\ count = 0

Liveness == <>Batman
```

This is the specification of the system

```
-- MODULE ThemeSong --

EXTENDS Integers

VARIABLES s, count

vars == <<s, count>>

Sinit == s = "na" /\ count = 2

Na == s' = "na" /\ count' = count - 1
ToB == s' = "Batman" /\ count' = 0

Snext == IF count > 0 THEN /\ ( Na \/ ToB)
                      ELSE /\ ToB

Spec == Sinit /\ [][Snext]_vars /\ WF_vars(ToB)

Batman == s = "Batman" /\ count = 0

Liveness == <>Batman
```

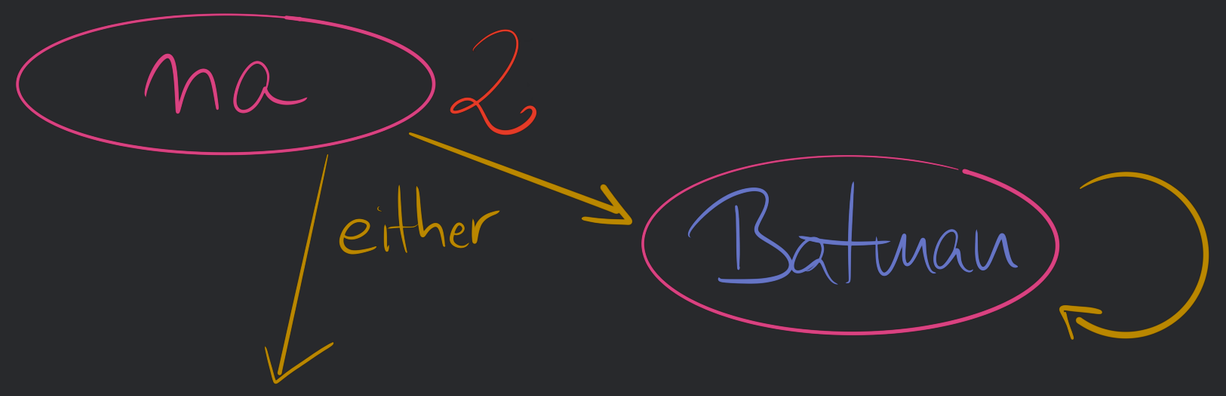
And this is the initial state

```
-- MODULE ThemeSong --

EXTENDS Integers

VARIABLES s, count

vars == <<s, count>>

Sinit == s = "na" /\ count = 2

Na == s' = "na" /\ count' = count - 1
ToB == s' = "Batman" /\ count' = 0

Snext == IF count > 0 THEN /\ ( Na \/ ToB)
                      ELSE /\ ToB

Spec == Sinit /\ [][Snext]_vars /\ WF_vars(ToB)

Batman == s = "Batman" /\ count = 0

Liveness == <>Batman
```



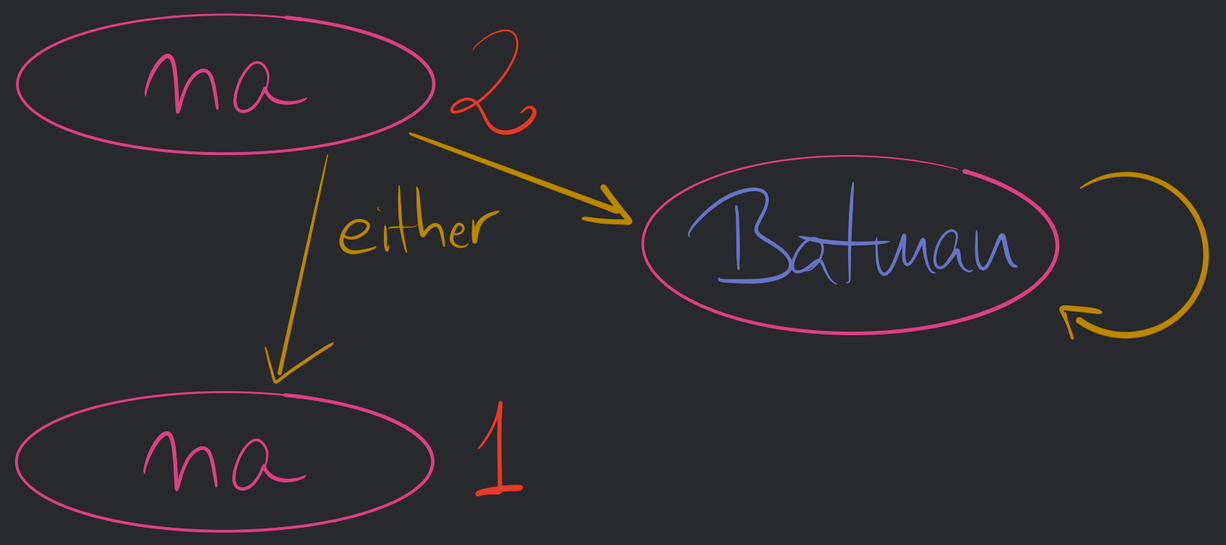
The next state after is an "either" state when `count > 0`

```
-- MODULE ThemeSong --

EXTENDS Integers

VARIABLES s, count

vars == <<s, count>>

Sinit == s = "na" /\ count = 2

Na == s' = "na" /\ count' = count - 1
ToB == s' = "Batman" /\ count' = 0

Snext == IF count > 0 THEN /\ ( Na \/ ToB)
                      ELSE /\ ToB

Spec == Sinit /\ [][Snext]_vars /\ WF_vars(ToB)

Batman == s = "Batman" /\ count = 0

Liveness == <>Batman
```



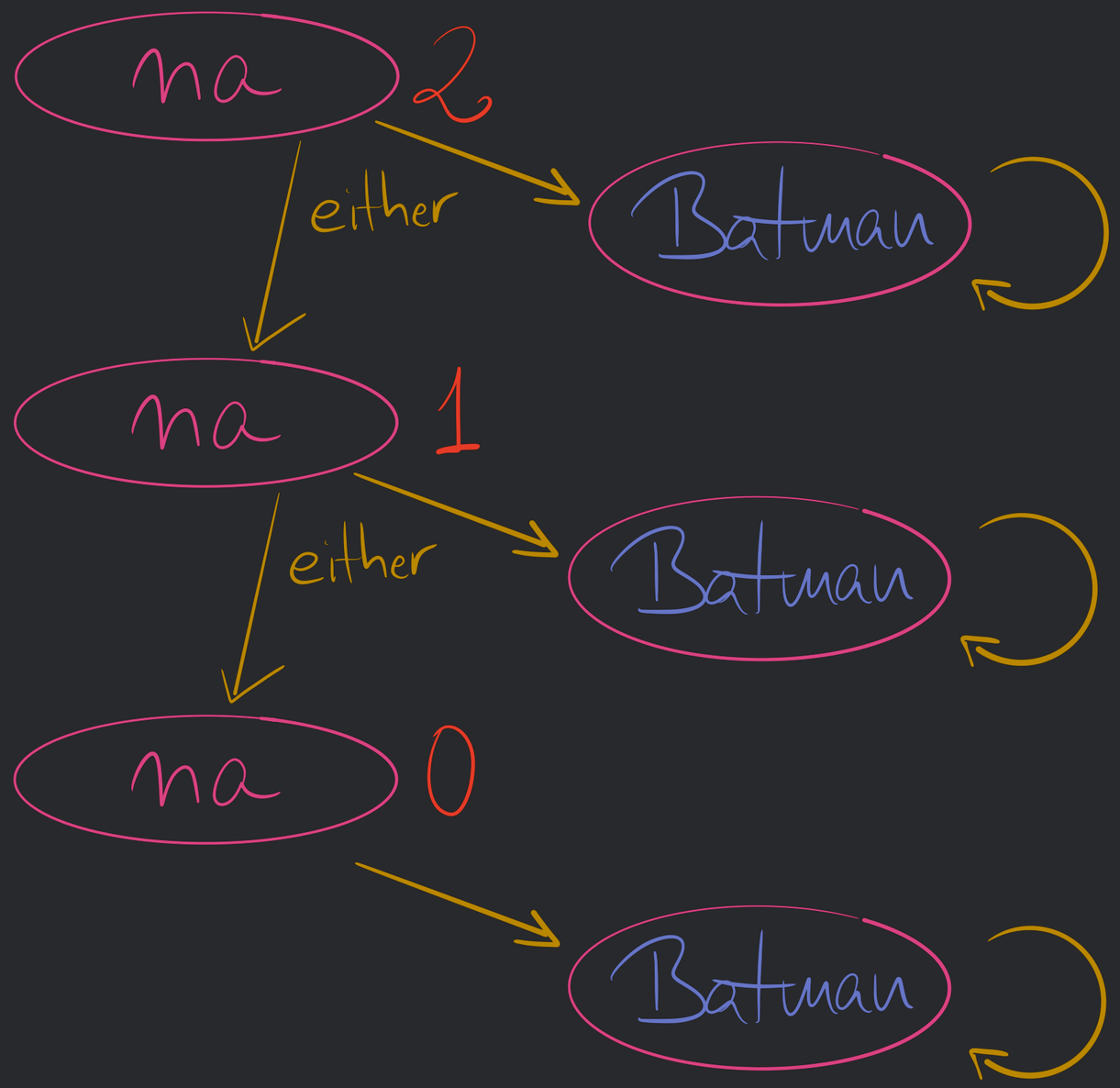
When this branch is considered, all future states are the same

```
-- MODULE ThemeSong --

EXTENDS Integers

VARIABLES s, count

vars == <<s, count>>

Sinit == s = "na" /\ count = 2

Na == s' = "na" /\ count' = count - 1
ToB == s' = "Batman" /\ count' = 0

Snext == IF count > 0 THEN /\ ( Na \/ ToB)
                      ELSE /\ ToB

Spec == Sinit /\ [][Snext]_vars /\ WF_vars(ToB)

Batman == s = "Batman" /\ count = 0

Liveness == <>Batman
```

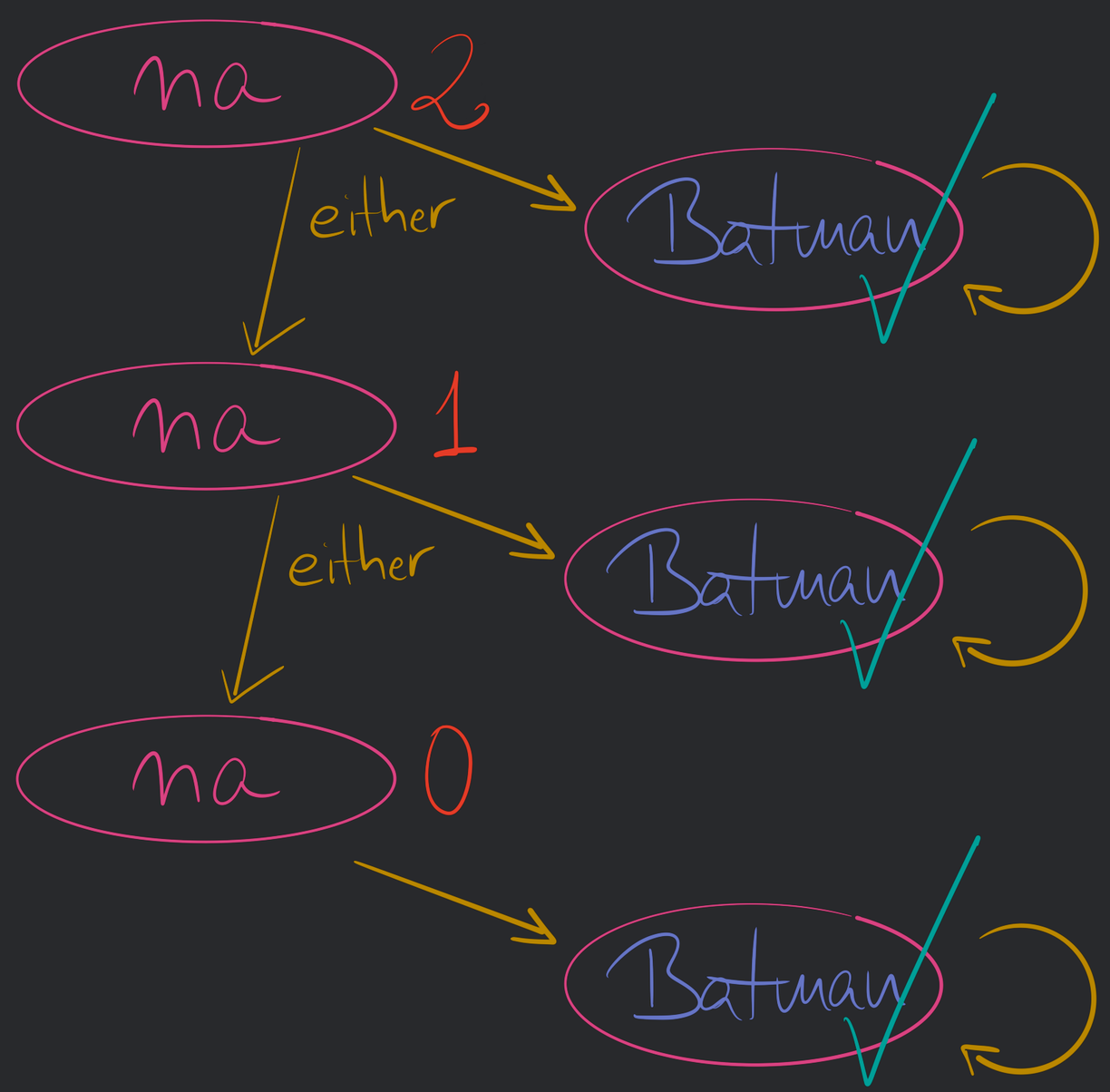


The other branch decreases the counter, and then we can repeat

```
-- MODULE ThemeSong --

EXTENDS Integers

VARIABLES s, count

vars == <<s, count>>

Sinit == s = "na" /\ count = 2

Na == s' = "na" /\ count' = count - 1
ToB == s' = "Batman" /\ count' = 0

Snext == IF count > 0 THEN /\ ( Na \/ ToB)
                      ELSE /\ ToB

Spec == Sinit /\ [][Snext]_vars /\ WF_vars(ToB)

Batman == s = "Batman" /\ count = 0

Liveness == <>Batman
```



```
-- MODULE ThemeSong --

EXTENDS Integers

VARIABLES s, count

vars == <<s, count>>

Sinit == s = "na" /\ count = 2

Na == s' = "na" /\ count' = count - 1
ToB == s' = "Batman" /\ count' = 0

Snext == IF count > 0 THEN /\ ( Na \/ ToB)
                      ELSE /\ ToB

Spec == Sinit /\ [][Snext]_vars /\ WF_vars(ToB)

Batman == s = "Batman" /\ count = 0

Liveness == <>Batman
```



And this is a condition we can request TLC to verify: eventually the state of the system has "Batman" and 0

HOW WOULD THE EQUIVALENT PLUSCAL ALGORITHM LOOK LIKE?

```
(*--fair algorithm ThemeSong {
    variables count = 2, s = "na";
      {
        while (count > 0) {
            either {
                s := "Batman";
                count := 0;
            } or {
                s := "na";
                count := count - 1;
            }
        };
        if (count = 0) {
            s := "Batman";
        }
      }
   }
} -*)
```

Next slide will contain horrible autogenerated code

```
(*--fair algorithm ThemeSong {
    variables count = 2, s = "na";
      {
        while (count > 0) {
            either {
                s := "Batman";
                count := 0;
            } or {
                s := "na";
                count := count - 1;
            }
        };
        if (count = 0) {
            s := "Batman";
        }
      }
   }
} -*)
```

Next slide will contain horrible autogenerated code

```
(*--fair algorithm ThemeSong {
    variables count = 2, s = "na";
      {
        while (count > 0) {
            either {
                s := "Batman";
                count := 0;
            } or {
                s := "na";
                count := count - 1;
            }
        };
        if (count = 0) {
            s := "Batman";
        }
      }
   }
} -*)
```

Next slide will contain horrible autogenerated code

```
(*--fair algorithm ThemeSong {
    variables count = 2, s = "na";
      {
        while (count > 0) {
            either {
                s := "Batman";
                count := 0;
            } or {
                s := "na";
                count := count - 1;
            }
        };
        if (count = 0) {
            s := "Batman";
        }
      }
   }
} -*)
```

Next slide will contain horrible autogenerated code

```
VARIABLES count, s, pc

vars == << count, s, pc >>

Init == /\ count = 2
        /\ s = "na"
        /\ pc = "ThemeSong"

ThemeSong == /\ pc = "ThemeSong"
             /\ IF count > 0
                   THEN /\ \/ /\ s' = "Batman"
                              /\ count' = 0
                           \/ /\ s' = "na"
                              /\ count' = count - 1
                        /\ pc' = "ThemeSong"
                   ELSE /\ IF count = 0
                              THEN /\ s' = "Batman"
                              ELSE /\ TRUE
                                   /\ s' = s
                        /\ pc' = "Done"
                        /\ count' = count

Next == ThemeSong
           \/ (pc = "Done" /\ UNCHANGED vars)

Spec == /\ Init /\ [][Next]_vars /\ WF_vars(Next)
```

WE CAN "EASILY" MODEL WHOLE SYSTEMS USING PLUSCAL

LIKE . . .

```
case class Trigger(payload: Payload)

class SimpleActor extends Actor {
  def receive = {
    case Trigger(payload) => ThirdParty().trigger(payload)
    case _                => log.warn("Duh")
  }
}
```

Here we have a very, very simple actor that on receiving a `Trigger` message, will ping a third party with the paylod.

WITH PLUSCAL WE CAN MODEL CONCURRENT (OR NOT) SYSTEMS BY USING `process`

Although can express more models than PlusCal can, it's way easier to define sequential processes running possibly in parallel in PlusCal

```
variables actorInboxes = [ actor |-> <<>> ];
          triggered = FALSE;

procedure trigger(trigger_content="?") {
    triggerLabel:
      triggered := TRUE;
      return;
}

fair process (actor = "actor")
variables currentMessage = <<"?", "no_content">>;
  kind = "?";
  content = "no_content";
{
  Send:
    actorInboxes["actor"] := Append(actorInboxes["actor"], <<"trigger", "foo">>);
  WaitForMessages:+
    while (TRUE) {
      if (actorInboxes["actor"] /= <<>>) {
         currentMessage := Head(actorInboxes["actor"]);
         content := Head(Tail(currentMessage));
         kind := Head(currentMessage);
         actorInboxes["actor"] := Tail(actorInboxes["actor"]);
        };
        ProcessMessage:
         if (kind = "trigger") {
           call trigger(content);
         }
    }
}
```

This is how an actor system roughly equivalent to the previous actor would look like. I'll break it up in the next slides

```
variables actorInboxes = [ actor |-> <<>> ];
          triggered = FALSE;

procedure trigger(trigger_content="?") {
    triggerLabel:
      triggered := TRUE;
      return;
}
```

This just sets up some inbox system, ignores the fact that the third party provider may fail receiving the request and sets the scene for checking some property

```
variables actorInboxes = [ actor |-> <<>> ];
          triggered = FALSE;

procedure trigger(trigger_content="?") {
    triggerLabel:
      triggered := TRUE;
      return;
}
```

This just sets up some inbox system, ignores the fact that the third party provider may fail receiving the request and sets the scene for checking some property

```
fair process (actor = "actor")
variables currentMessage = <<"?", "no_content">>;
  kind = "?";
  content = "no_content";
{
  Send:
    actorInboxes["actor"] := Append(actorInboxes["actor"], <<"trigger", "foo">>);
  WaitForMessages:+
    while (TRUE) {
      if (actorInboxes["actor"] /= <<>>) {
         currentMessage := Head(actorInboxes["actor"]);
         content := Head(Tail(currentMessage));
         kind := Head(currentMessage);
         actorInboxes["actor"] := Tail(actorInboxes["actor"]);
        };
        ProcessMessage:
         if (kind = "trigger") {
           call trigger(content);
         }
    }
}
```

This is a possible approach of what the actor might look like. It starts by seeding the inbox with a trigger (you can alternatively add a different process to do so, but this is better when you don't care about the source). Then it calls the trigger process if it *receive*s a trigger message

```
fair process (actor = "actor")
variables currentMessage = <<"?", "no_content">>;
  kind = "?";
  content = "no_content";
{
  Send:
    actorInboxes["actor"] := Append(actorInboxes["actor"], <<"trigger", "foo">>);
  WaitForMessages:+
    while (TRUE) {
      if (actorInboxes["actor"] /= <<>>) {
         currentMessage := Head(actorInboxes["actor"]);
         content := Head(Tail(currentMessage));
         kind := Head(currentMessage);
         actorInboxes["actor"] := Tail(actorInboxes["actor"]);
        };
        ProcessMessage:
         if (kind = "trigger") {
           call trigger(content);
         }
    }
}
```

This is a possible approach of what the actor might look like. It starts by seeding the inbox with a trigger (you can alternatively add a different process to do so, but this is better when you don't care about the source). Then it calls the trigger process if it *receive*s a trigger message

```
fair process (actor = "actor")
variables currentMessage = <<"?", "no_content">>;
  kind = "?";
  content = "no_content";
{
  Send:
    actorInboxes["actor"] := Append(actorInboxes["actor"], <<"trigger", "foo">>);
  WaitForMessages:+
    while (TRUE) {
      if (actorInboxes["actor"] /= <<>>) {
         currentMessage := Head(actorInboxes["actor"]);
         content := Head(Tail(currentMessage));
         kind := Head(currentMessage);
         actorInboxes["actor"] := Tail(actorInboxes["actor"]);
        };
        ProcessMessage:
         if (kind = "trigger") {
           call trigger(content);
         }
    }
}
```

This is a possible approach of what the actor might look like. It starts by seeding the inbox with a trigger (you can alternatively add a different process to do so, but this is better when you don't care about the source). Then it calls the trigger process if it *receive*s a trigger message

```
fair process (actor = "actor")
variables currentMessage = <<"?", "no_content">>;
  kind = "?";
  content = "no_content";
{
  Send:
    actorInboxes["actor"] := Append(actorInboxes["actor"], <<"trigger", "foo">>);
  WaitForMessages:+
    while (TRUE) {
      if (actorInboxes["actor"] /= <<>>) {
         currentMessage := Head(actorInboxes["actor"]);
         content := Head(Tail(currentMessage));
         kind := Head(currentMessage);
         actorInboxes["actor"] := Tail(actorInboxes["actor"]);
        };
        ProcessMessage:
         if (kind = "trigger") {
           call trigger(content);
         }
    }
}
```

This is a possible approach of what the actor might look like. It starts by seeding the inbox with a trigger (you can alternatively add a different process to do so, but this is better when you don't care about the source). Then it calls the trigger process if it *receive*s a trigger message

```
fair process (actor = "actor")
variables currentMessage = <<"?", "no_content">>;
  kind = "?";
  content = "no_content";
{
  Send:
    actorInboxes["actor"] := Append(actorInboxes["actor"], <<"trigger", "foo">>);
  WaitForMessages:+
    while (TRUE) {
      if (actorInboxes["actor"] /= <<>>) {
         currentMessage := Head(actorInboxes["actor"]);
         content := Head(Tail(currentMessage));
         kind := Head(currentMessage);
         actorInboxes["actor"] := Tail(actorInboxes["actor"]);
        };
        ProcessMessage:
         if (kind = "trigger") {
           call trigger(content);
         }
    }
}
```

This is a possible approach of what the actor might look like. It starts by seeding the inbox with a trigger (you can alternatively add a different process to do so, but this is better when you don't care about the source). Then it calls the trigger process if it *receive*s a trigger message

A POSSIBLE CHECK ON THIS SYSTEM COULD THEN BE

```
Triggered == triggered = TRUE

Liveness == <>Triggered
```

We could for instance check that eventually the third party system is triggered. This is what I did in my problem in the beginning: Do the action happen on the remote server, and the confirmation eventually makes it into the actor system? In a space of around 14 thousand states, the answer was yes.


QUESTIONS?


THANKS!

FURTHER REFERENCES

DOWNLOADING THE TLA TOOLBOX

SPECIFYING SYSTEMS BY LESLIE LAMPORT[SSLL]

PRACTICAL TLA+ BY HILLEL WAYNE[PTHW]

[SSLL] THIS IS THE MAIN REFERENCE FOR TLA+. AND ALTHOUGH IT CAN LOOK A BIT DENSE IT READS VERY WELL.

[PTHW] THIS IS A LIGHTHER FIRST REFERENCE, FOCUSED IN PLUSCAL (USING THE P SYNTAX). I STARTED WITH THIS BOOK, AND WITH IT YOU CAN EASILY GET ENOUGH PLUSCAL TO DEFINE YOUR SPECIFICATIONS AND VALIDATE YOUR MODELS

TLA+ VIDEO COURSE BY LESLIE LAMPORT[TVC]

LEARN TLA ONLINE BOOK BY HILLEL WAYNE[LTHW]

[TVC] THIS IS A VIDEO COURSE BY LAMPORT. TOTAL OF AROUND 4 HOURS

[LTHW] AN ONLINE BOOK BY HILLEL WAYNE COVERING PARTS OF PRACTICAL TLA+

EOF