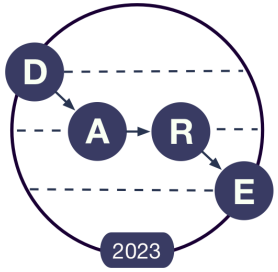


Modelling and validating distributed systems with TLA+

Carla Ferreira

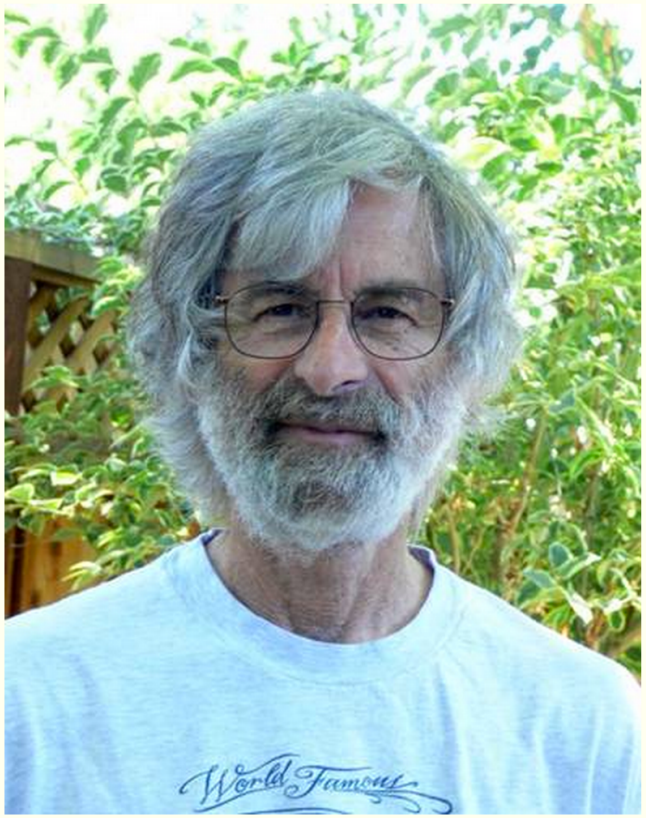
NOVA University Lisbon

13th September 2023

TLA+ specification language

- Formal language for describing and reasoning about distributed and concurrent systems.
- TLA+ is a model-oriented language:
  - based on mathematical logic and set theory plus temporal logic TLA (temporal logic of actions).
- Supported by the TLA Toolbox.
- References:
  - TLA+ Hyperbook (http://research.microsoft.com/en-us/um/people/lamport/tla/hyperbook.html)
  - TLA+ web page (http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html)

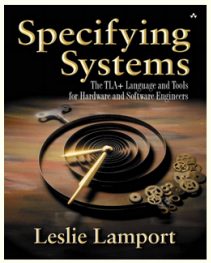
LESLIE LAMPORT'S HOME PAGE



NEW: TLA+ Use at Amazon

The TLA Web Page

My Collected Works



Turing Award 2013

For fundamental contributions to the theory and practice of distributed and concurrent systems, notably the invention of concepts such as causality and logical clocks, safety and liveness, replicated state machines, and sequential consistency.

Use of TLA+ at Amazon

“We have used TLA+ on 10 large complex real-world systems. In every case TLA+ has added significant value, either finding subtle bugs that we are sure we would not have found by other means, or giving us enough understanding and confidence to make aggressive performance optimizations without sacrificing correctness.“

First TLA+ Example

1-bit Clock

- Clock's possible behaviours:

$$b = 1 \longrightarrow b = 0 \longrightarrow b = 1 \longrightarrow b = 0 \longrightarrow \ldots$$

$$b = 0 \longrightarrow b = 1 \longrightarrow b = 0 \longrightarrow b = 1 \longrightarrow \ldots$$

1-bit Clock

- State variable:

  b

- Initial predicate:

  $b = 1 \lor b = 0$

- Next-step action (b' is the variable at the next state):

  $\lor (b = 0) \land (b' = 1)$
  $\lor (b = 1) \land (b' = 0)$

The initial state and next-step action are formulas in TLA

1-bit Clock

- State variable:

  b

- Initial predicate:

  $b = 1 \lor b = 0$

- Next-step action (b' is the variable at the next state):

  IF b = 0 THEN b' = 1
  ELSE b' = 0

The initial state and next-step action are formulas in TLA

1-bit Clock: TLA specification

```
---------------------------- MODULE OneBitClock ----------------------------
VARIABLE b

Init == (b=0) \/ (b=1)



Next == \/ b = 0 /\ b' = 1
        \/ b = 1 /\ b' = 0




==========================================================================
```

What about the clock properties?

System's properties

- Safety
  - Something bad never happens
  - E.g. system never deadlocks, the account balance is greater or equal to zero
- Liveness
  - Something good eventually happens
  - E.g. if a process request access to a critical region it will eventually be granted access, the light will eventually turn green

Let's ignore liveness properties for now

1-bit Clock: TLA specification

```
---------------------------- MODULE OneBitClock ----------------------------
VARIABLE b

Init == (b=0) \/ (b=1)

TypeInv == b \in {0,1}

Next == \/ b = 0 /\ b' = 1
        \/ b = 1 /\ b' = 0

Spec == Init /\ [][Next]_<<b>>
----------------------------------------------------------------------------

THEOREM Spec => []TypeInv

============================================================================
```

Typing information (TLA+ is untyped)

1-bit Clock: TLA specification

```
---------------------------- MODULE OneBitClock ----------------------------
VARIABLE b

Init == (b=0) \/ (b=1)

TypeInv == b \in {0,1}

Next == \/ b = 0 /\ b' = 1
        \/ b = 1 /\ b' = 0

Spec == Init /\ [][Next]_<<b>>

THEOREM Spec => []TypeInv

=============================================================================
```

```
[A]_<<f>> == A \/ (f' = f)
```

The initial state satisfies *Init*
Every transition satisfies *Next* or leaves *b* unchanged

1-bit Clock: TLA specification

```
---------------------------- MODULE OneBitClock ----------------------------
VARIABLE b

Init == (b=0) \/ (b=1)

TypeInv == b \in {0,1}

Next == \/ b = 0 /\ b' = 1
        \/ b = 1 /\ b' = 0

Spec == Init /\ [][Next]_<<b>>
----------------------------------------------------------------------------

THEOREM Spec => []TypeInv

============================================================================
```

Theorem specifies an invariant property

TLC model checker

- Exhaustive breath-first search of all reachable states
  - Finds (one of) the shortest path to the property violation

Computing all possible behaviours

- State graph is a directed graph G

  1. Put into G to the set of all initial states

  2. For every state s in G compute all possible states t such that $s \longrightarrow t$ can be a step in a behaviour

  3. For every state t found in step 2 not in G, draw an edge from s to t

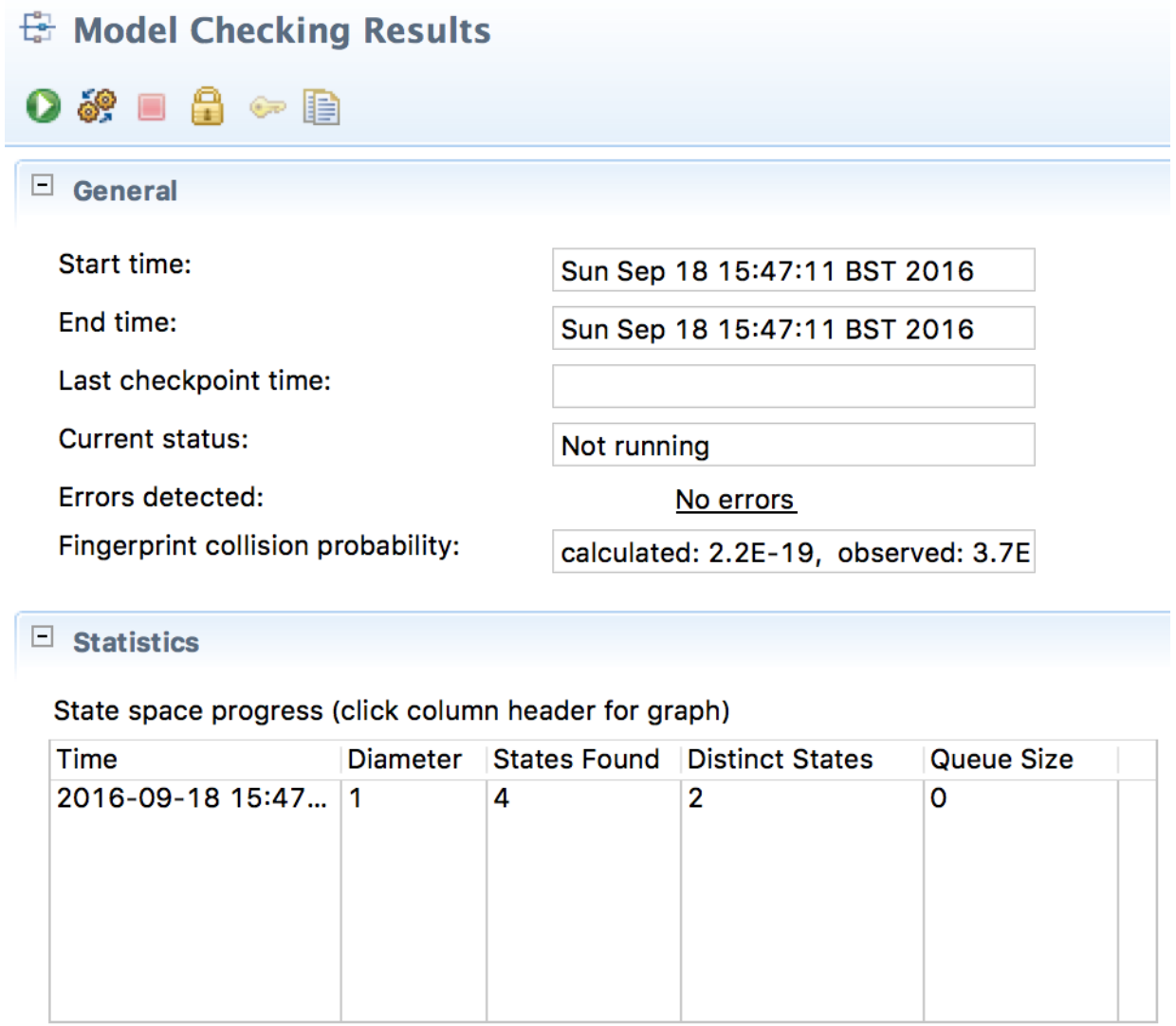
  4. Repeat the previous steps until no new states or edges can be added to G

TLC: state space progress

- Diameter
  - Number of states in the longest path of G with no repeated states
- States found
  - Total number of states it examined in step 1 and 2
- Distinct states
  - Number of states that form the set of nodes of G
- Queue size
  - Number of states s in G for which step 2 has not yet been explored

1-bit Clock: Model checking

- Checking the 1-bit clock with TLC model checker (demo)



Verifying a producer-consumer a protocol

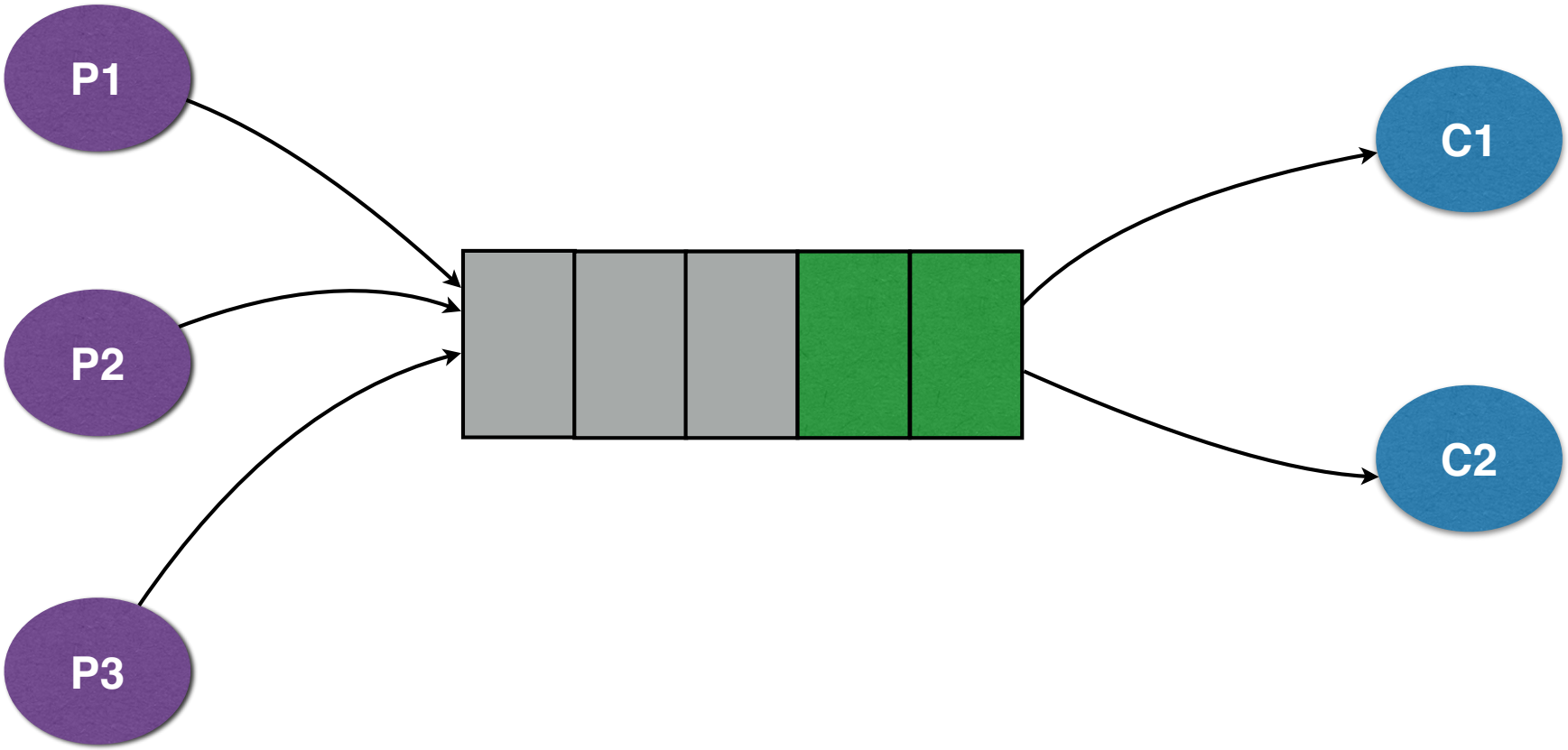
Producer-Consumer Protocol

- System of producers and consumers threads
- Communicate with each other through a shared buffer
  - Producer threads produce data
  - Consumer threads consume data

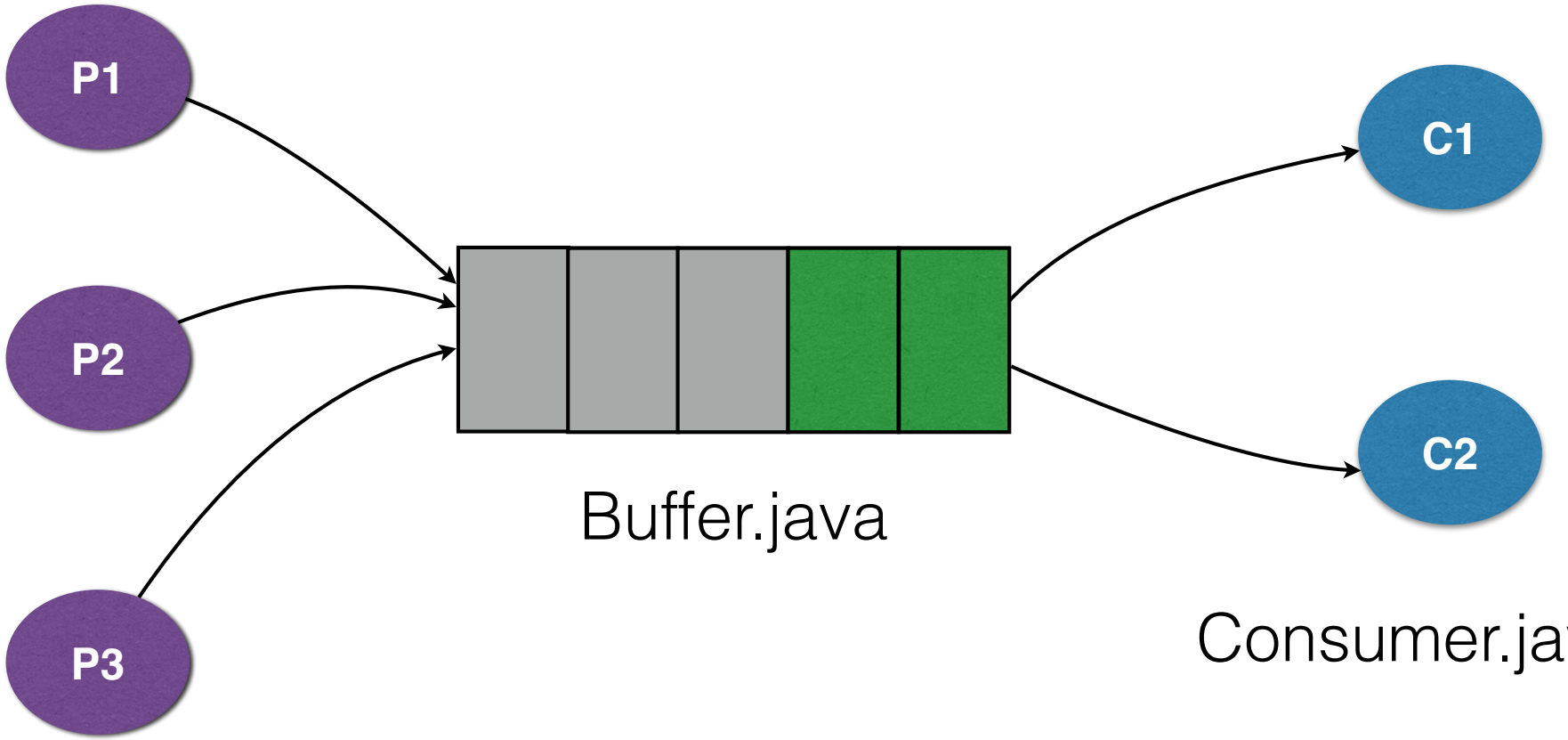
Producer-Consumer Protocol

- P producer threads
- C consumer threads
- Bounded buffer with N entries
- Requirements:
  - Producers block when buffer is full
  - Consumers block when buffer is empty

Producer-Consumer Protocol



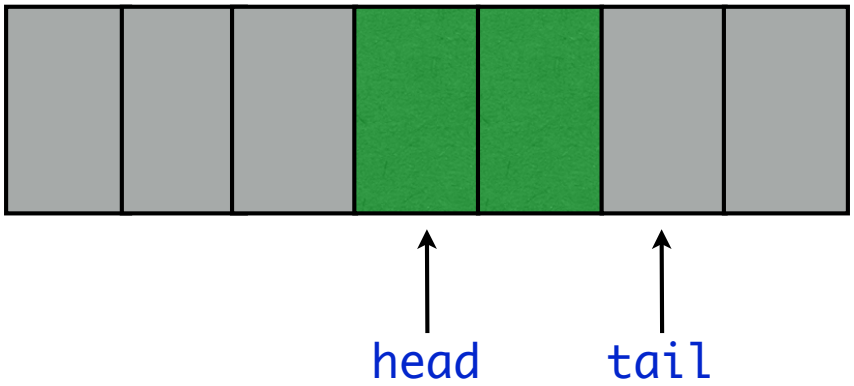
Producer-Consumer Protocol



Buffer.java

```
public class Buffer<E> {
    public final int capacity;
    private final E[] store;
    private int head, tail, size;

    ...

}
```

Circular vector



Buffer.java

```
public synchronized void put (E e) {
   while (isFull())
      wait();
   store[tail] = e;
   tail = next(tail);
   size++;
   notify();
}
```

There is a bug in the code!

```
public class Buffer<E> {
    private final E[] store;
    private int head, tail, size;

    public synchronized void put (E e) {
        while (isFull())
            wait();
        notify();
        store[tail] = e;
        tail = next(tail);
        size++;
    }

    public synch
        whil
                                ad];
                    ] = null;
            = next(head);
        size--;
        return e;
    }
}
```

System deadlocks!

It reaches a state where all threads are blocked

How to test concurrent code?

- Errors in concurrent programs are difficult to reproduce
  - Simulate nondeterministic interleaving of threads
  - Rely on luck :-(

Tests vs TLC model checker

Buffer entries	Producers	Consumers	Test found a deadlock	TLC found a deadlock
10	5	5	NO	**NO**
5	10	3	YES	**YES**
2	3	2	NO	**YES**

TLA+ specification: Constants

```
---------------------------- MODULE BoundedBuffer ----------------------------
EXTENDS Naturals, Sequences

CONSTANTS Producers,   (* the set of producers *)
          Consumers,   (* the set of consumers *)
          Capacity,    (* the capacity of the buffer *)
          Data         (* the set of values that can be produced and consumed *)

ASSUME
    /\ Producers /= {}                          (* at least one producer *)
    /\ Consumers /= {}                          (* at least one consumer *)
    /\ Producers \intersect Consumers = {}      (* no thread is both consumer and
                                                   producer *)
    /\ Capacity > 0                             (* buffer capacity is at least 1 *)
    /\ Data /= {}                               (* the type of data is nonempty  *)
-----------------------------------------------------------------------------
```

TLA+ specification: Variables

```
VARIABLES buffer, (* the buffer *)
          waiting (* the wait set *)

Participants == Producers \union Consumers
RunningThreads == Participants \ waiting

TypeInv == /\ buffer \in Seq(Data) (* the buffer, as a sequence of objects *)
           /\ Len(buffer) \in 0..Capacity
           /\ waiting \subseteq Participants

Notify == IF waiting /= {}          (* corresponds to method notify in Java *)
          THEN \E x \in waiting : waiting' = waiting \ {x}
          ELSE UNCHANGED waiting

Wait(t) == waiting' = waiting \union {t} (* corresponds to method wait in Java *)
```

TLA+ specification: Actions

```
Init == buffer = <<>> /\ waiting = {}

Put(t,m) == IF Len(buffer) < Capacity
            THEN /\ buffer' = Append(buffer, m)
                 /\ Notify
            ELSE /\ Wait(t)
                 /\ UNCHANGED buffer
```

Java code

```
public synchronized void put (E e) {
      while (isFull())
         wait();
      store[tail] = e;
      tail = next(tail);
      size++;
      notify();
}
```

TLA+ specification: Actions

```
Init == buffer = <<>> /\ waiting = {}

Put(t,m) == IF Len(buffer) < Capacity
            THEN /\ buffer' = Append(buffer, m)
                 /\ Notify
            ELSE /\ Wait(t)
                 /\ UNCHANGED buffer

Get(t) == IF Len(buffer) > 0
          THEN /\ buffer' = Tail(buffer)
               /\ Notify
          ELSE /\ Wait(t)
               /\ UNCHANGED buffer
```

Java code

```
public synchronized E get () {
    while (isEmpty())
        wait();
    E e = store[head];
    store[head] = null;
    head = next(head);
    size--;
    notify();
    return e;
}
```

TLA+ specification: Actions

```
Init == buffer = <<>> /\ waiting = {}

Put(t,m) == IF Len(buffer) < Capacity
            THEN /\ buffer' = Append(buffer, m)
                 /\ Notify
            ELSE /\ Wait(t)
                 /\ UNCHANGED buffer

Get(t) == IF Len(buffer) > 0
          THEN /\ buffer' = Tail(buffer)
               /\ Notify
          ELSE /\ Wait(t)
               /\ UNCHANGED buffer

Next == \E t \in RunningThreads : \/ t \in Producers /\ \E m \in Data : Put(t,m)
                                  \/ t \in Consumers /\ Get(t)

Spec == Init /\ [][Next]_<<buffer, waiting>>
```

TLA+ specification: Properties

```
NoDeadlock == [](RunningThreads # {})

THEOREM Spec => []TypeInv
```

Model checking with TLC

- Build several models:
  - 1st model
    - 5 producers, 5 consumers, buffer capacity is 10
  - 2nd model
    - 10 producers, 3 consumers, buffer capacity is 10
  - 3rd model
    - 3 producers, 2 consumers, buffer capacity is 5

Buffer entries	Producers	Consumers	Test found a deadlock	TLC found a deadlock
10	5	5	NO	**NO**
5	10	3	YES	**YES**
2	3	2	NO	**YES**

TLC model checker: 1st Model

```
SPECIFICATION Spec
CONSTANTS     Producers = {p1,p2,p3,p4,p5}
              Consumers = {c1,c2,c3,c4,c5}
              Capacity = 10
              Data = {m}
INVARIANT     TypeInv
PROPERTY      NoDeadlock
```

```
Starting... (2015-09-20 21:38:47)
Computing initial states...
Finished computing initial states: 1 distinct state generated.
Model checking completed. No error has been found.
3461 states generated, 223 distinct states found, 0 states left on queue.
The depth of the complete state graph search is 20.
Finished. (2015-09-20 21:38:48)
```

TLC model checker: 2nd Model

```
SPECIFICATION Spec
CONSTANTS     Producers = {p1,p2,p3,p4,p5,p6,p7,p8,p9,p10}
              Consumers = {c1,c2,c3}
              Capacity = 5
              Data = {m}
INVARIANT     TypeInv
PROPERTY      NoDeadlock
```

```
Starting... (2015-09-20 21:00:34)
Computing initial states...
Finished computing initial states: 1 distinct state generated.
Error: Invariant NoDeadlock is violated.
```

TLC model checker: 2nd Model

```
SPECIFICATION Spec
CONSTANTS     Producers = {p1,p2,p3,p4,p5,p6,p7,p8,p9,p10}
              Consumers = {c1,c2,c3}
              Capacity = 5
              Data = {m}
INVARIANT     TypeInv
PROPERTY      NoDeadlock
```

```
State 38: <Action line 54, col 9 to line 55, col 62 of module BoundedBuffer>
/\ buffer = <<m, m, m, m, m>>
/\ waiting = {p1, p2, p3, p4, p5, p6, p7, p8, p9, c1, c2, c3}

State 39: <Action line 54, col 9 to line 55, col 62 of module BoundedBuffer>
/\ buffer = <<m, m, m, m, m>>
/\ waiting = {p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, c1, c2, c3}

1166149 states generated, 37138 distinct states found, 1556 states left on queue.
The depth of the complete state graph search is 39.
Finished. (2015-09-20 21:00:38)
```

TLC model checker: 3rd Model

```
SPECIFICATION Spec
CONSTANTS     Producers = {p1,p2,p3}
              Consumers = {c1,c2}
              Capacity = 2
              Data = {m}
INVARIANT     TypeInv
PROPERTY      NoDeadlock
```

```
Starting... (2015-09-20 21:10:26)
Computing initial states...
Finished computing initial states: 1 distinct state generated.
Error: Invariant NoDeadlock is violated.
```

TLC model checker: 3rd Model

```
SPECIFICATION Spec
CONSTANTS     Producers = {p1,p2,p3}
              Consumers = {c1,c2}
              Capacity = 2
              Data = {m}
INVARIANT     TypeInv
PROPERTY      NoDeadlock
```

```
State 23: <Action line 54, col 9 to line 55, col 62 of module BoundedBuffer>
/\ buffer = <<m, m>>
/\ waiting = {p1, p2, c1, c2}

State 24: <Action line 54, col 9 to line 55, col 62 of module BoundedBuffer>
/\ buffer = <<m, m>>
/\ waiting = {p1, p2, p3, c1, c2}

385 states generated, 86 distinct states found, 3 states left on queue.
The depth of the complete state graph search is 24.
Finished. (2015-09-20 21:10:29)
```

TLC model checker: 3rd Model

```
S1:  buffer = <<>>  /\ waiting = {} /\ running = {p1, p2, p3, c1, c2}
S2:  buffer = <<m>> /\ waiting = {} /\ running = {p1, p2, p3, c1, c2}
...
```

TLC model checker: 3rd Model

```
S1:  buffer = <<>>  /\ waiting = {} /\ running = {p1, p2, p3, c1, c2}
S2:  buffer = <<m>> /\ waiting = {} /\ running = {p1, p2, p3, c1, c2}
...
S8:  buffer = <<>>   /\ waiting = {p3}         /\ running = {p1, p2, c1, c2}
S9:  buffer = <<>>   /\ waiting = {p3, c1}     /\ running = {p1, p2, c2}
S10: buffer = <<>>   /\ waiting = {p3, c1, c2} /\ running = {p1, p2}
S11: buffer = <<m>>  /\ waiting = {c1, c2}     /\ running = {p1, p2, p3}
...
```

TLC model checker: 3rd Model

```
S1:  buffer = <<>>  /\ waiting = {} /\ running = {p1, p2, p3, c1, c2}
S2:  buffer = <<m>> /\ waiting = {} /\ running = {p1, p2, p3, c1, c2}
...
S8:  buffer = <<>>   /\ waiting = {p3}         /\ running = {p1, p2, c1, c2}
S9:  buffer = <<>>   /\ waiting = {p3, c1}     /\ running = {p1, p2, c2}
S10: buffer = <<>>   /\ waiting = {p3, c1, c2} /\ running = {p1, p2}
S11: buffer = <<m>>  /\ waiting = {c1, c2}     /\ running = {p1, p2, p3}
...
```

TLC model checker: 3rd Model

```
S1:  buffer = <<>>  /\ waiting = {} /\ running = {p1, p2, p3, c1, c2}
S2:  buffer = <<m>> /\ waiting = {} /\ running = {p1, p2, p3, c1, c2}
...
S8:  buffer = <<>>   /\ waiting = {p3}         /\ running = {p1, p2, c1, c2}
S9:  buffer = <<>>   /\ waiting = {p3, c1}     /\ running = {p1, p2, c2}
S10: buffer = <<>>   /\ waiting = {p3, c1, c2} /\ running = {p1, p2}
S11: buffer = <<m>>  /\ waiting = {c1, c2}     /\ running = {p1, p2, p3}
...
```

TLC model checker: 3rd Model

```
S1:  buffer = <<>>  /\ waiting = {} /\ running = {p1, p2, p3, c1, c2}
S2:  buffer = <<m>> /\ waiting = {} /\ running = {p1, p2, p3, c1, c2}
...
S8:  buffer = <<>>   /\ waiting = {p3}          /\ running = {p1, p2, c1, c2}
S9:  buffer = <<>>   /\ waiting = {p3, c1}      /\ running = {p1, p2, c2}
S10: buffer = <<>>   /\ waiting = {p3, c1, c2}  /\ running = {p1, p2}
S11: buffer = <<m>>  /\ waiting = {c1, c2}      /\ running = {p1, p2, p3}
...
```

TLC model checker: 3rd Model

```
S1:  buffer = <<>>  /\ waiting = {} /\ running = {p1, p2, p3, c1, c2}
S2:  buffer = <<m>> /\ waiting = {} /\ running = {p1, p2, p3, c1, c2}
...
S8:  buffer = <<>>   /\ waiting = {p3}         /\ running = {p1, p2, c1, c2}
S9:  buffer = <<>>   /\ waiting = {p3, c1}     /\ running = {p1, p2, c2}
S10: buffer = <<>>   /\ waiting = {p3, c1, c2} /\ running = {p1, p2}
S11: buffer = <<m>>  /\ waiting = {c1, c2}     /\ running = {p1, p2, p3}
...
S19: buffer = <<>>     /\ waiting = {p2,p3,c1,c2}    /\ running = {p1}
S20: buffer = <<m>>    /\ waiting = {p3,c1,c2}       /\ running = {p1, p2}
S21: buffer = <<m, m>> /\ waiting = {c1,c2}          /\ running = {p1, p2, p3}
S22: buffer = <<m, m>> /\ waiting = {p1,c1,c2}       /\ running = {p2, p3}
S23: buffer = <<m, m>> /\ waiting = {p1,p2,c1,c2}    /\ running = {p3}
S24: buffer = <<m, m>> /\ waiting = {p1,p2,p3,c1,c2} /\ running = {}
```

Analyse the trace

- Producers and consumers wait on the same wait set
- When method `get` calls `notify()`
  - It should notify a producer (buffer has a slot available)
  - Occasionally it notifies a consumer
- When method `put` calls `notify()`
  - It should notify a consumer (buffer has data available)
  - Occasionally it notifies a producer

Analyse the trace

- Producers and consumers wait on the same wait set
- When method `get` calls `notify()`
  - It should notify a producer (buffer not available)
  - Occasionally it notifies a consumer
- When method `put` calls `notify()`
  - It should notify a consumer (buffer has data available)
  - Occasionally it notifies a producer

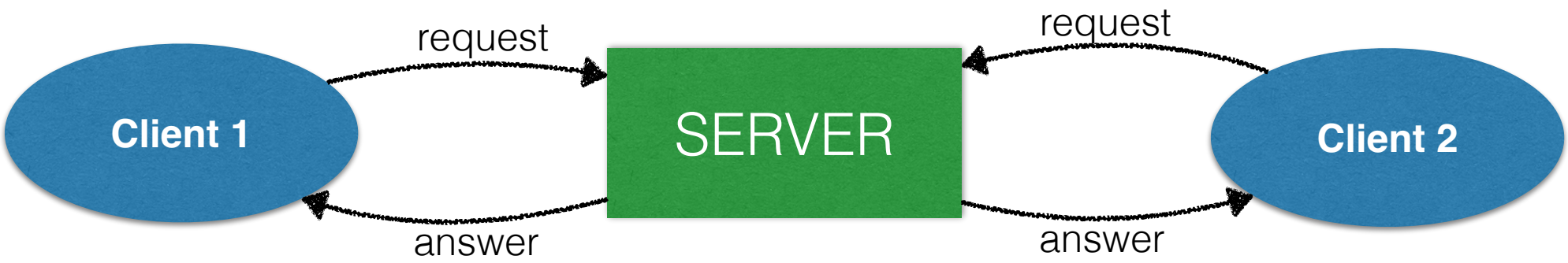
May lead to a deadlock!

Next steps

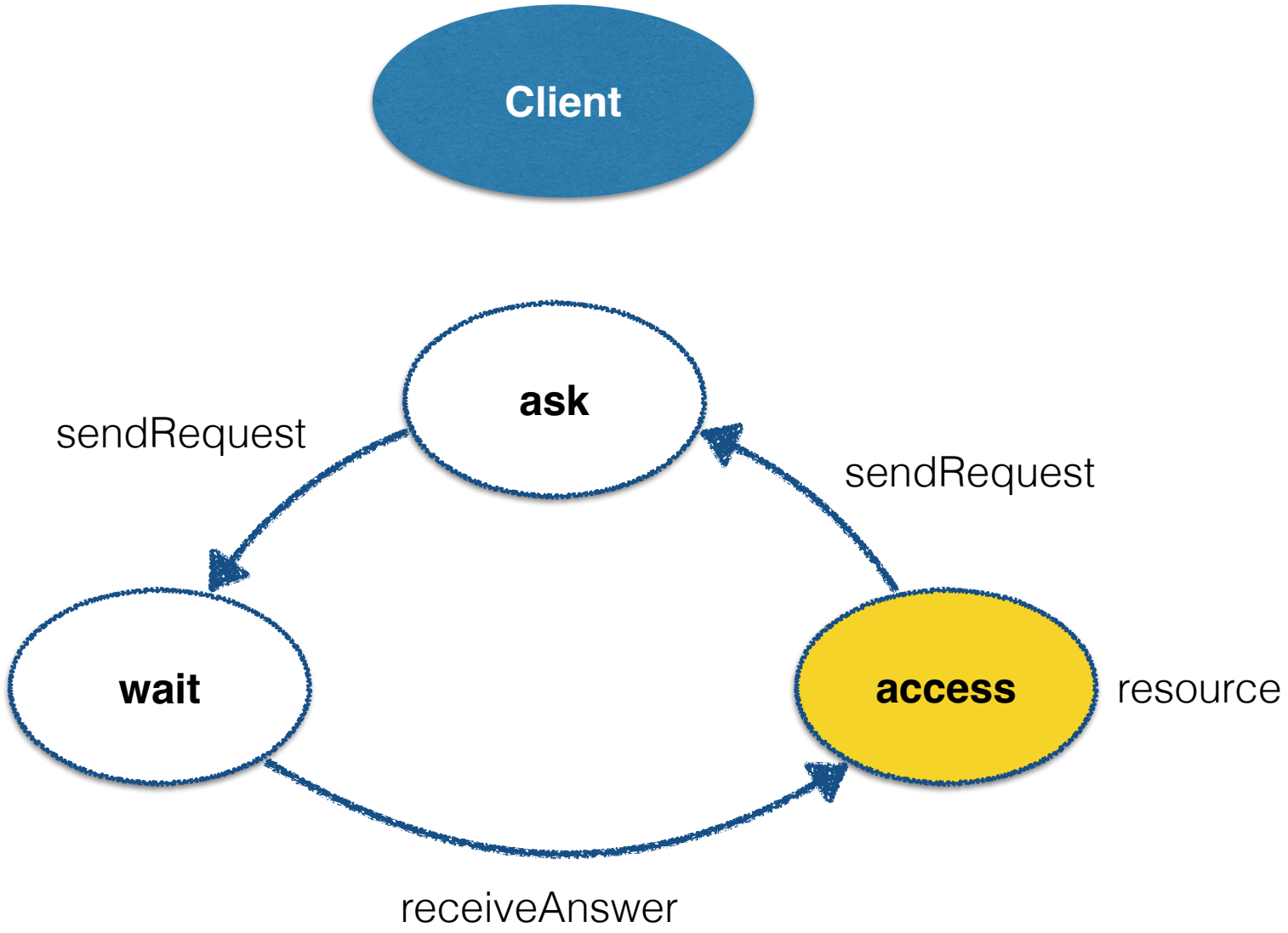
- In the TLA+ model replace every call to `notify` by a call to `notifyAll`
- Model check with TLC the updated model
- Fix the java code

Client-Server System

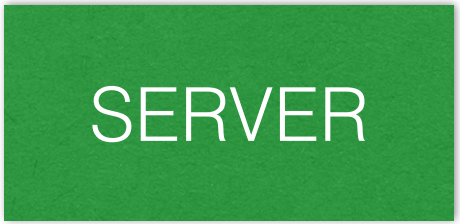
Distributed system of a server and two clients where the server maintains a shared resource which it grants to at most one of the clients at a time:



Client-Server System



Client-Server System



- Receives a request from a client to access the resource
  - Resource is free, assigns the resource to that client
  - Resource is busy, client is put on hold waiting on the resource
- Receives the release of the resource from a client
  - There is a client waiting for the resource, assign the resource to one of the clients waiting
  - No client is waiting for the resource, resource is free

TLA+ specification: Variables

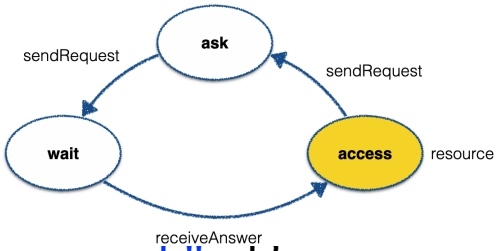
```
------------------------------ MODULE Server ------------------------------
EXTENDS Naturals

CONSTANT N

ASSUME N \in Nat

VARIABLES   (* Server variables *)
            current,    (* current client accessing the resource *)
            waiting,    (* clients waiting for accessing the resource *)
            sender,     (* message read by the server *)
            (* Client variables *)
            state,     (* state of clients *)
            (* Message variables *)
            comm        (* messages to and from the server *)

msg == [client: 1..N, type: {"start", "request", "answer", "release"}]

TypeInv ==
    /\ current \in 0..N
    /\ waiting \subseteq 1..N
    /\ sender \in 0..N
    /\ state \in  [1..N -> { "ask", "wait", "access", "release"}]
    /\ comm \in SUBSET msg

Init ==
    /\ current = 0 /\ waiting = {} /\ sender = 0 /\ comm = {}
    /\ state = [ c \in 1..N |-> "ask" ]
```

TLA+ specification

```
VARIABLES   ...
            waiting,        (* clients waiting for accessing the resource *)
            comm.           (* messages to and from the server *)

TypeInv == /\ ...
           /\ waiting \subseteq 1..N
           /\ comm \in SUBSET msg
```

- Design decision:
  - waiting is a set: the order of requests is lost
  - comm is a set: asynchronous communication without order

TLA+ specification: Actions




```
SendRequestClient(id) ==
    /\ state[id] = "ask"
    /\ state' = [state EXCEPT ![id] = "wait"]
    /\ comm' = comm \union {[ client |-> id, type |-> "request" ]}
    /\ UNCHANGED<<current,waiting,sender>>

ReceiveAnswerClient(id) ==
    /\ state[id] = "wait"
    /\ \E m \in comm : m.client = id /\ m.type = "answer"
    /\ state' = [state EXCEPT ![id] = "access"]
    /\ comm' = comm \ {[ client |-> id, type |-> "answer" ]}
    /\ UNCHANGED<<current,waiting,sender>>

SendReleaseClient(id) ==
    /\ state[id] = "access"
    /\ state' = [state EXCEPT ![id] = "ask"]
    /\ comm' = comm \union {[ client |-> id, type |-> "release" ]}
    /\ UNCHANGED<<current,waiting,sender>>
```

TLA+ specification: Actions

```
RcvRequestServer == (* there is at least one request message *)
    /\ \E m \in comm : sender' = m.client /\ m.type = "request"
    /\ current # sender' (* sender not the current client with the resource *)
    /\ IF current = 0
       THEN /\ current' = sender'
            /\ comm' = (comm \ {[ client |-> sender', type |-> "request" ]} )
                        \union {[ client |-> current', type |-> "answer" ]}
            /\ UNCHANGED<<waiting,state>>
       ELSE /\ comm' = comm \ {[ client |-> sender', type |-> "request" ]}
            /\ waiting' = waiting \union {sender'}
            /\ UNCHANGED<<current,state>>
```

TLA+ specification: Actions

```
RcvReleaseServer == (* there is a release message *)
    /\ \E m \in comm : sender' = m.client /\ m.type = "release"
    /\ current = sender'
    /\ IF waiting = {}
       THEN /\ current' = 0
            /\ comm' = comm \ {[ client |-> sender', type |-> "release" ]}
            /\ UNCHANGED<<waiting,state>>
       ELSE /\ \E c \in waiting : current' = c
            /\ waiting' = waiting \ {current'}
            /\ comm' = (comm \ {[ client |-> sender', type |-> "release" ]})
                       \union {[ client |-> current', type |-> "answer" ]}
            /\ UNCHANGED<<state>>
```

TLA+ specification: Actions

```
Client(c) == \/ SendRequestClient(c)
             \/ ReceiveAnswerClient(c)
             \/ SendReleaseClient(c)

Next ==  \E c \in 1..N :
           RcvReleaseServer \/ RcvRequestServer \/ Client(c)

Spec ==
    /\ Init
    /\ [][Next]_<<vars>>
```

TLA+ specification: Safety properties

```
MutualExclusion == \A c1, c2 \in 1..N :
                    state[c1] = "access" /\ state[c2] = "access" => c1 = c2

THEOREM Spec => []TypeInv /\ MutualExclusion
```

System's properties

- Safety
  - Something bad never happens
  - No two processes can access the critical section at the same time
- Liveness
  - Something good eventually happens
  - If a client wants to access a resource, it will eventually get access to that resource

Client-Server properties

- We defined following **safety properties**:
  - **Type correctness**: all state variables maintain reasonable types.
  - **Mutual exclusion**: there cannot be two clients accessing the resource at the same time.
- Now we are going to define **liveness properties**:
  - **Deadlock freedom**: if some client is trying to access the resource, then some process will eventually access the resource.
  - **Starvation freedom**: every client that wants to to access the resource eventually gets access to the resource.

TLA

- Temporal Logic of Actions:
  - Action formulas: describe state and state transitions
  - Temporal formulas: describe state sequences (traces)

Action Formulas

- `[A]<<e>> == A \/ e' = e`
- `<A><<e>> == A /\ ~(e' = e)`

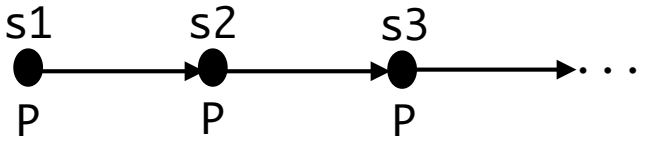
```
---------------------------------- MODULE Server ---------------------------------

...

Client(c) == SendRequestClient(c) \/ ReceiveAnswerClient(c) \SendReleaseClient(c)

Next ==  \E c \in 1..N : RcvReleaseServer \/ RcvRequestServer \/ Client(c)

Spec == Init /\ [][Next]_<<vars>>

---------------------------------------------------------------------------------
```

Temporal Formulas

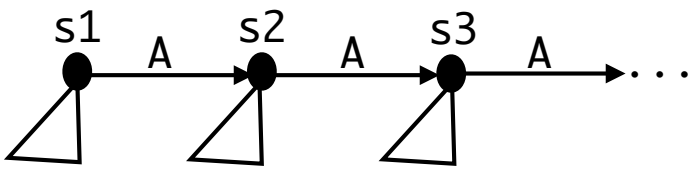
- Temporal operators
  - `[] F` — F is always true
  - `<> F` — F is eventually true
  - `F ~> G` — F leads to G
  - `WF_<<e>> (A)` — Weak fairness for action A
  - `SF_<<e>> (A)` — Strong fairness for action A

Temporal Formulas

- []F **F is always true**
  - Formula []P, where P is a state predicate, is true iff P is true in every state



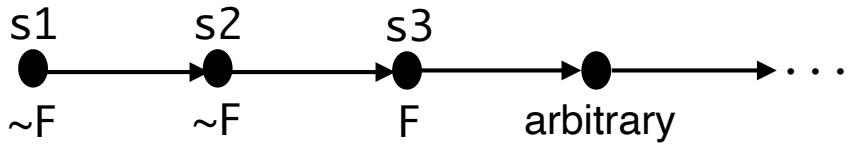
  - Formula [][A]_<<e>>, where A is an action and e a state function, is true iff every successive step is an [A]_<<e>> step



Temporal Formulas

- <>F **F is eventually true**
  - <>F == ~[](~F)

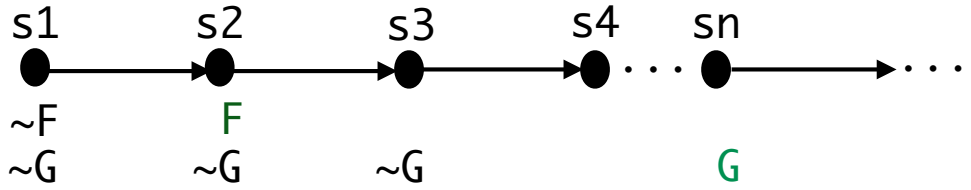
    F is not always false



Temporal Formulas

- `F ~> G` **F leads to G**
  - `F ~> G == [](F => <>G)`

    Whenever `F` is true, then `G` is eventually true



- Every request leads to a response

  `request ~> response`

Temporal Formulas

- []<>F **Infinitely often** (Progress)
  - E.g. the traffic light is green infinitely often.
- <>[]F **Eventually always** (Stability)
  - E.g. eventually all messages are delivered.

Liveness

- To prove liveness properties it is necessary to make some assumptions about the system environment.
- TLA has two forms of fairness:
  - Strong fairness
  - Weak fairness

Temporal Formulas

- Temporal operators
  - `[] F` F is always true
  - `<> F` F is eventually true
  - `F ~> G` F leads to G
  - `WF_<<e>> (A)` Weak fairness for action A
  - `SF_<<e>> (A)` Strong fairness for action A

Weak fairness

- `WF_<<e>> (A)` **Weak fairness for action A**
  - `(<>[] ENABLED <A>_<<e>>) => ([]<> <A>_<<e>>)`

    If **A** ever becomes forever enabled, then an **A** step must eventually occur

Strong fairness

- `SF_<<e>> (A)` **Strong fairness for action A**
  - `([]<> ENABLED <A>_<<e>>) => ([]<> <A>_<<e>>)`

    If **A** is infinitely often enabled, then infinitely many **A** steps occur

Fairness in TLA

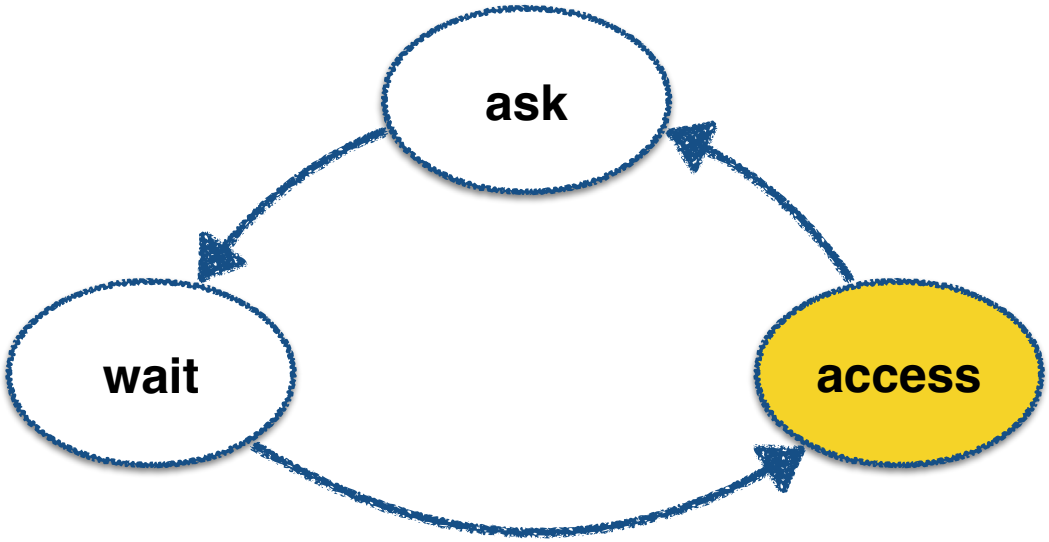
- **Weak fairness** of **A** asserts that an **A** step must eventually occur if **A** is continuously enabled
  - continuously -> without interruption
- **Strong fairness** of **A** asserts that an **A** step must eventually occur if **A** is continually enabled
  - continually -> repeatedly, possible with interruptions

Fairness in TLA

- In a **weakly fair system** you need to raise a flag saying that you want access, and you have to keep this flag up all the time, until you get access.
- In a **strongly fair system** it is enough that you raise the flag every now and then, and you know that eventually you'll get access.

```
Strong Fairness => Weak Fairness
```

Client-Server liveness properties

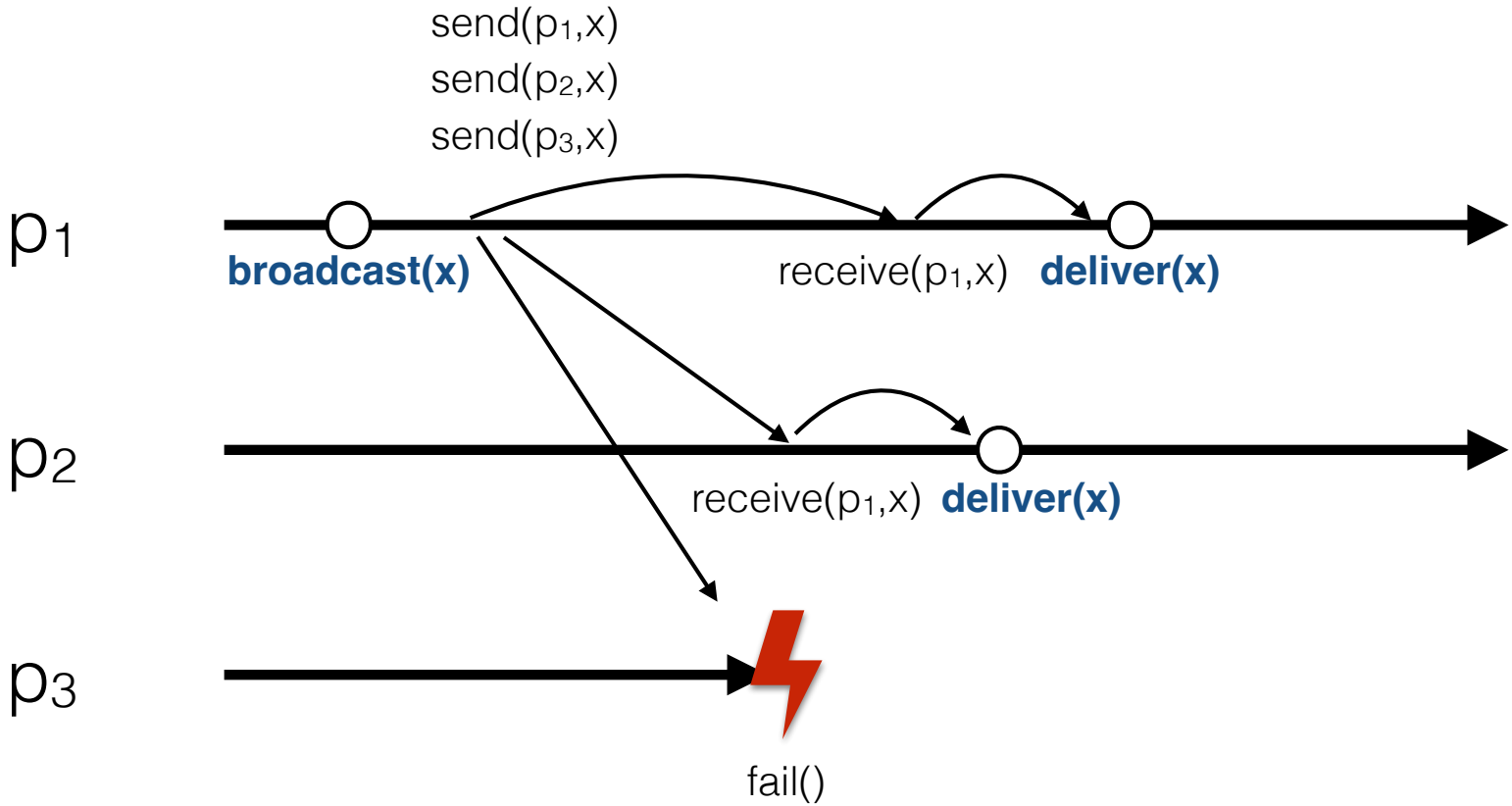



```
DeadlockFree ==
    \A c1 \in 1..N : (state[c1] = "wait" \/ state[c1] = "ask") ~>
                     (\E c2 \in 1..N : state[c2] = "access")


StarvationFree ==
    \A c \in 1..N : (state[c] = "wait" \/ state[c] = "ask") ~>
                     state[c] = "access"
```

Client-Server liveness properties

```
Client(c) == SendRequestClient(c) \/ ReceiveAnswerClient(c) \/
SendReleaseClient(c)

Next ==  \E c \in 1..N :  RcvReleaseServer \/ RcvRequestServer \/ Client(c)

Fairness ==
    /\ WF_<<vars>>(RcvReleaseServer)
    /\ WF_<<vars>>(RcvRequestServer)
    /\ \A c \in 1..N :
            /\ WF_<<vars>>(SendRequestClient(c))
            /\ WF_<<vars>>(ReceiveAnswerClient(c))
            /\ WF_<<vars>>(SendReleaseClient(c))

Spec == Init /\ [][Next]_<<vars>> /\ Fairness
```

Best effort broadcast



- For any two correct processes i and j, every message **broadcast** by i is eventually **delivered** by j.
- No message is **delivered** more than once.
- If a correct process j **delivers** a message m, then m was **broadcast** to j by some process i.

Best effort broadcast

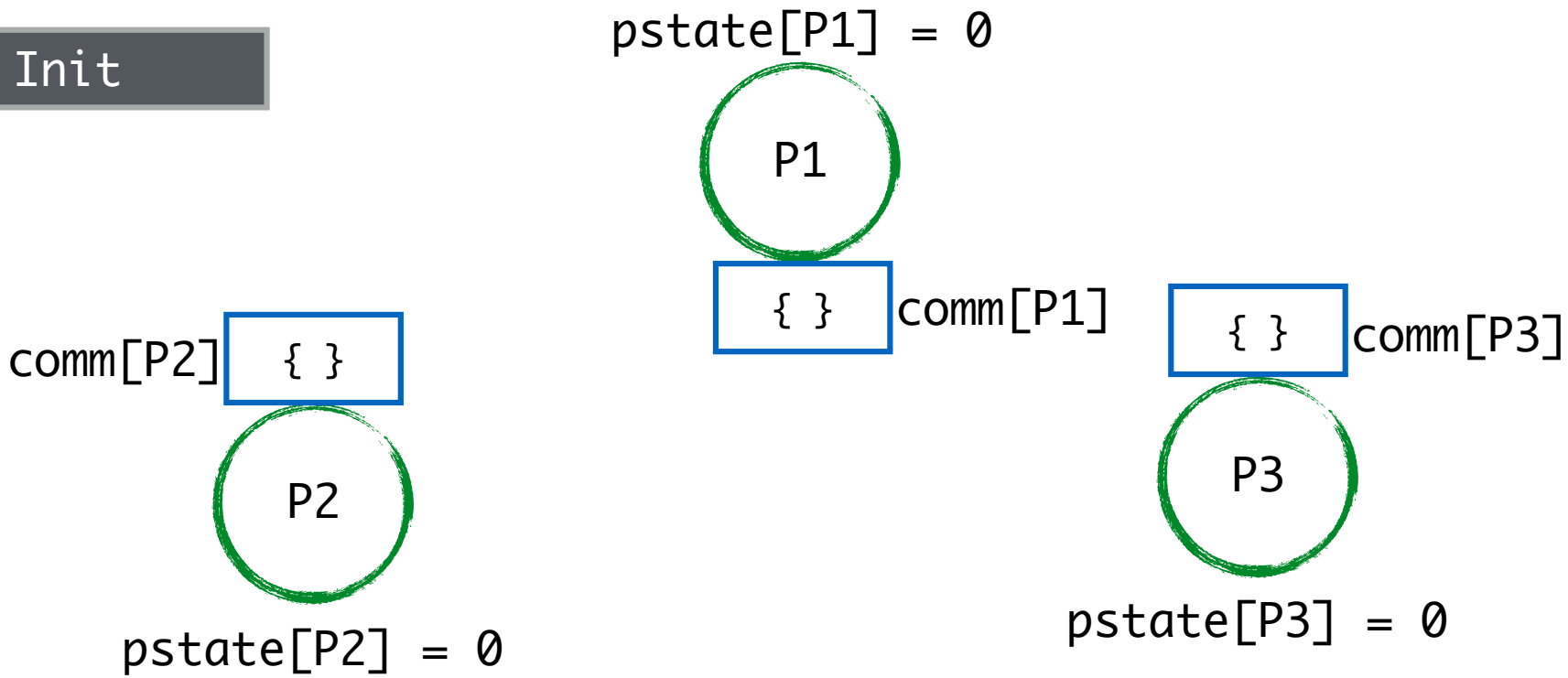
```
---------------------------- MODULE BestEffortBroadcast ----------------------------
EXTENDS Naturals, FiniteSets

CONSTANTS
    Process,        (* set of processes *)
    MaxBroadcasts   (* maximum number of broadcast messages *)

ASSUME
    /\ Process # {}
    /\ MaxBroadcasts > 0


VARIABLES
    comm,           (* point-to-point communication between processes *)
    broadcasted,    (* global set of broadcasted messages *)
    delivered,      (* global set of delivered messages *)
    pstate,         (* process local state *)
    alive,          (* set of alive/active processes *)
    correct         (* set of correct processes *)
```

Best effort broadcast

```
---------------------------- MODULE BestEffortBroadcast ----------------------------
VARIABLES
    comm,           (* point-to-point communication between processes *)
    broadcasted,    (* global set of broadcasted messages *)
    delivered,      (* global set of delivered messages *)
    pstate,         (* process local state *)
    alive,          (* set of alive/active processes *)
    correct         (* set of correct processes *)

Message == [sdr : Process, mid : 0..MaxBroadcasts] (* broadcast message *)

MessageDel == [rcv : Process, msg : Message] (* deliver message *)

TypeInv ==
    /\ comm \in [Process -> SUBSET Message]
    /\ broadcasted \in SUBSET Message
    /\ delivered \in SUBSET MessageDel
    /\ pstate \in [Process -> Nat] (* local state stores the message count *)
    /\ alive \in SUBSET Process
    /\ correct \in SUBSET Process

Init ==
    /\ comm = [p \in Process |-> {}]
    /\ broadcasted = {}
    /\ delivered = {}
    /\ pstate = [p \in Process |-> 0]
    /\ alive = Process
    /\ correct \in SUBSET Process
```

broadcasted	{ }
delivered	{ }
alive	{ P1, P2, P3 }
correct	{ P2 }

Init

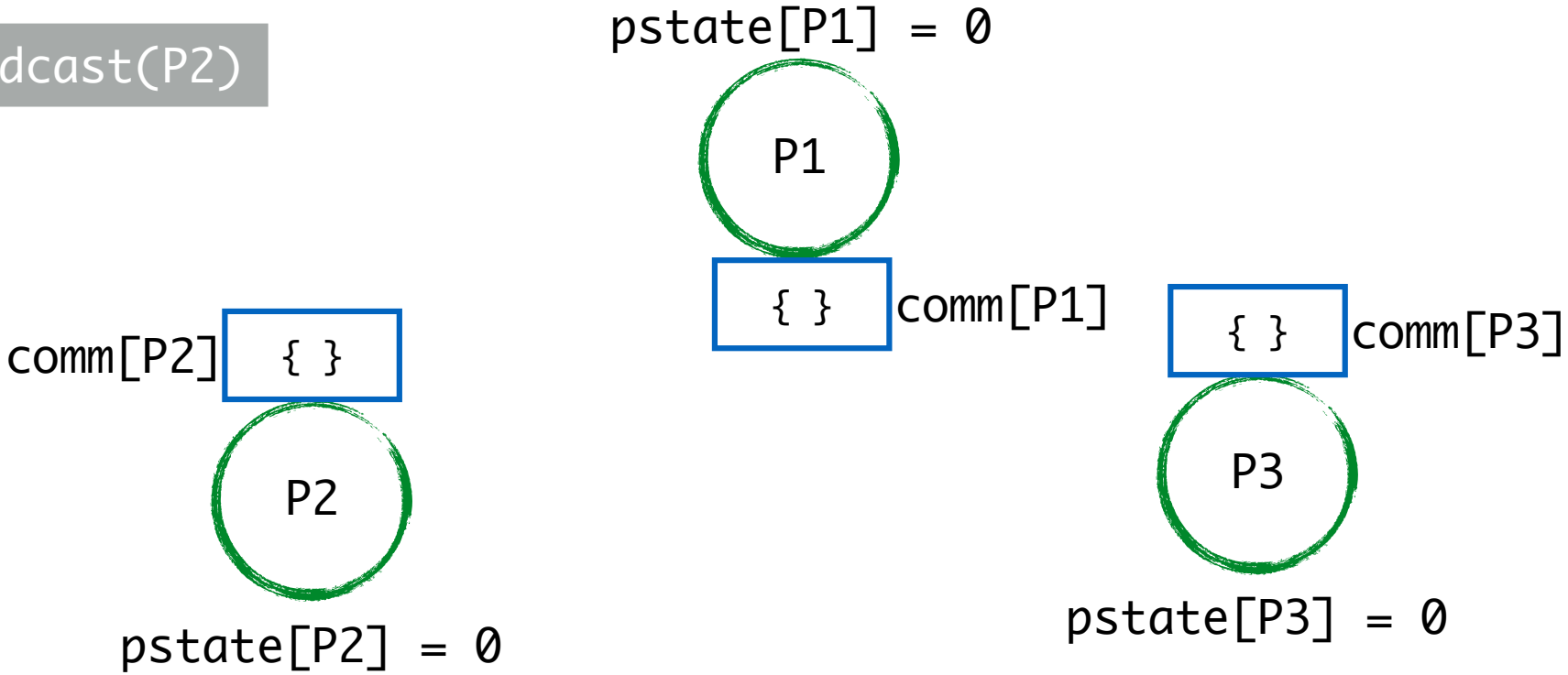


Best effort broadcast

```
---------------------------- MODULE BestEffortBroadcast ----------------------------
TypeInv ==
    /\ comm \in [Process -> SUBSET Message]   /\   broadcasted \in SUBSET Message
    /\ delivered \in SUBSET MessageDel        /\   pstate \in [Process -> Nat]
    /\ alive \in SUBSET Process               /\   correct \in SUBSET Process
------------------------------------------------------------------------------
broadcast(p) ==
    /\ p \in alive
    /\ Cardinality(broadcasted) < MaxBroadcasts
    /\ LET msg == [sdr |-> p, mid |-> pstate[p]]
       IN
        /\ broadcasted' = broadcasted \union { msg }
        /\ comm'= [q \in Process |-> comm[q] \union { msg }]
        /\ pstate' = [pstate EXCEPT ![p] = pstate[p] + 1]
    /\ UNCHANGED<<delivered,alive,correct>>
```

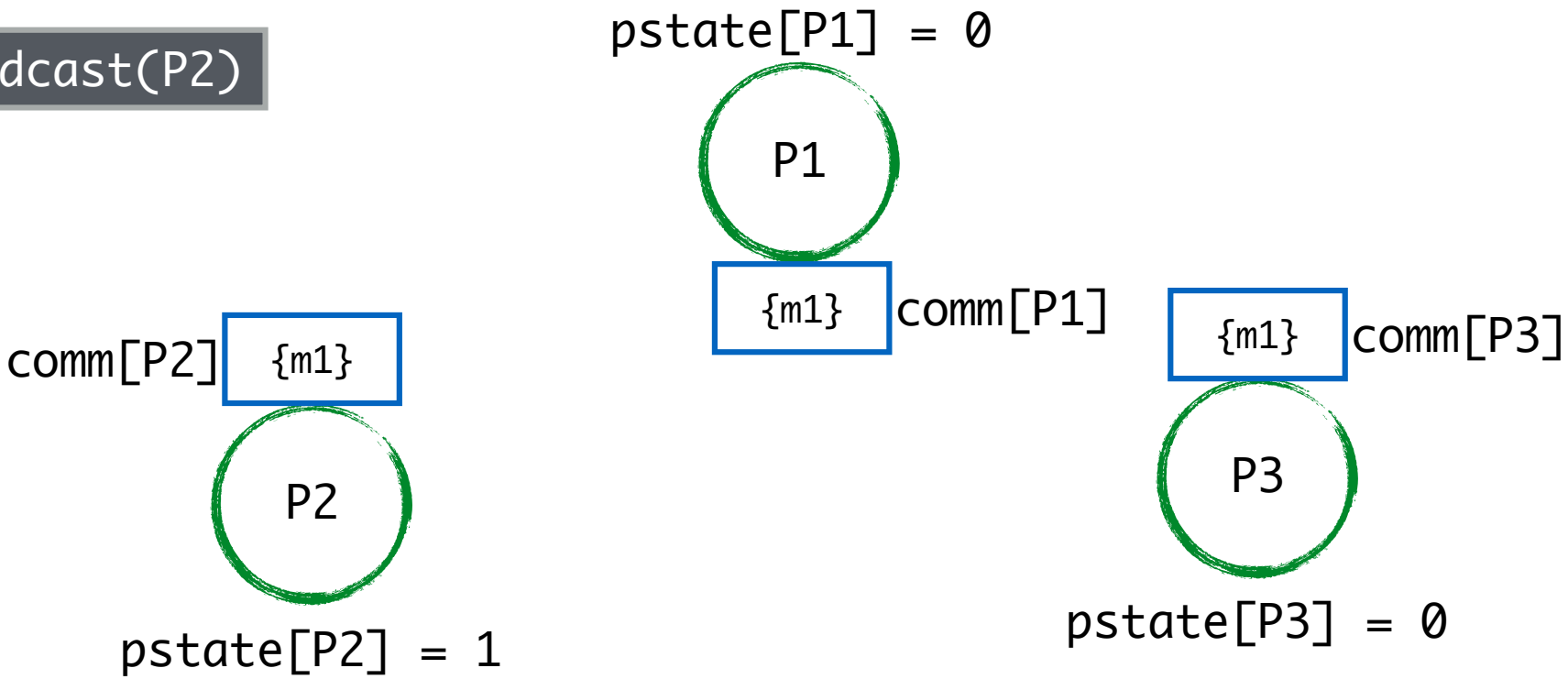
broadcasted	{ }
delivered	{ }
alive	{ P1, P2, P3 }
correct	{ P2 }

broadcast(P2)



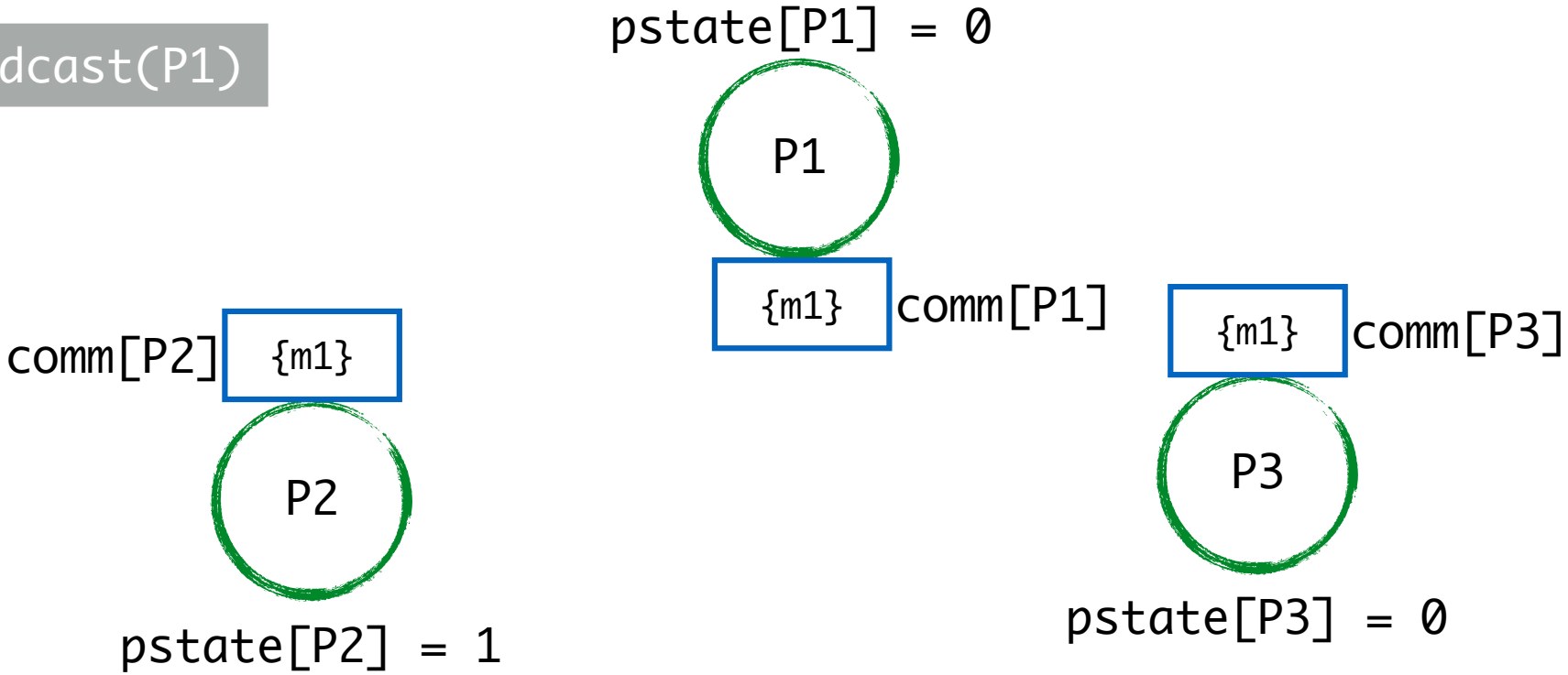
broadcasted	{ m1 }
delivered	{ }
alive	{ P1, P2, P3 }
correct	{ P2 }

broadcast(P2)



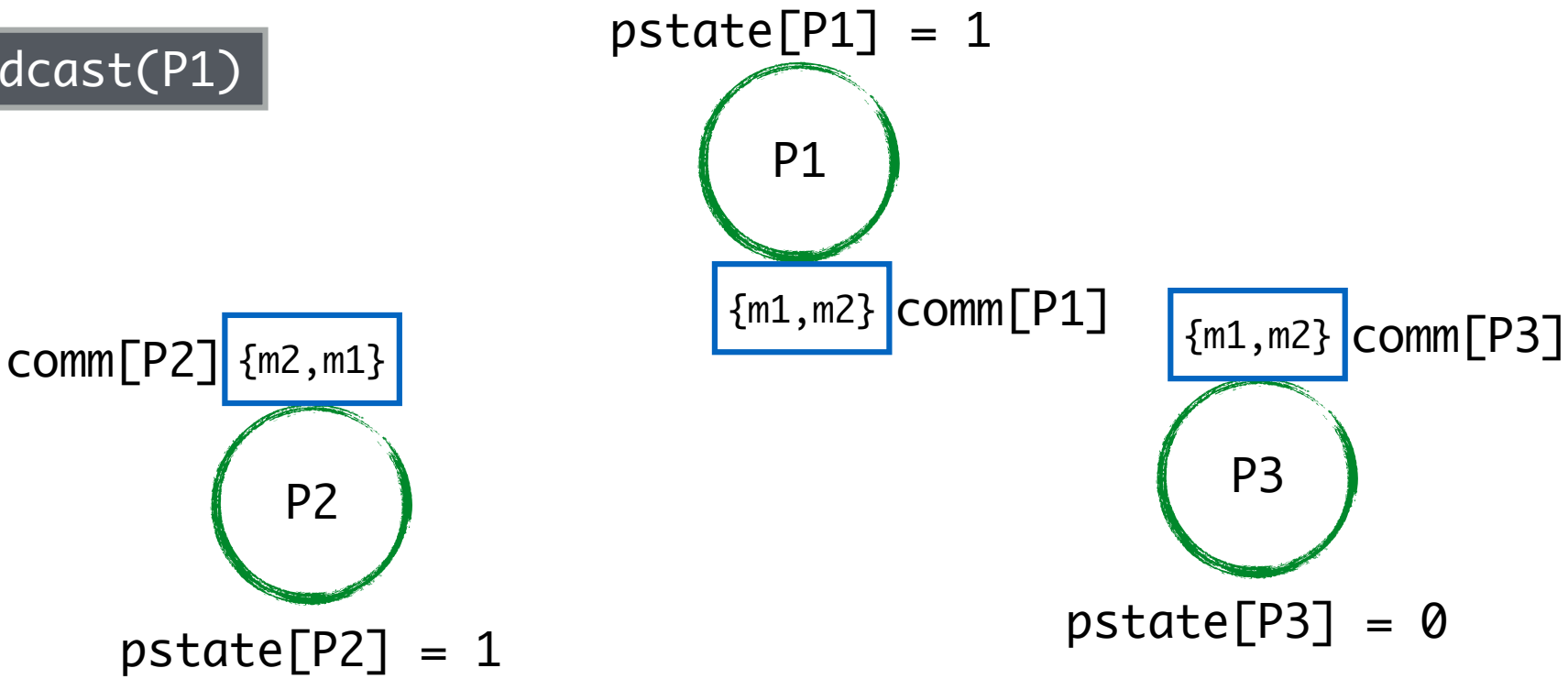
broadcasted	{ m1 }
delivered	{ }
alive	{ P1, P2, P3 }
correct	{ P2 }

broadcast(P1)



broadcasted	{ m1, m2 }
delivered	{ }
alive	{ P1, P2, P3 }
correct	{ P2 }

broadcast(P1)

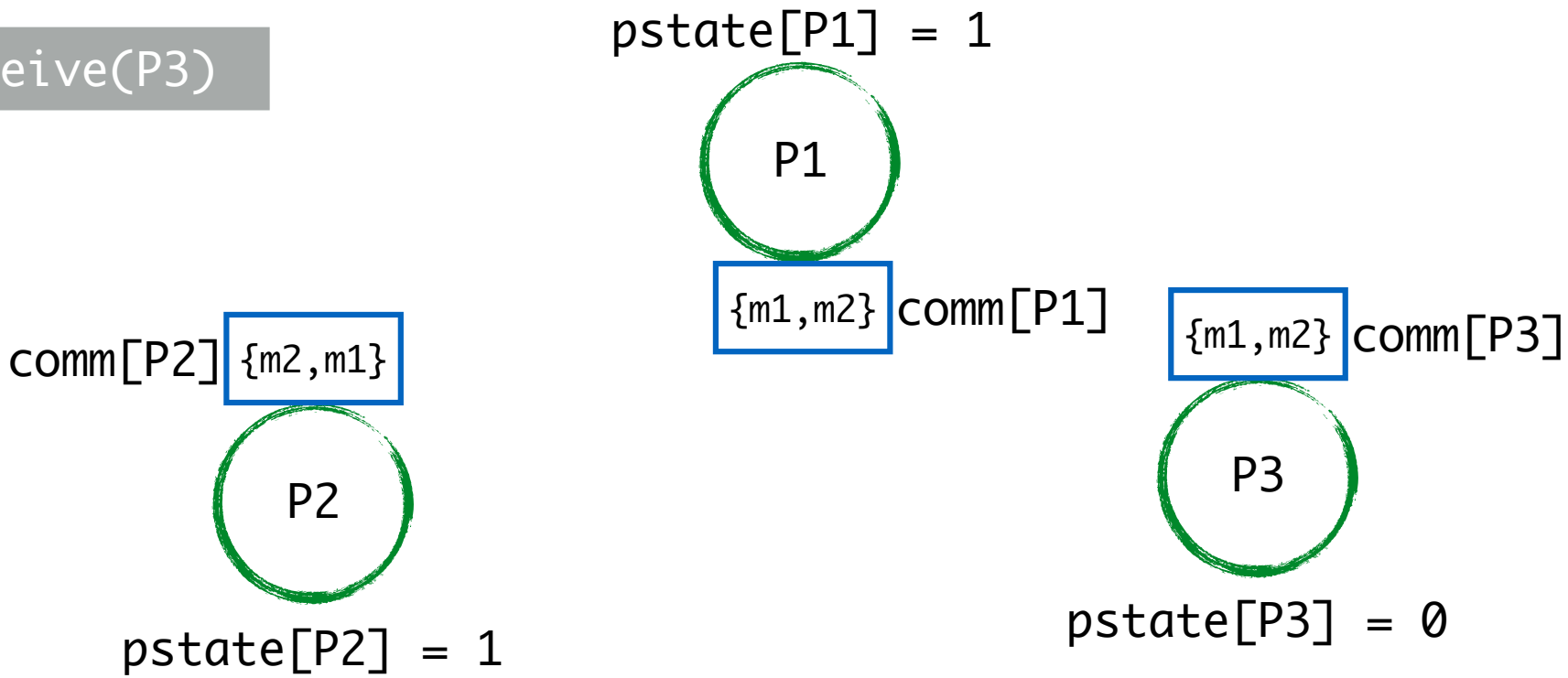


Best effort broadcast

```
---------------------------- MODULE BestEffortBroadcast ----------------------------
TypeInv ==
    /\ comm \in [Process -> SUBSET Message]   /\   broadcasted \in SUBSET Message
    /\ delivered \in SUBSET MessageDel        /\   pstate \in [Process -> Nat]
    /\ alive \in SUBSET Process.              /\   correct \in SUBSET Process
------------------------------------------------------------------------------------
receive(p) ==
    /\ p \in alive
    /\ \E m \in comm[p] :
        /\ comm' = [comm EXCEPT ![p] = comm[p] \ {m}]
        /\ delivered' = delivered \union {[rcv |-> p, msg |-> m]}
        /\ UNCHANGED<<broadcasted,alive,pstate,correct>>
```

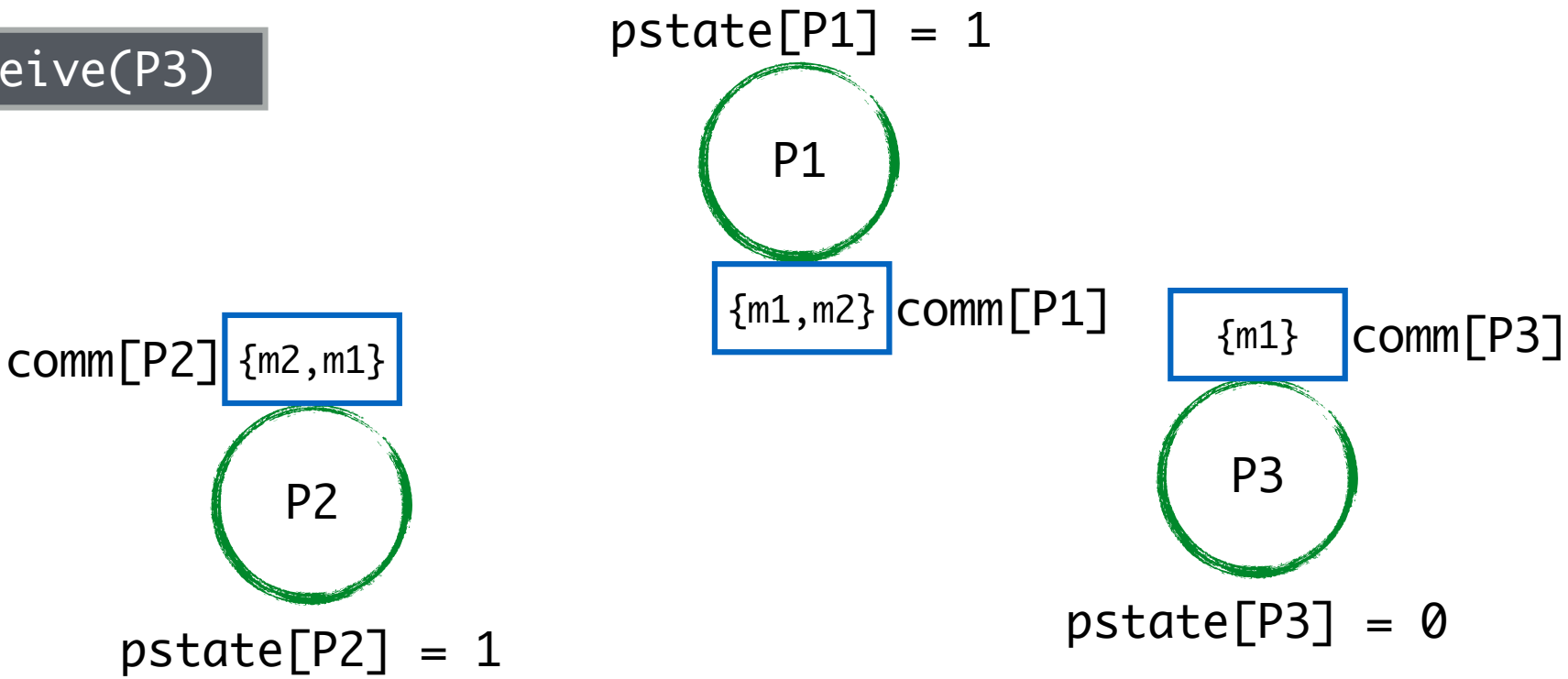
broadcasted	{ m1, m2 }
delivered	{ }
alive	{ P1, P2, P3 }
correct	{ P2 }

receive(P3)



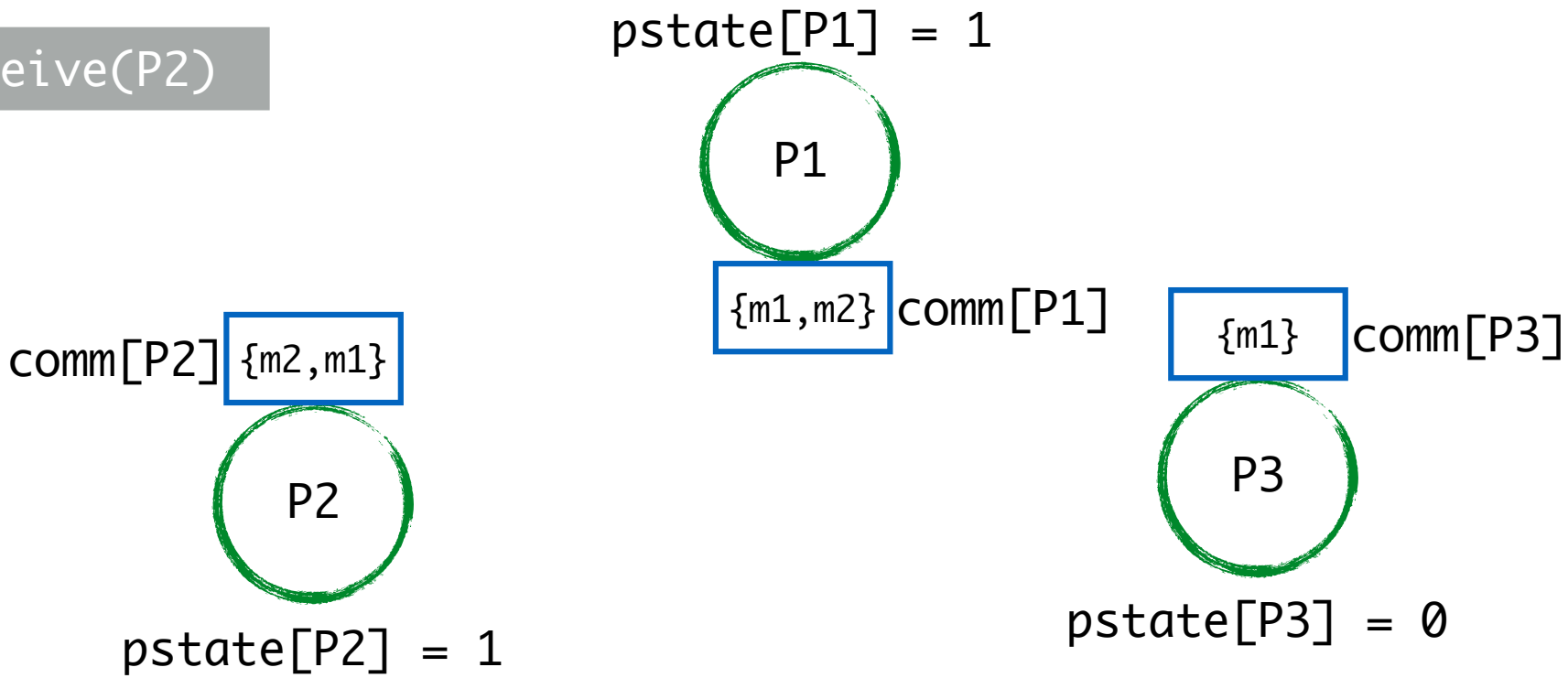
broadcasted	{ m1, m2 }
delivered	{ P3_m2 }
alive	{ P1, P2, P3 }
correct	{ P2 }

receive(P3)



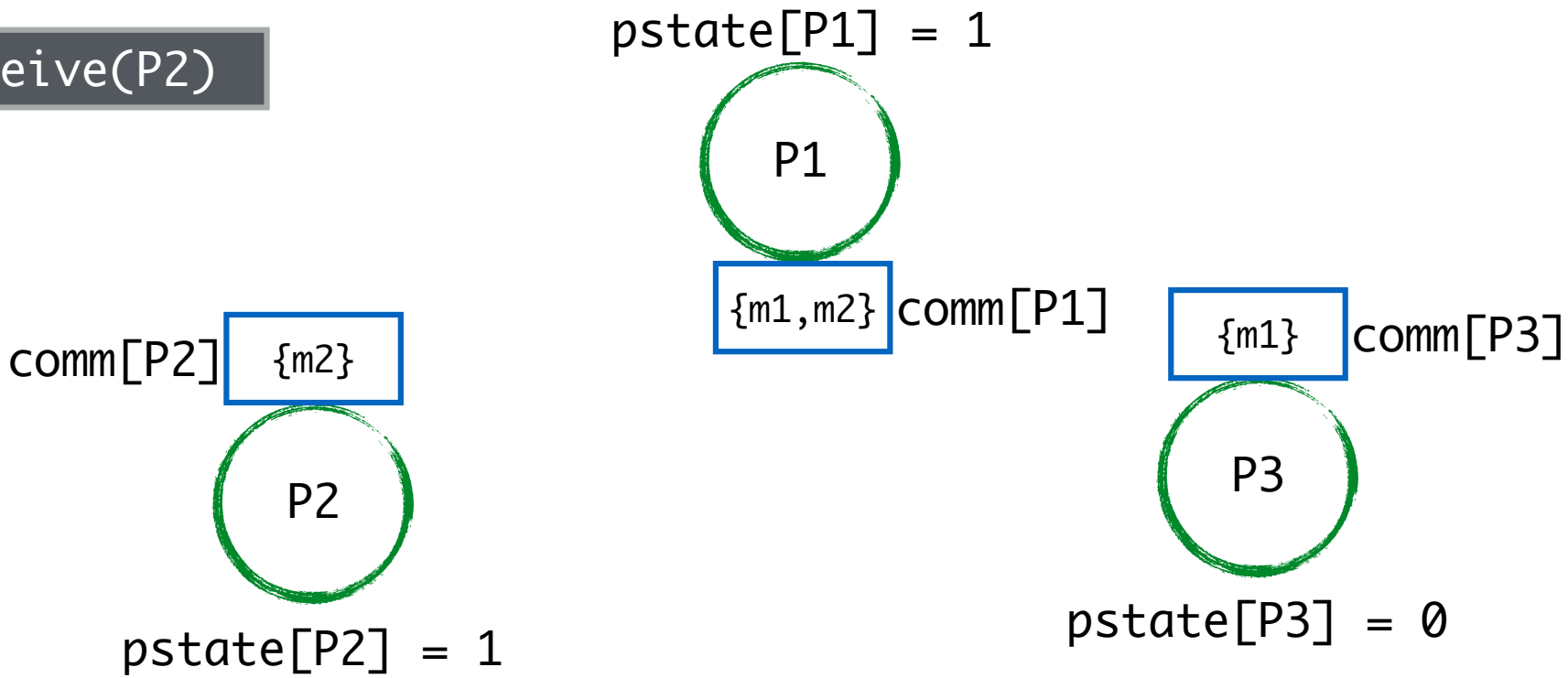
broadcasted	{ m1, m2 }
delivered	{ P3_m2 }
alive	{ P1, P2, P3 }
correct	{ P2 }

receive(P2)



broadcasted	{ m1, m2 }
delivered	{ P2_m1, P3_m2 }
alive	{ P1, P2, P3 }
correct	{ P2 }

receive(P2)

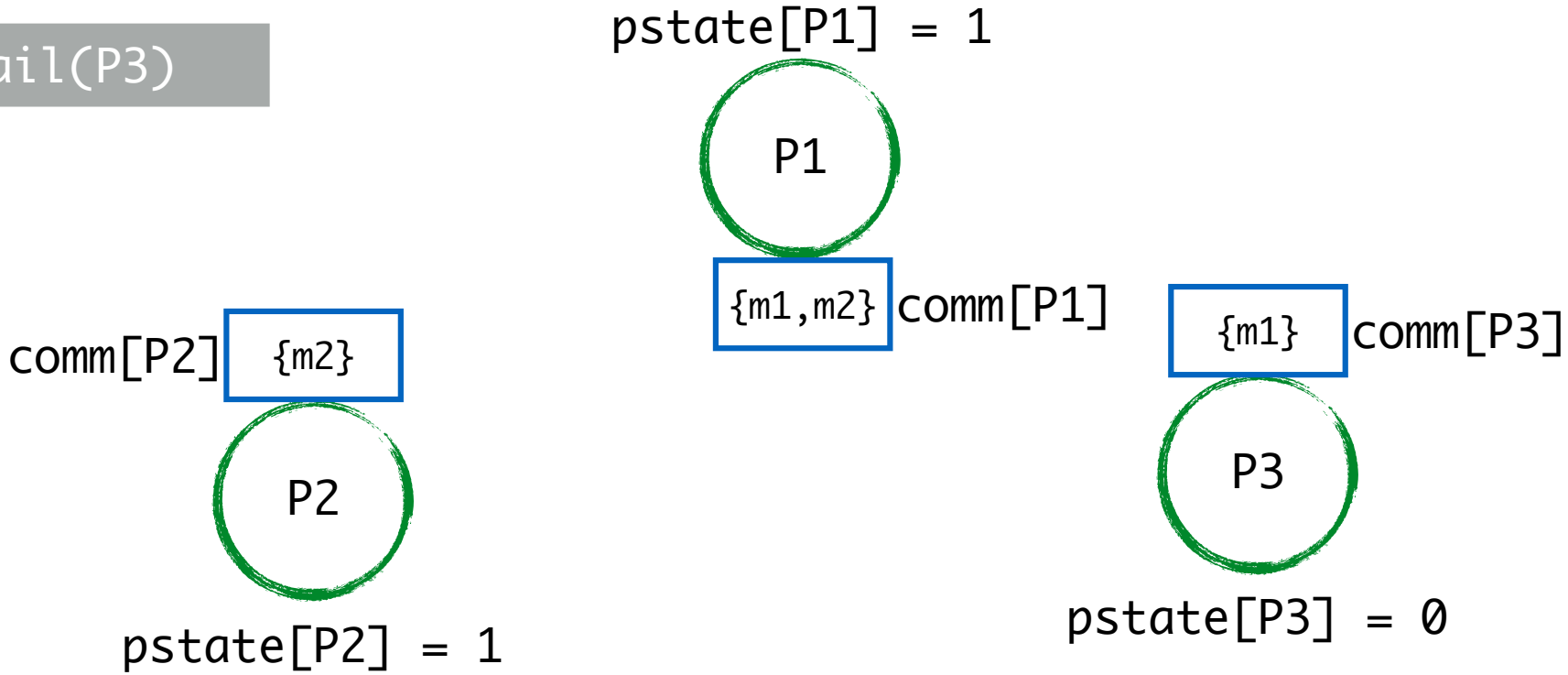


Best effort broadcast

```
---------------------------- MODULE BestEffortBroadcast ----------------------------
TypeInv ==
    /\ comm \in [Process -> SUBSET Message]   /\   broadcasted \in SUBSET Message
    /\ delivered \in SUBSET MessageDel        /\   pstate \in [Process -> Nat]
    /\ alive \in SUBSET Process.              /\   correct \in SUBSET Process
------------------------------------------------------------------------------
fail(p) ==
    /\ p \in alive
    /\ p \notin correct
    /\ alive' = alive \ {p}
    /\ UNCHANGED<<comm,broadcasted,delivered,pstate,correct>>
```

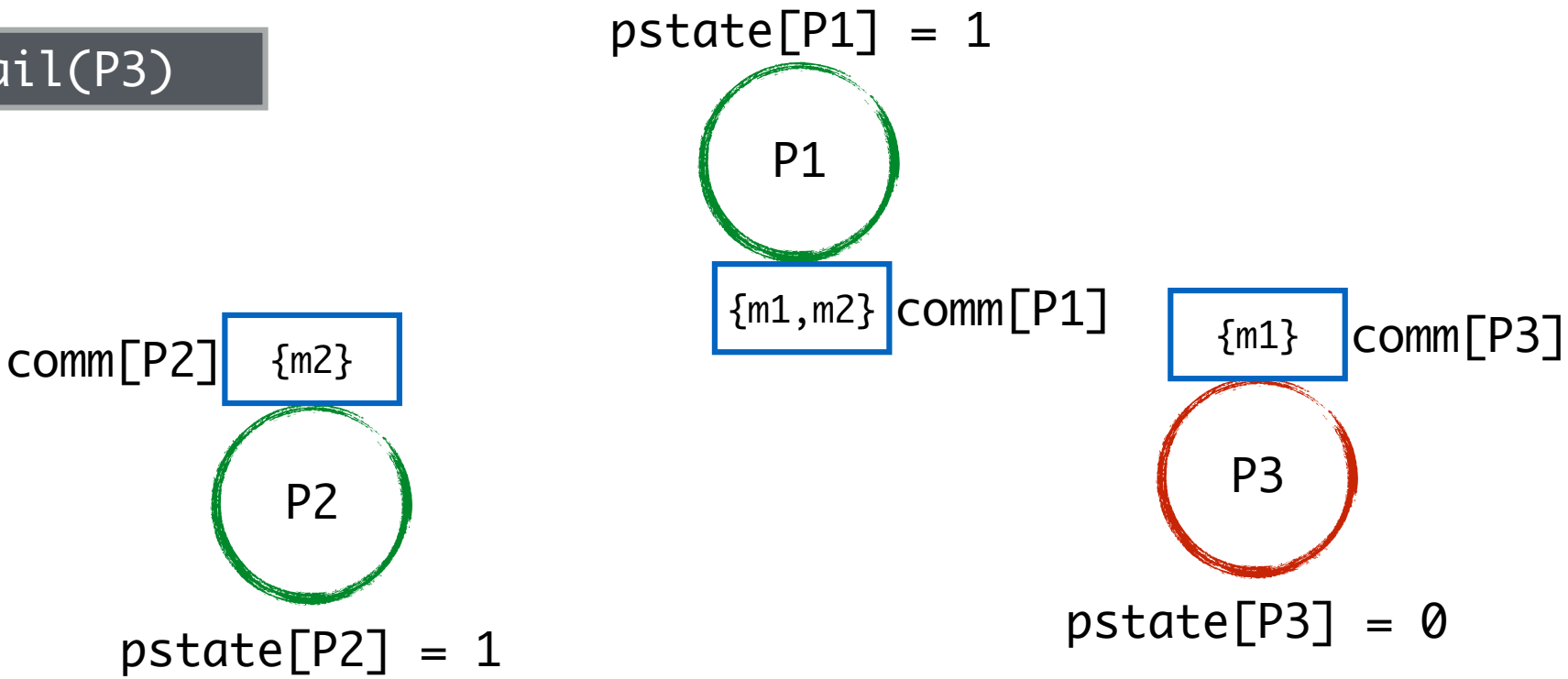
broadcasted	{ m1, m2 }
delivered	{ P2_m1, P3_m2 }
alive	{ P1, P2, P3 }
correct	{ P2 }

fail(P3)



broadcasted	{ m1, m2 }
delivered	{ P2_m1, P3_m2 }
alive	{ P1, P2 }
correct	{ P2 }

fail(P3)



Best effort broadcast

```
---------------------------- MODULE BestEffortBroadcast ----------------------------

Next == \E p \in Process : broadcast(p) \/ receive(p) \/ fail(p)

------------------------------------------------------------------------------

Spec == /\ Init
        /\ [][Next]_<<comm, broadcasted, delivered, pstate, alive, correct>>
==============================================================================
```

To model check with TLC this specification we need to define some models:

No. of processes	No. of broadcasted messages	No. of distinct states	Time
2	1	81	14s
3	2	11043	15s
3	5	20387619	5m 47s

Best effort broadcast

```
broadcast(p) ==
    /\ p \in alive
    /\ Cardinality(broadcasted) < MaxBroadcasts
    /\ LET msg == [sdr |-> p, mid |-> pstate[p]]
       IN
        /\ broadcasted' = broadcasted \union { msg }
        /\ comm'= [q \in Process |-> comm[q] \union { msg }]
        /\ pstate' = [pstate EXCEPT ![p] = pstate[p] + 1]
    /\ UNCHANGED<<delivered,alive,correct>>

receive(p) ==
    /\ p \in alive
    /\ \E m \in comm[p] :
        /\ comm' = [comm EXCEPT ![p] = comm[p] \ {m}]
        /\ delivered' = delivered \union {[rcv |-> p, msg |-> m]}
        /\ UNCHANGED<<broadcasted,alive,pstate,correct>>

fail(p) ==
    /\ p \in alive
    /\ p \notin correct
    /\ alive' = alive \ {p}
    /\ UNCHANGED<<comm,broadcasted,delivered,pstate,correct>>
```

Is this a “good” specification of the protocol?

Best effort broadcast (2nd try)

```
---------------------------- MODULE BestEffortBroadcast ----------------------------
EXTENDS Naturals, FiniteSets

CONSTANTS
    Process,        (* set of processes *)
    MaxBroadcasts   (* maximum number of broadcast messages *)

ASSUME
    /\ Process # {}
    /\ MaxBroadcasts > 0


VARIABLES
    comm,           (* point-to-point communication between processes *)
    broadcasted,    (* global set of broadcasted messages *)
    delivered,      (* global set of delivered messages *)
    pstate,         (* process local state *)
    alive,          (* set of alive/active processes *)
    correct         (* set of correct processes *)
```

Exactly the same as before!

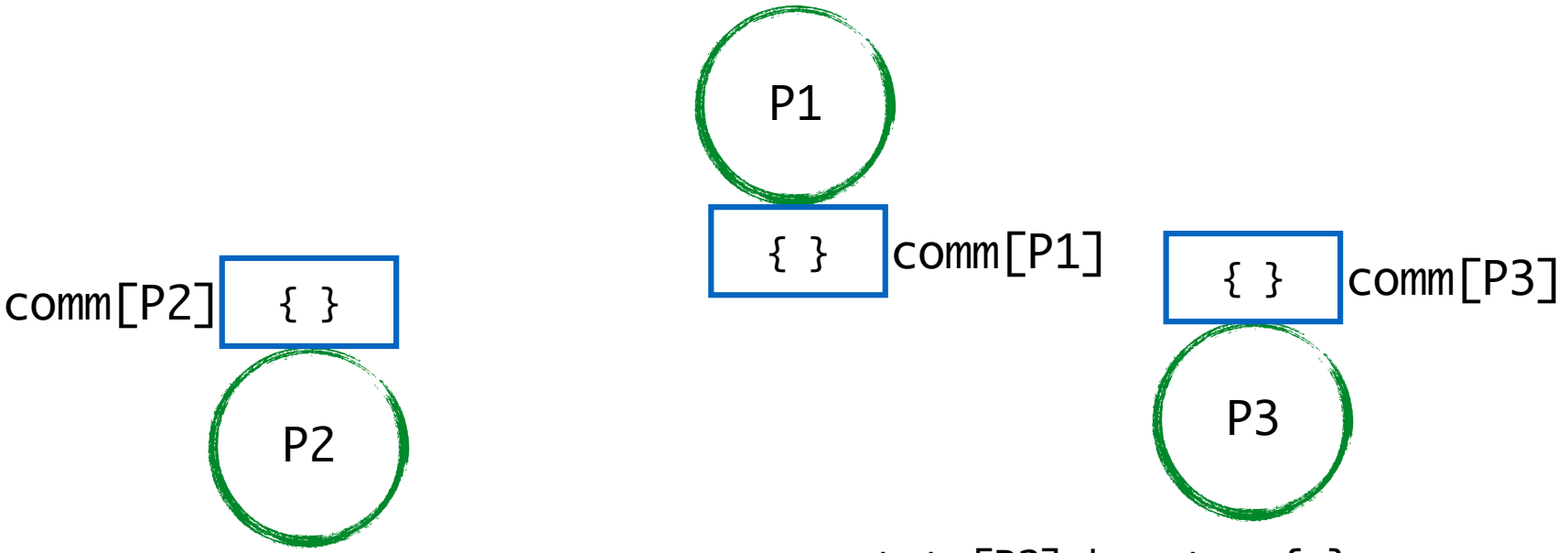
Best effort broadcast (2nd try)

```
---------------------------- MODULE BestEffortBroadcast ----------------------------
Message == [sdr : Process, mid : 0..MaxBroadcasts]  (* broadcast message *)

MessageDel == [rcv : Process, msg : Message]        (* deliver message *)

                                                    (* process local state *)
LocalState == [bcast: SUBSET Message,               (* messages broadcasted *)
               msent: [Process -> SUBSET Message],  (* messages sent *)
               mcount : Nat]                        (* broadcast messages count *)

VARIABLES
    comm, broadcasted, delivered, pstate, alive, correct

TypeInv ==
    /\ comm \in [Process -> SUBSET Message]
    /\ broadcasted \in SUBSET Message
    /\ delivered \in SUBSET MessageDel
    /\ pstate \in [Process -> LocalState]
    /\ alive \in SUBSET Process
    /\ correct \in SUBSET Process

Init ==
    /\ comm = [p \in Process |-> {}]
    /\ broadcasted = {}
    /\ delivered = {}
    /\ pstate = [p \in Process |-> [bcast |-> {},
                                    msent |-> [q \in Process |-> {}],
                                    mcount |-> 0]]
    /\ alive = Process
    /\ correct \in SUBSET Process
```

broadcasted	{ }
delivered	{ }
alive	{ P1, P2, P3 }
correct	{ P2 }

Init

```
pstate[P1].bcast = { }
pstate[P1].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P1].mcount = 0
```




```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

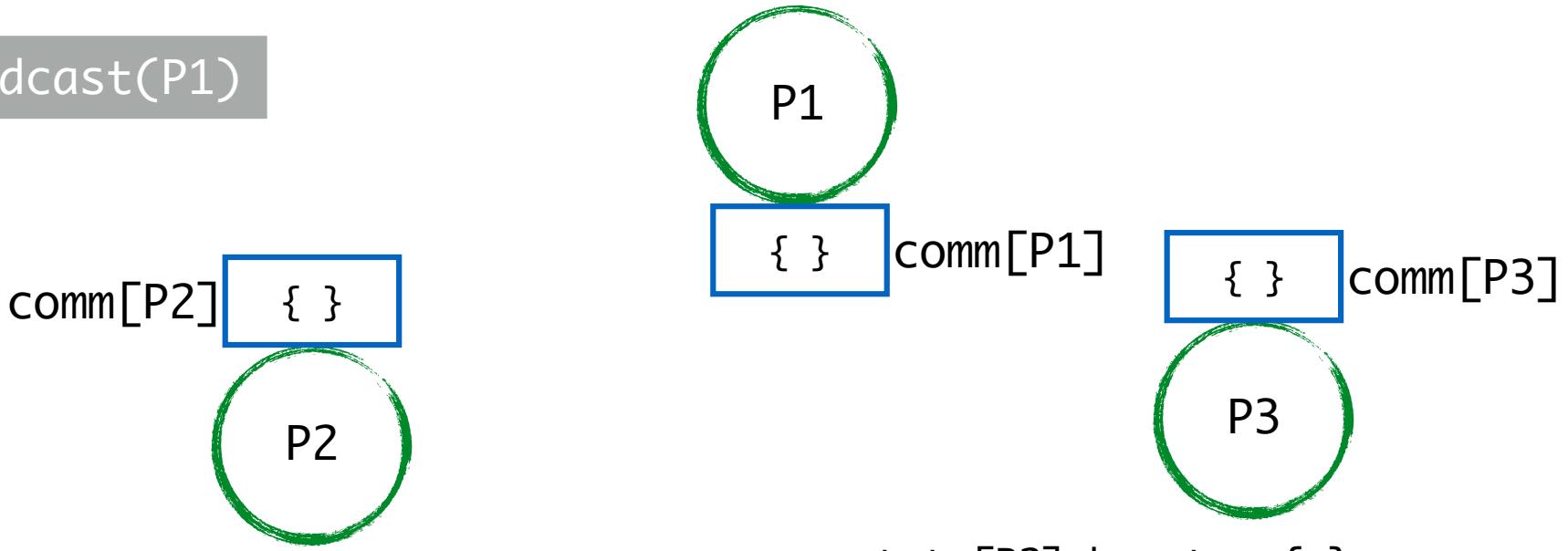
Best effort broadcast (2nd try)

```
---------------------------- MODULE BestEffortBroadcast ----------------------------
TypeInv ==
    /\ comm \in [Process -> SUBSET Message]   /\   broadcasted \in SUBSET Message
    /\ delivered \in SUBSET MessageDel        /\   pstate \in [Process -> LocalState]
    /\ alive \in SUBSET Process.              /\   correct \in SUBSET Process
------------------------------------------------------------------------------------
broadcast(p) ==
    /\ p \in alive
    /\ Cardinality(pstate[p].bcast) < MaxBroadcasts
    /\ LET msg == [sdr |-> p, mid |-> pstate[p].mcount]
       IN
        /\ broadcasted' = broadcasted \union { msg }
        /\ pstate' = [pstate EXCEPT ![p] = [bcast |-> pstate[p].bcast \union {msg},
                                            msent |-> pstate[p].msent,
                                            mcount |-> pstate[p].mcount + 1]]
    /\ UNCHANGED<<comm,delivered,alive,correct>>
```

broadcasted	{ }
delivered	{ }
alive	{ P1, P2, P3 }
correct	{ P2 }

```
pstate[P1].bcast = { }
pstate[P1].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P1].mcount = 0
```

broadcast(P1)



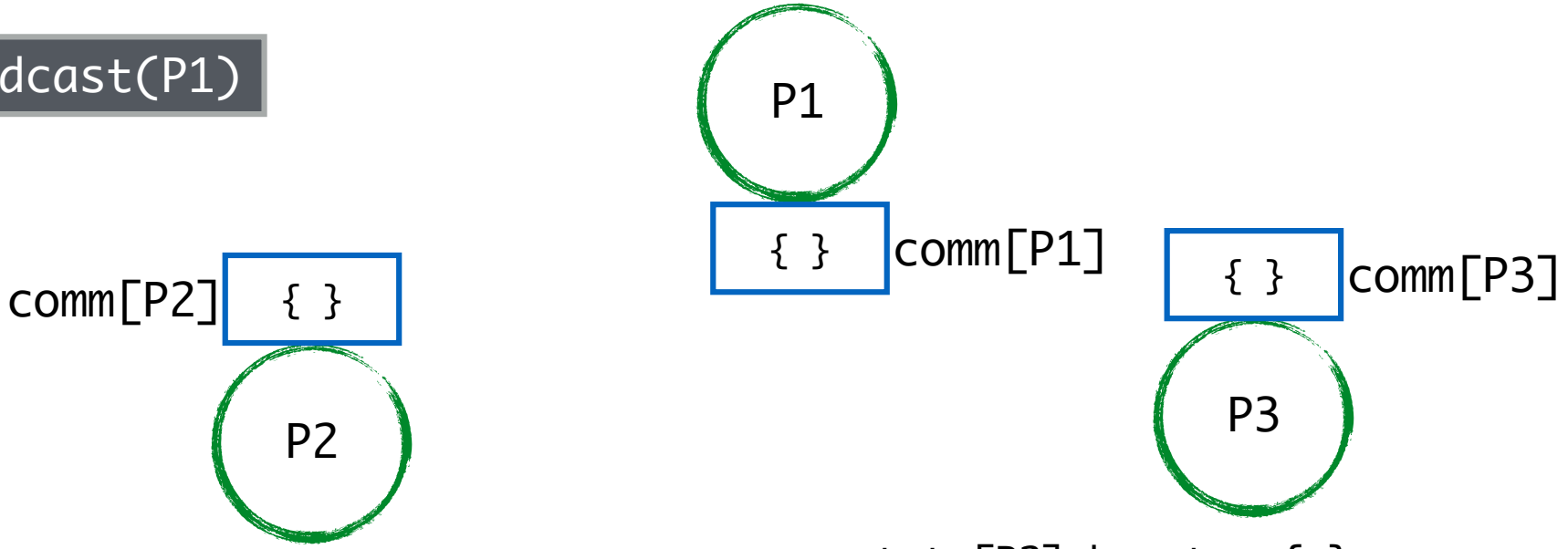

```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

broadcasted	{ m1 }
delivered	{ }
alive	{ P1, P2, P3 }
correct	{ P2 }

```
pstate[P1].bcast = { m1 }
pstate[P1].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P1].mcount = 1
```

broadcast(P1)




```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

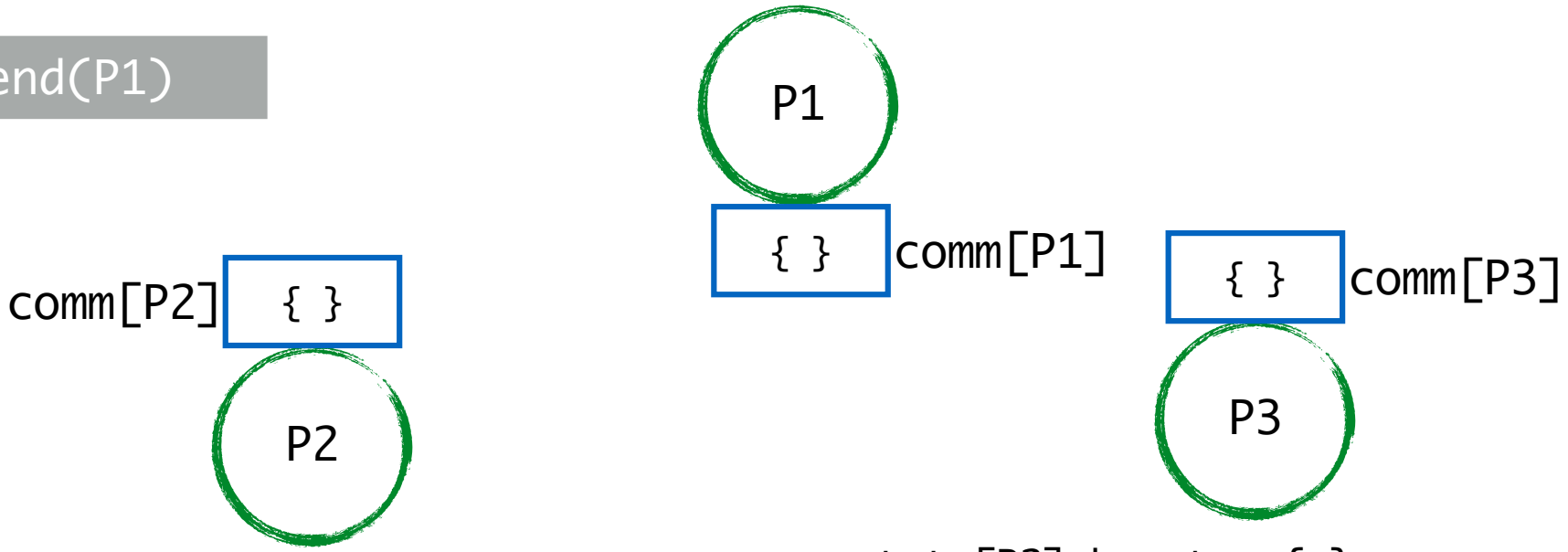
Best effort broadcast (2nd try)

```
---------------------------- MODULE BestEffortBroadcast ----------------------------
TypeInv ==
    /\ comm \in [Process -> SUBSET Message]   /\   broadcasted \in SUBSET Message
    /\ delivered \in SUBSET MessageDel        /\   pstate \in [Process -> LocalState]
    /\ alive \in SUBSET Process.              /\   correct \in SUBSET Process
------------------------------------------------------------------------------------
send(p) ==
    /\ p \in alive
    /\ \E m \in pstate[p].bcast, q \in Process :
        /\ m \notin pstate[p].msent[q] (* m was not sent to q *)
        /\ comm' = [comm EXCEPT ![q] = comm[q] \union {m}]
        /\ pstate' = [pstate EXCEPT ![p] =
                            [bcast |-> pstate[p].bcast,
                             msent |-> [pstate[p].msent
                                        EXCEPT ![q] = pstate[p].msent[q] \union {m}],
                             mcount |-> pstate[p].mcount]]
        /\ UNCHANGED<<broadcasted,delivered,alive,correct>>
```

broadcasted	{ m1 }
delivered	{ }
alive	{ P1, P2, P3 }
correct	{ P2 }

```
pstate[P1].bcast = { m1 }
pstate[P1].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P1].mcount = 1
```

send(P1)



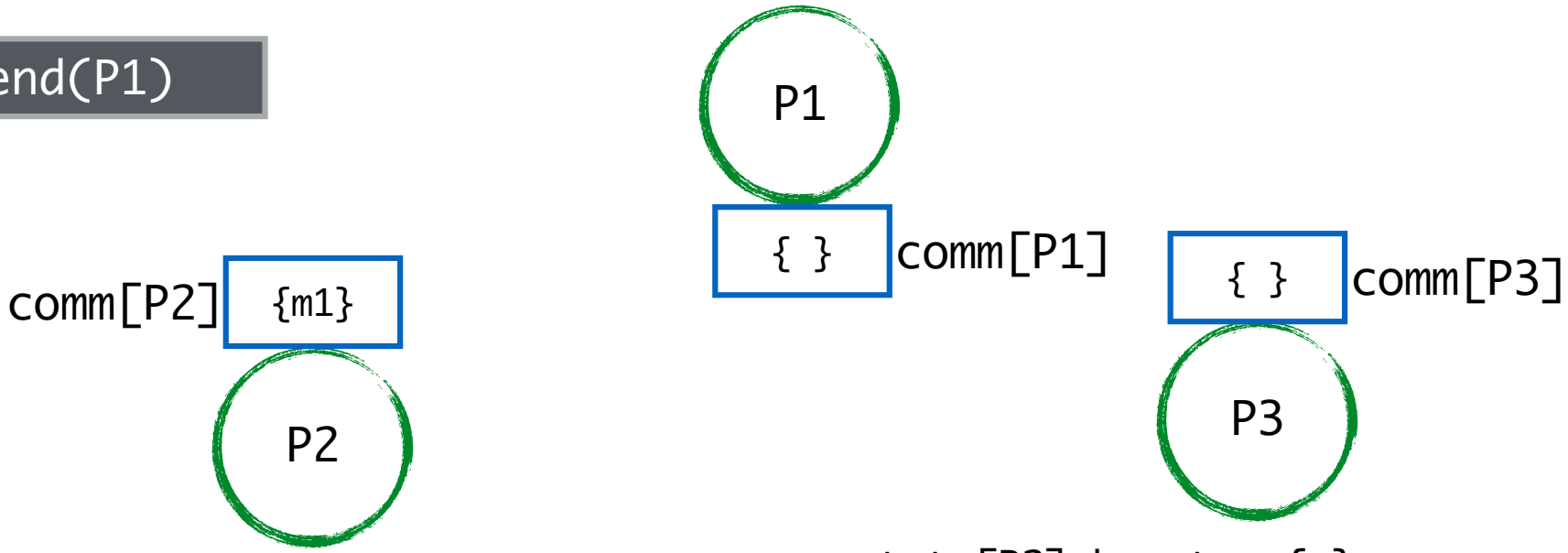

```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

broadcasted	{ m1 }
delivered	{ }
alive	{ P1, P2, P3 }
correct	{ P2 }

```
pstate[P1].bcast = { m1 }
pstate[P1].msent = {P1↦{},P2↦{m1},P3↦{}}
pstate[P1].mcount = 1
```

send(P1)



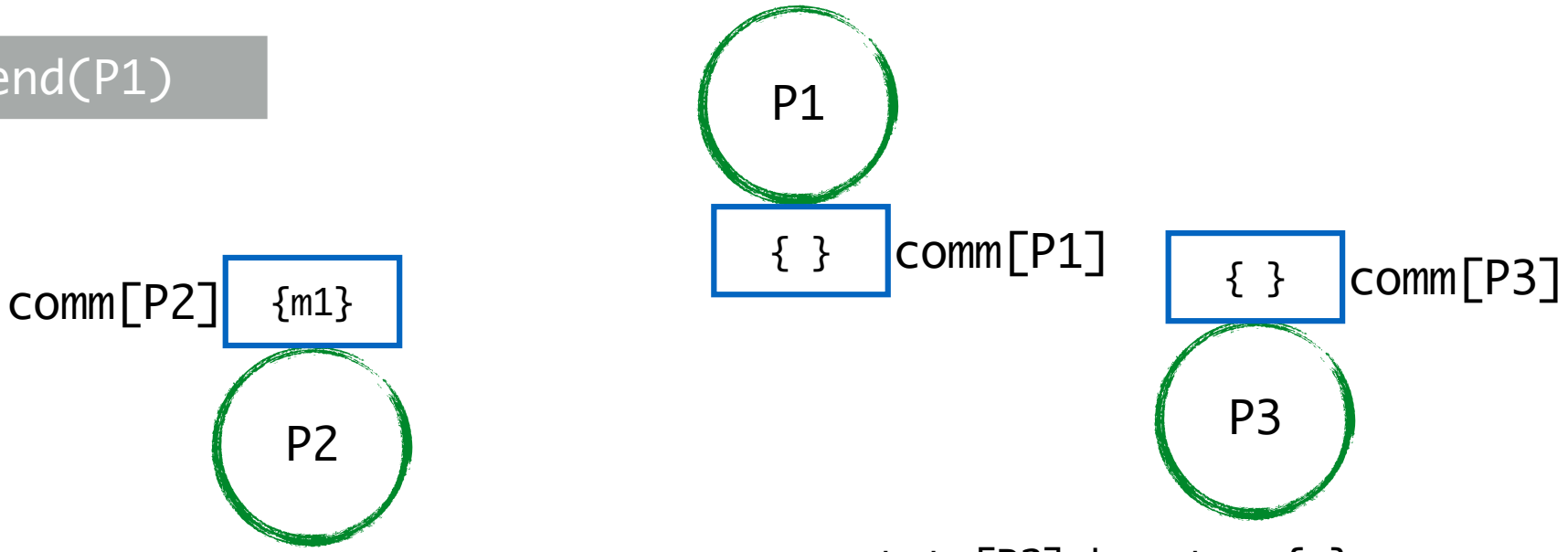

```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

broadcasted	{ m1 }
delivered	{ }
alive	{ P1, P2, P3 }
correct	{ P2 }

```
pstate[P1].bcast = { m1 }
pstate[P1].msent = {P1↦{},P2↦{m1},P3↦{}}
pstate[P1].mcount = 1
```

send(P1)



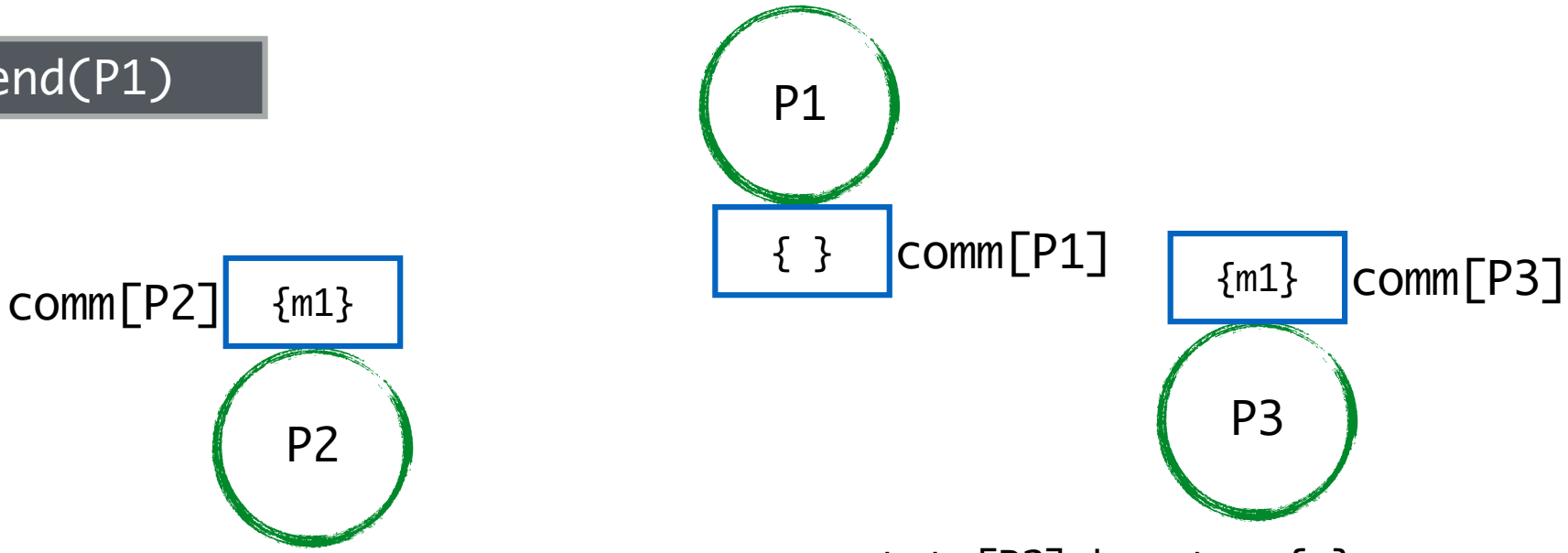

```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

broadcasted	{ m1 }
delivered	{ }
alive	{ P1, P2, P3 }
correct	{ P2 }

```
pstate[P1].bcast = { m1 }
pstate[P1].msent = {P1↦{},P2↦{m1},P3↦{m1}}
pstate[P1].mcount = 1
```

send(P1)




```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

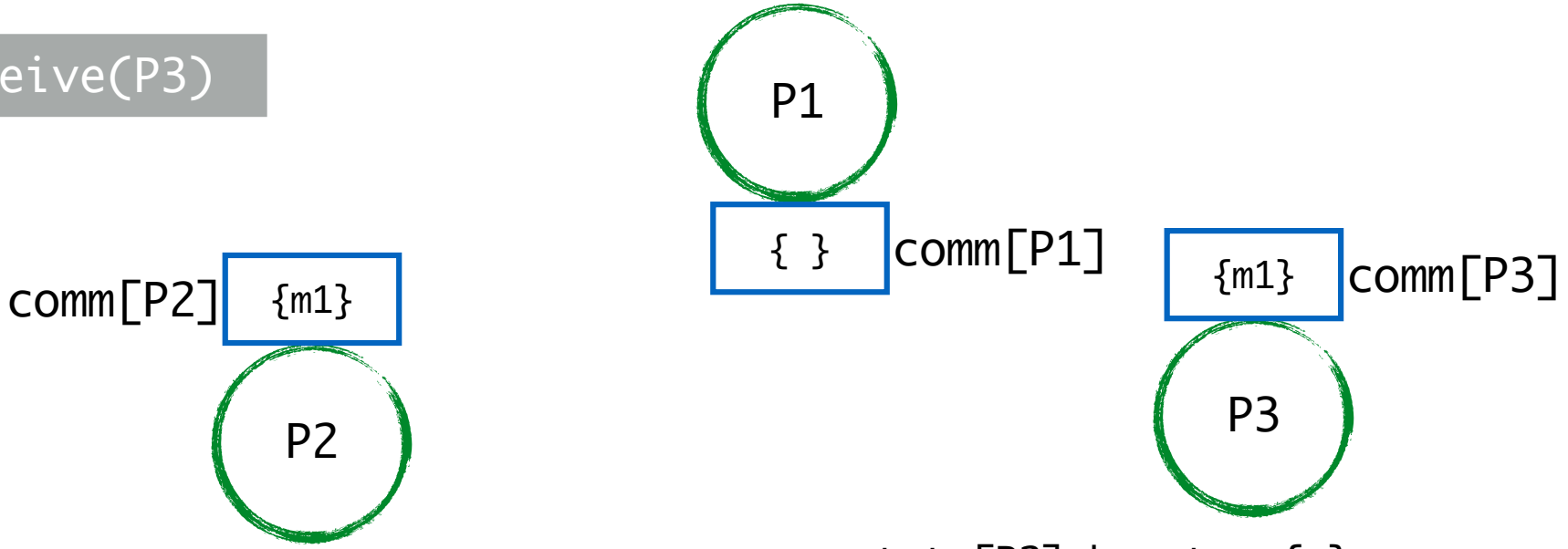
Best effort broadcast (2nd try)

```
---------------------------- MODULE BestEffortBroadcast ----------------------------
TypeInv ==
    /\ comm \in [Process -> SUBSET Message]   /\   broadcasted \in SUBSET Message
    /\ delivered \in SUBSET MessageDel        /\   pstate \in [Process -> LocalState]
    /\ alive \in SUBSET Process.              /\   correct \in SUBSET Process
------------------------------------------------------------------------------------
receive(p) ==
    /\ p \in alive
    /\ \E m \in comm[p] :
        /\ comm' = [comm EXCEPT ![p] = comm[p] \ {m}]
        /\ delivered' = delivered \union {[rcv |-> p, msg |-> m]}
        /\ UNCHANGED<<broadcasted,alive,pstate,correct>>
```

broadcasted	{ m1 }
delivered	{ }
alive	{ P1, P2, P3 }
correct	{ P2 }

```
pstate[P1].bcast = { m1 }
pstate[P1].msent = {P1↦{},P2↦{m1},P3↦{m3}}
pstate[P1].mcount = 1
```

receive(P3)



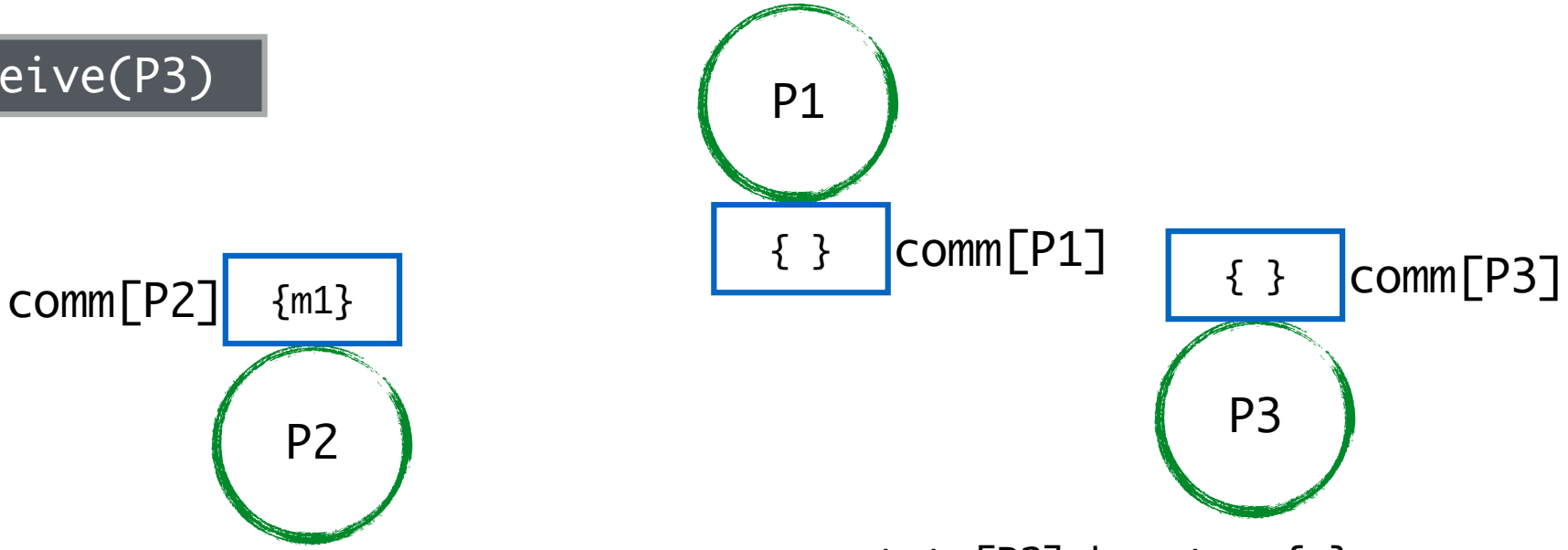

```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

broadcasted	{ m1 }
delivered	{ P3_m1 }
alive	{ P1, P2, P3 }
correct	{ P2 }

```
pstate[P1].bcast = { m1 }
pstate[P1].msent = {P1↦{},P2↦{m1},P3↦{m3}}
pstate[P1].mcount = 1
```

receive(P3)



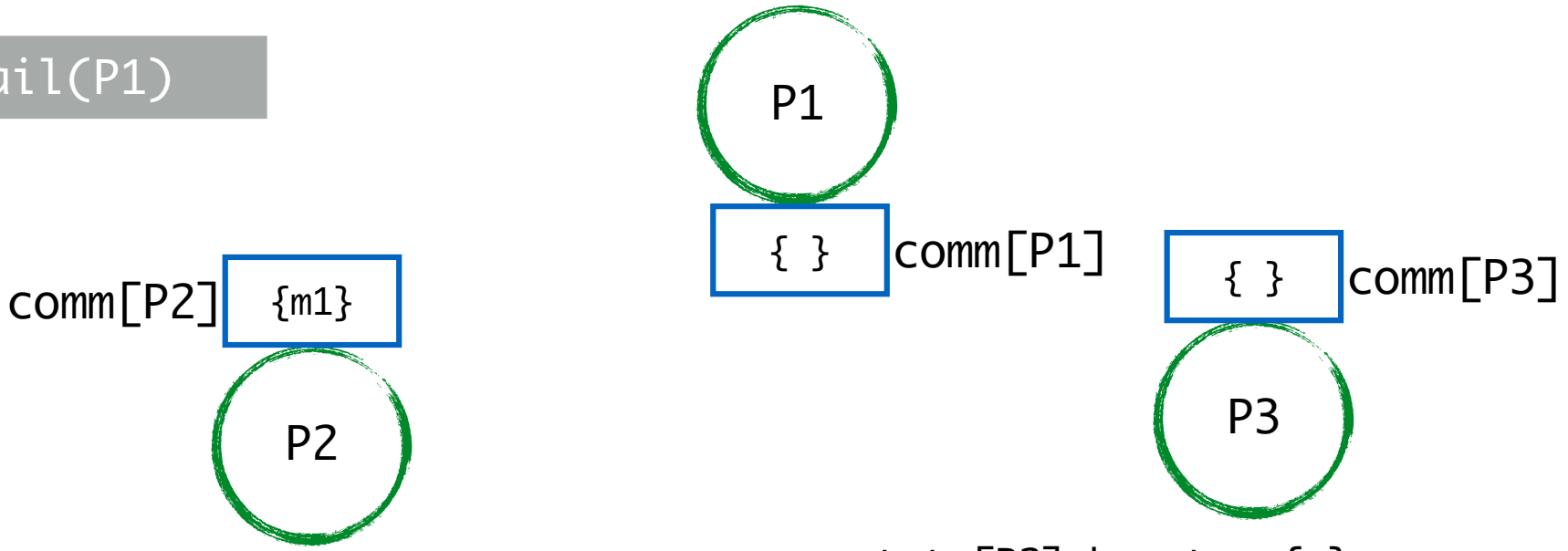

```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

broadcasted	{ m1 }
delivered	{ P3_m1 }
alive	{ P1, P2, P3 }
correct	{ P2 }

```
pstate[P1].bcast = { m1 }
pstate[P1].msent = {P1↦{},P2↦{m1},P3↦{m3}}
pstate[P1].mcount = 1
```

fail(P1)



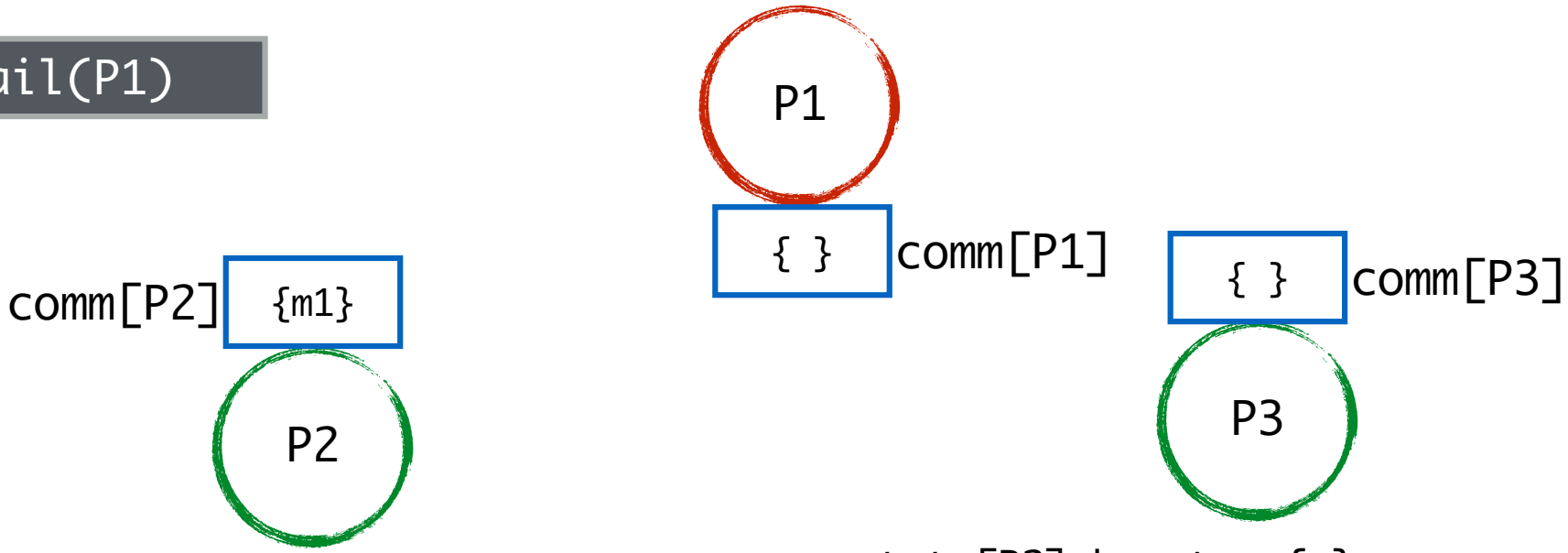

```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

broadcasted	{ m1 }
delivered	{ P3_m1 }
alive	{ P2, P3 }
correct	{ P2 }

```
pstate[P1].bcast = { m1 }
pstate[P1].msent = {P1↦{},P2↦{m1},P3↦{m3}}
pstate[P1].mcount = 1
```

```
fail(P1)
```




```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

```
pstate[P2].bcast = { }
pstate[P2].msent = {P1↦{},P2↦{},P3↦{}}
pstate[P2].mcount = 0
```

Best effort broadcast (2nd try)

```
---------------------------- MODULE BestEffortBroadcast ----------------------------

Next == \E p \in Process : broadcast(p) \/ sent(p) \/ receive(p) \/ fail(p)

------------------------------------------------------------------------------

Spec == /\ Init
        /\ [][Next]_<<comm, broadcasted, delivered, pstate, alive, correct>>
==============================================================================
```

Just checking type safety!
What about specific protocol properties?

Liveness properties

- Validity
  - For any two correct processes i and j, every message broadcast by i is eventually delivered by j.

- Agreement
  - If a message m is delivered by some correct process i, then m is eventually delivered by every correct process j.

Express this properties in TLA.

Fairness conditions

```
---------------------------- MODULE BestEffortBroadcast ----------------------------

Next == \E p \in Process : broadcast(p) \/ sent(p) \/ receive(p) \/ fail(p)

------------------------------------------------------------------------------

vars == <<comm, broadcasted, delivered, pstate, alive, correct>>

Fairness == \A p \in Process :
    /\ SF_<<vars>>(broadcast(p)) /\ SF_<<vars>>(send(p))
    /\ SF_<<vars>>(receive(p)) /\ SF_<<vars>>(fail(p))

Spec == /\ Init
        /\ [][Next]_<<vars>>
        /\ Fairness
==============================================================================
```