RDMA & Concurrent Algorithms

4 Nov 2024

Igor Zablotchi

MystenLabs

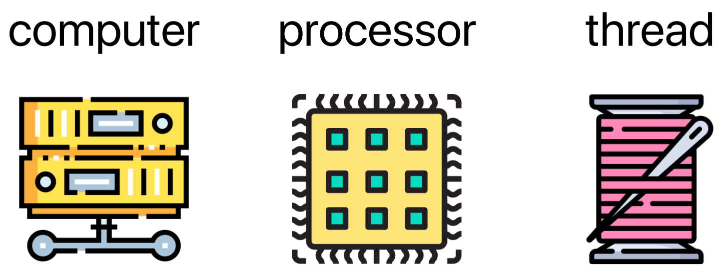
Based on joint work with, and slides from:

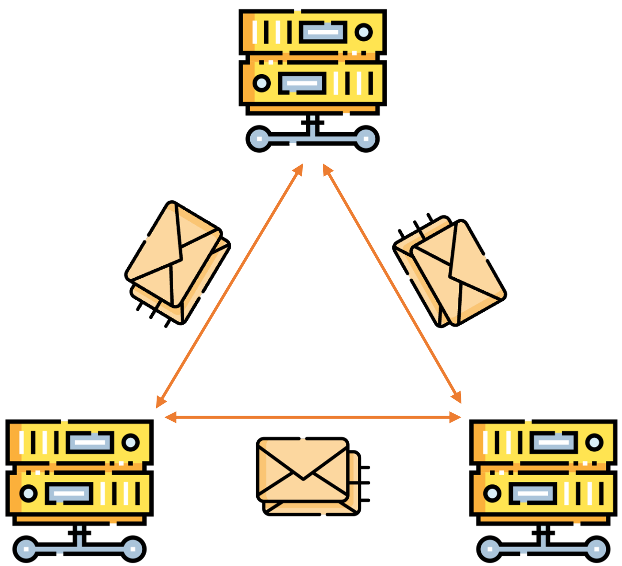
Marcos Aguilera, Naama Ben-David, Clément Burgelin, Rachid Guerraoui, Virendra Marathe, Antoine Murat, Dalia Papuc, Athanasios Xygkis

EPFL vmware Microsoft Research Carnegie Mellon University Oracle Labs

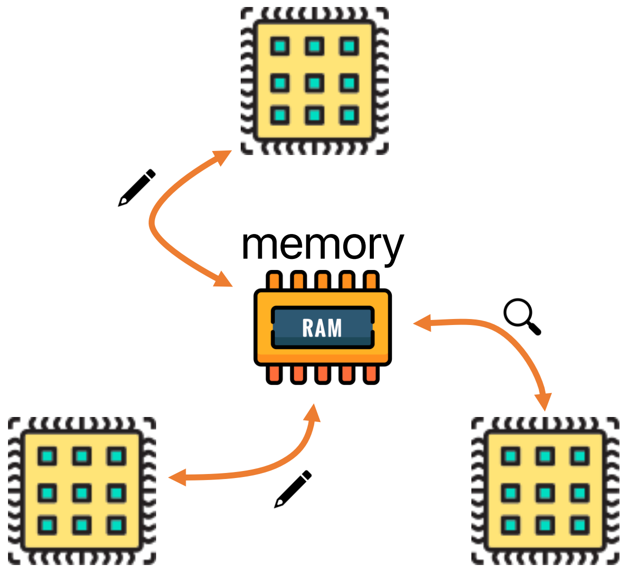
A Tale of Two Models

- processes
- collaborate on some common task
- improve performance or robustness





message-passing



shared-memory

Equal But Not Quite

The two models are equivalent [Attiya, Bar-Noy, Dolev 1995]
=
One can simulate the other

but, e.g., for solving consensus:

n = num processes f = num failures	Crash	
	Fault Tolerance	Common-case Complexity
Message Passing	$f < n/2$	**2** [Lamport'98]
Shared Memory	$\boldsymbol{f < n}$	4 [GL'02]

Models Reflect Technology

The two standard models reflect existing technology

BUT

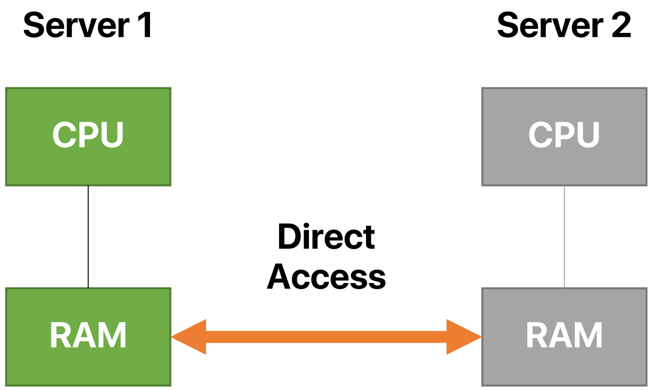
Technology evolves, new technologies emerge

SO

We need new models

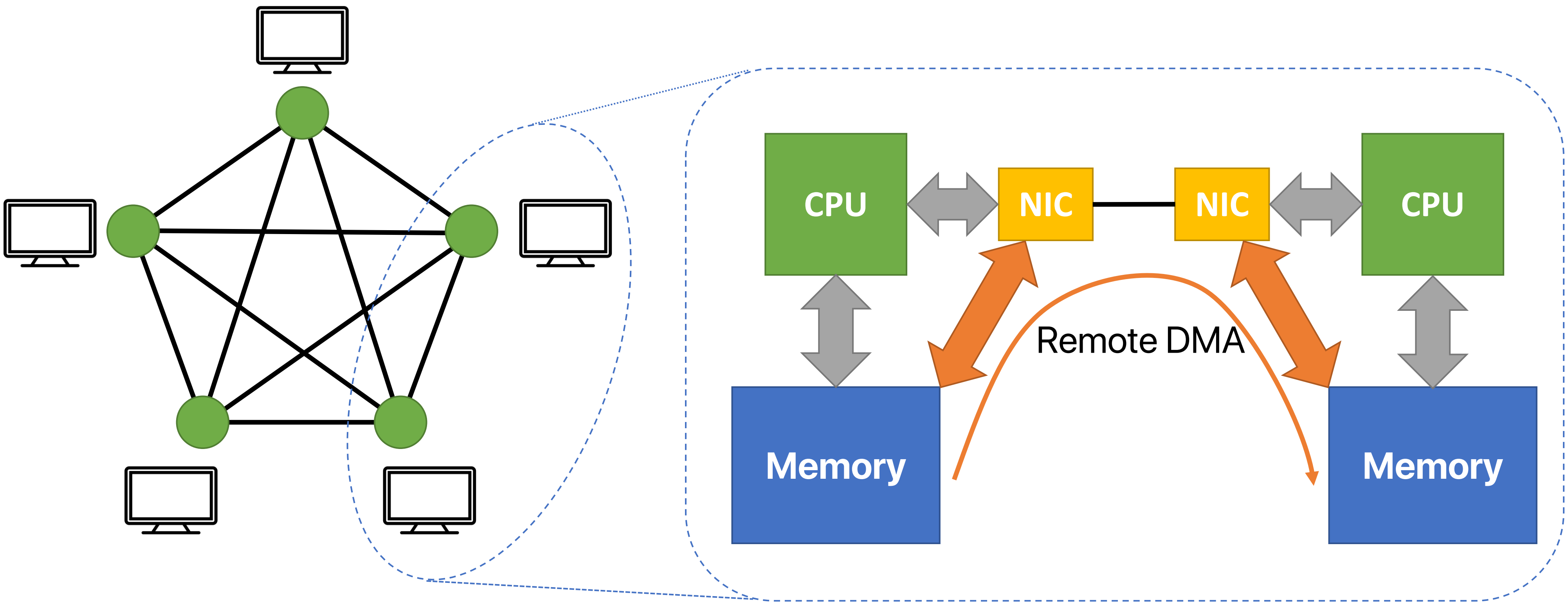
RDMA: Overview

- Networking hardware feature
- Direct access to remote memory
  - No CPU at remote side
  - No OS at either side
- Good performance
  - ~1 us latency
  - ~100-800 Gbps bandwidth
- Configurable access permissions

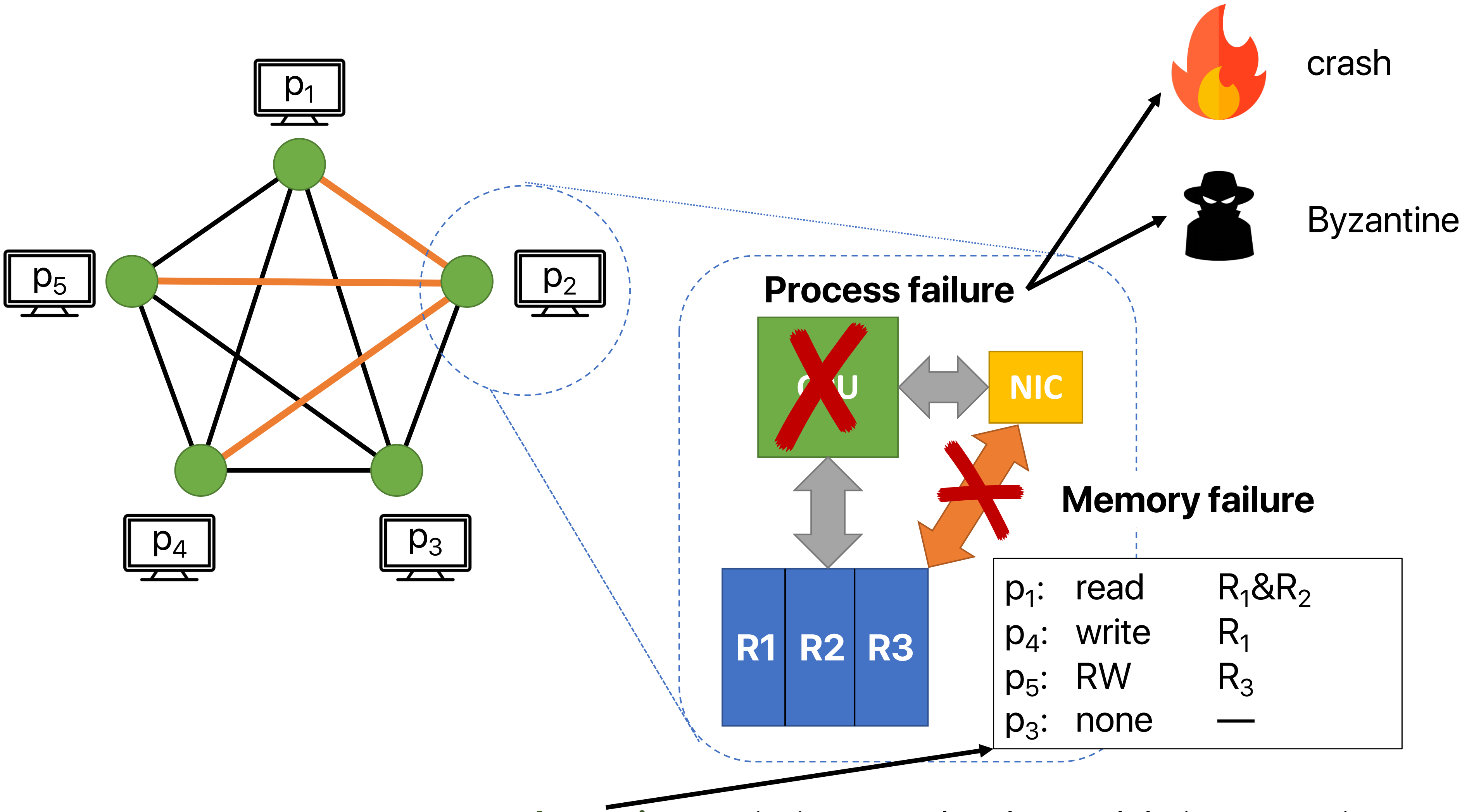


RDMA

Remote Direct Memory Access (RDMA)

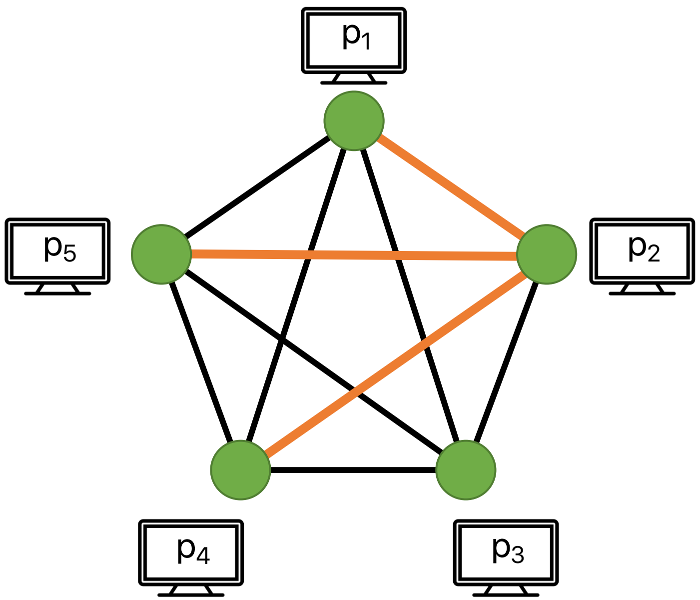


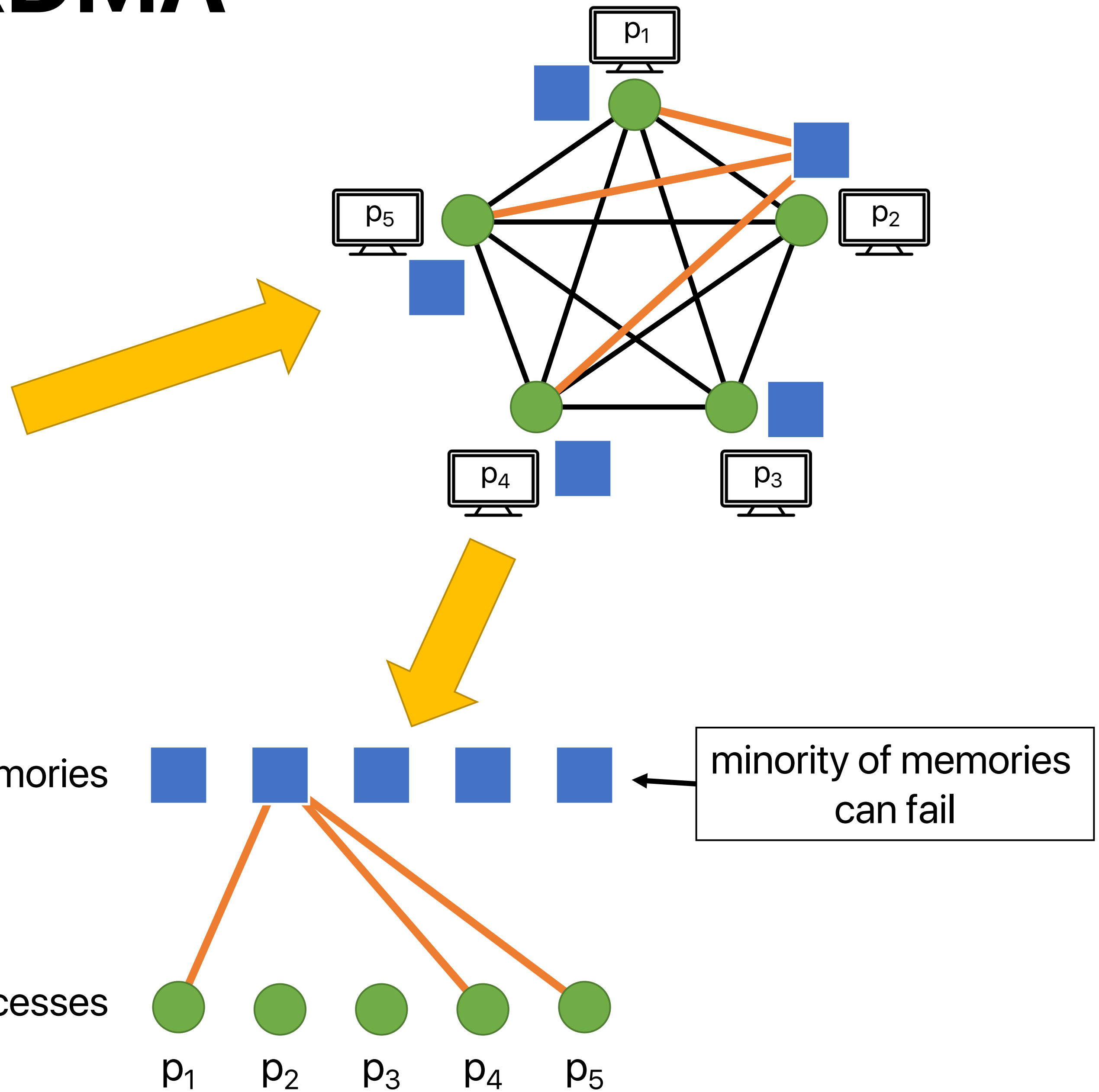
RDMA: Permissions and Failures



dynamic permissions: can be changed during execution

Modelling RDMA





Also called
"disaggregated memory"

Outline

- Introduction
- 3 remarkable results with RDMA:
  - Consensus with crash faults
  - Broadcast with Byzantine faults
  - Fast memory replication

Best of Both Worlds

n = num processes f = num failures	Crash	
	Fault Tolerance	Common-case Performance
Message Passing	$n > 2f$	**2** [Lamport'98]
Shared Memory	$\boldsymbol{n > f}$	4 [GL'02]
RDMA	$\boldsymbol{n > f}$	**2**

Refresher: O-Consensus

Paxos in Shared Memory

```
propose(v):
 while(true)
   Reg[i].T.write(ts);                    announce my timestamp
   val := Reg[1,..,n].highestTspValue();
   if val = ⊥ then val := v;
   Reg[i].V.write(val,ts);                announce my value, ts
   if ts = Reg[1,..,n].highestTsp() then
      return(val)
   ts := ts + n
```

adopt value with highest ts (or mine if none)

if my timestamp is the highest, decide

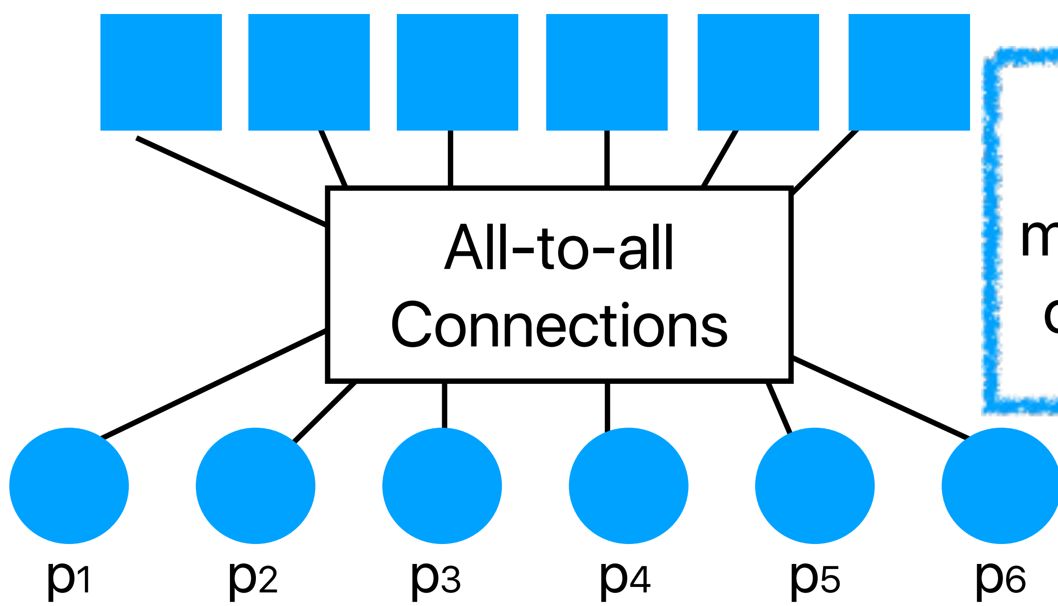
This assumes that shared memory never fails.

🤔 **What if memory can fail?** 🤔

Handling Memory Failures

Replication: Treat all memories the same

Send all write/read requests to all memories, wait to hear acknowledgement from majority



Instead of many faulty memories, we can now think of one non-faulty memory!

O-Consensus w Memory Failures

Disk Paxos [GafniLamport2002]

```
propose(v):
 while(true)
   for every memory m in parallel:
        Reg[m][i].T.write(ts);
        temp[m][1..n] = Reg[m][1..n].read();
    until completed for majority of memories
    val := temp[1..m][1..n].highestTspValue();
    if val = ⊥ then val := v;
    for every memory m in parallel:
        Reg[m][i].V.write(val,ts);
        temp[m][1..n] = Reg[m][1..n].read();
    until completed for majority of memories
    if ts = temp[1..m][1..n].highestTsp() then
        return(val)
    ts := ts + n
```

announce my timestamp

adopt value with highest ts (or mine if none)

announce my value, ts

if my timestamp is the highest, decide

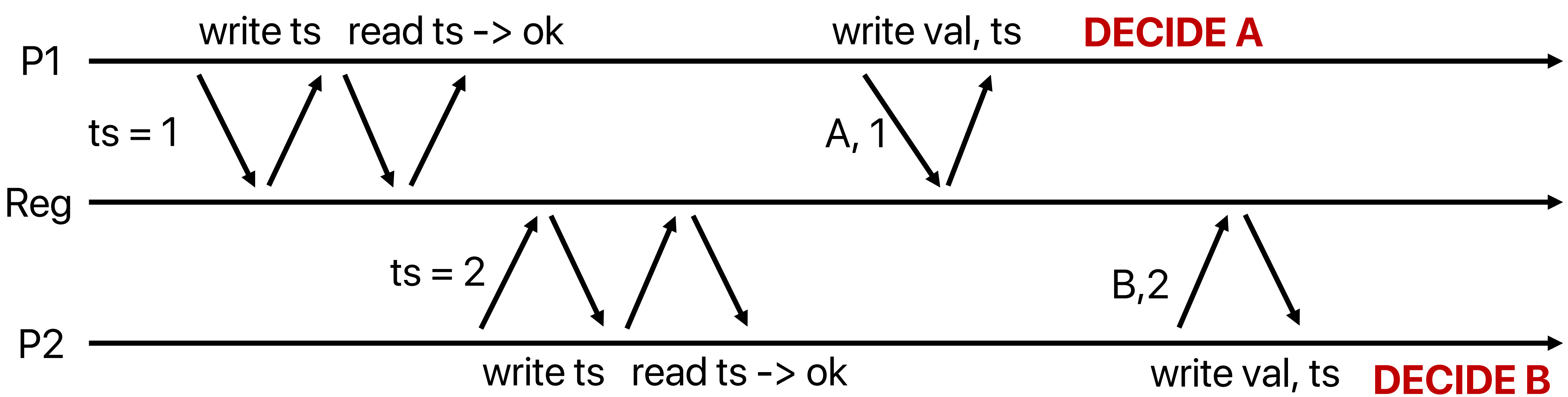
O-Consensus w Memory Failures

```
propose(v):
 while(true)
   for every memory m in parallel:
        Reg[m][i].T.write(ts);
        temp[m][1..n] = Reg[m][1..n].read();
    until completed for majority of memories
    val := temp[1..m][1..n].highestTspValue();
    if val = ⊥ then val := v;
    for every memory m in parallel:
        Reg[m][i].V.write(val,ts);
        temp[m][1..n] = Reg[m][1..n].read();
    until completed for majority of memories
    if ts = temp[1..m][1..n].highestTsp() then
        return(val)
    ts := ts + n
```

Why read again here?

☝️ **Need to check if I ran alone!**

What If We Didn't Read?



O-Consensus w Memory Failures

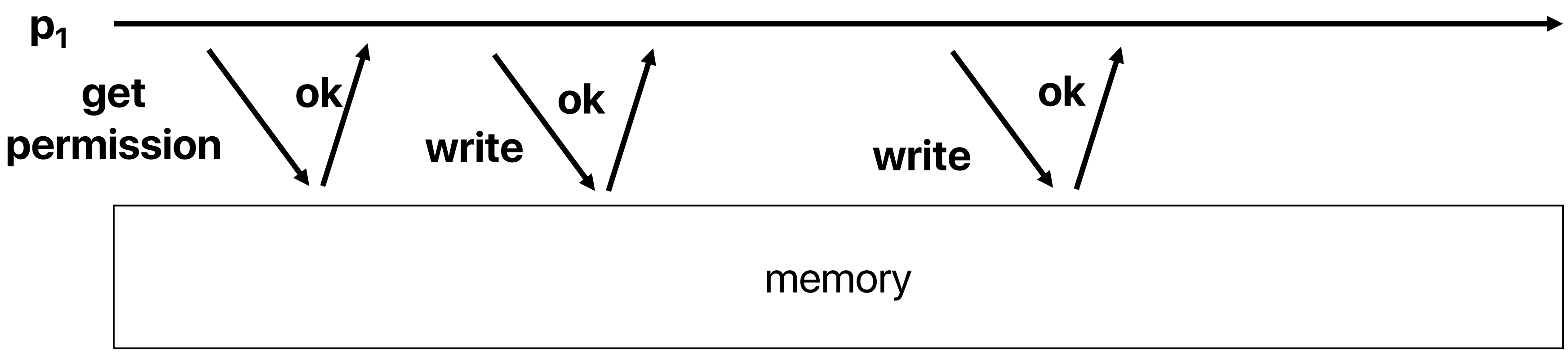
- If we don’t read again, we might miss a concurrent process’s timestamp
- This could lead to violation of agreement

- What if there was another way to determine if there was a concurrent process?
- We wouldn’t need the last read!

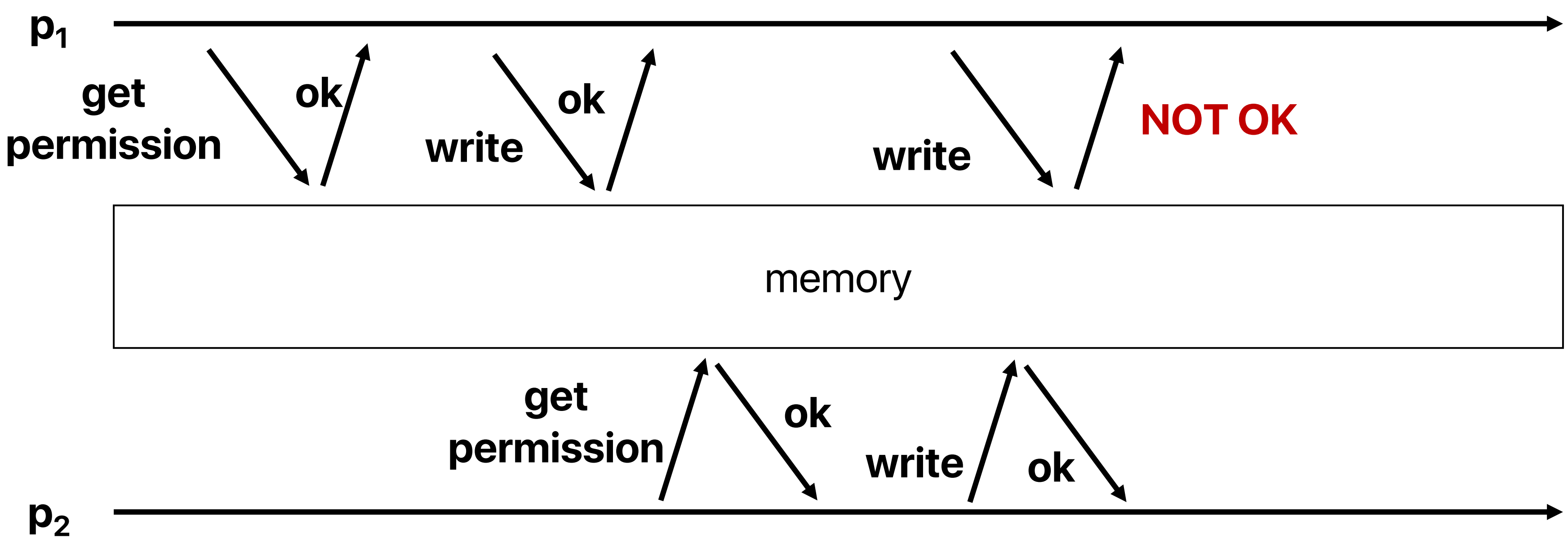
→ better complexity

Solo Detection w/ Permissions

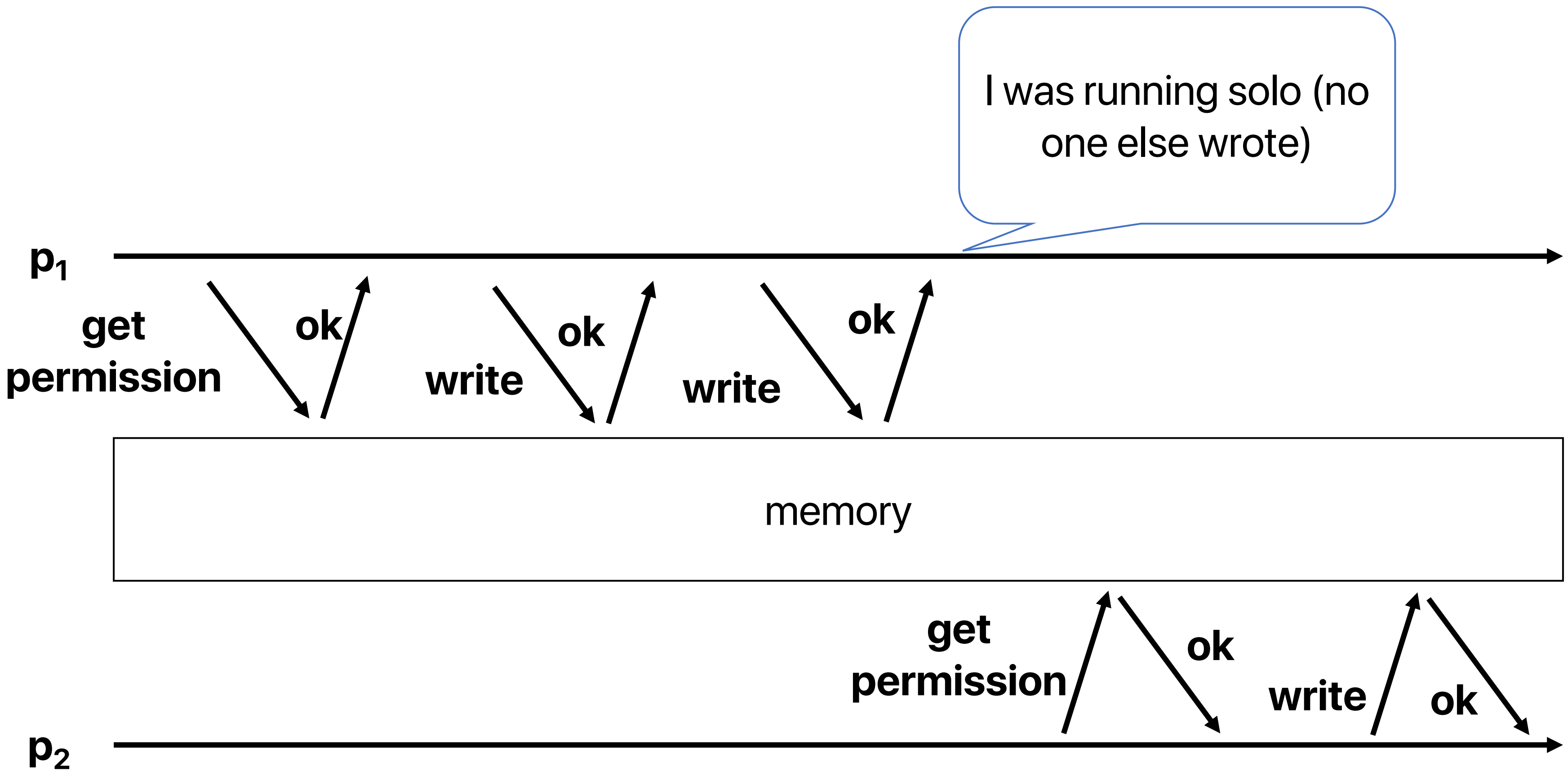
Idea: Memory gives write permission to the last process that requested it.
→ Only one process has write permission on a memory at any time.



Solo Detection w/ Permissions



Solo Detection w/ Permissions



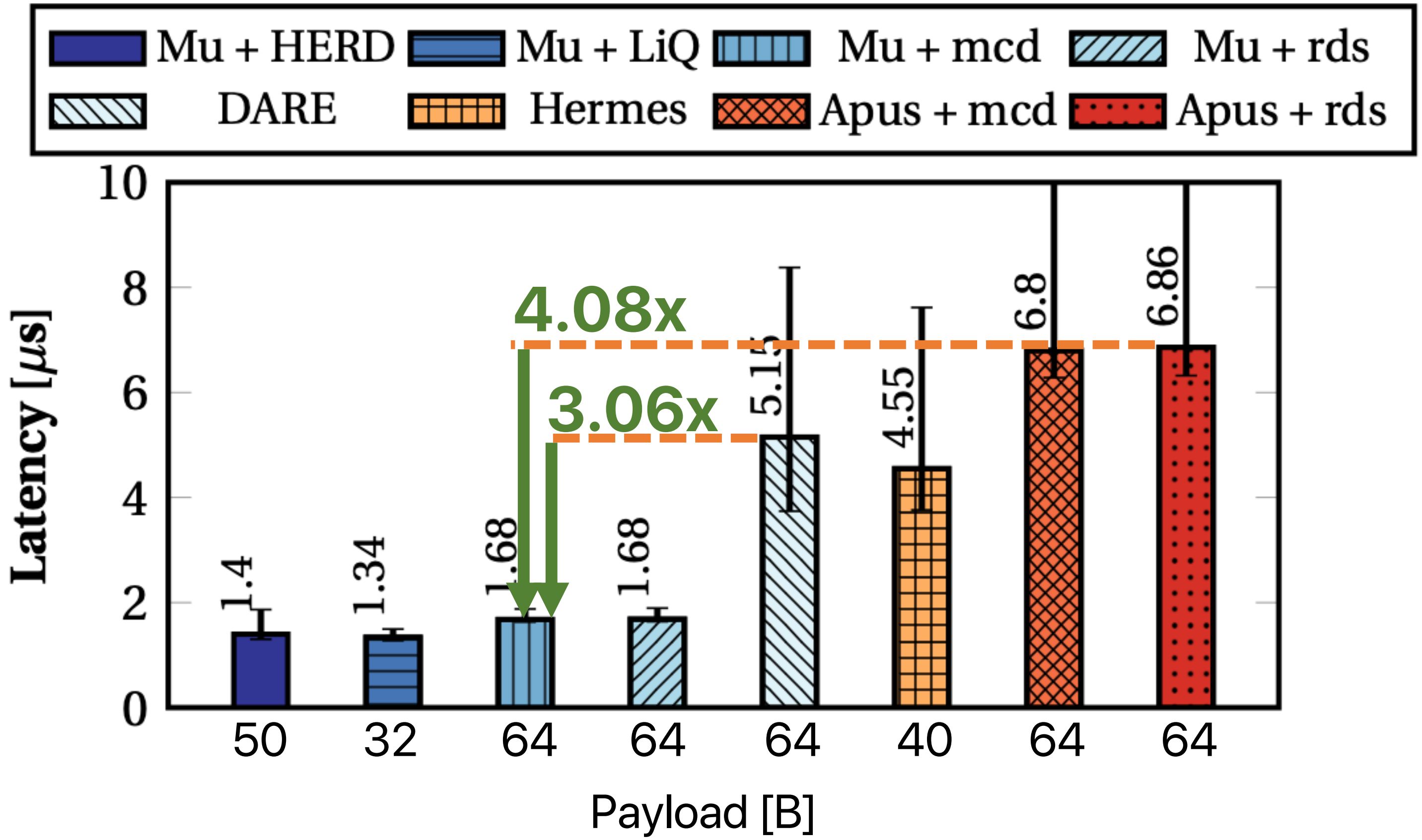
O-Consensus with Memory Failures and Permissions

```
propose(v):
 while(true)
    ts := ts + n
    for every memory m in parallel:
         m.getPermission();
         Reg[m][i].T.write(ts);
         temp[m][1..n] = Reg[m][1..n].read();
     until completed for majority of memories
     if ts < temp[1..m][1..n].highestTsp() then continue;
     val := temp[1..m][1..n].highestTspValue();
     if val = ⊥ then val := v;
     for every memory m in parallel:
         Reg[m][i].V.write(val,ts);
         temp[m][1..n] = Reg[m][1..n].read();
     until completed for majority of memories
     if writes succeeded at majority of memories then
         return(val)
```

No need to read again!

Quick Look: Replication Latency

[3x replication, 100Gbps Infiniband]



3-4x faster than state-of the art

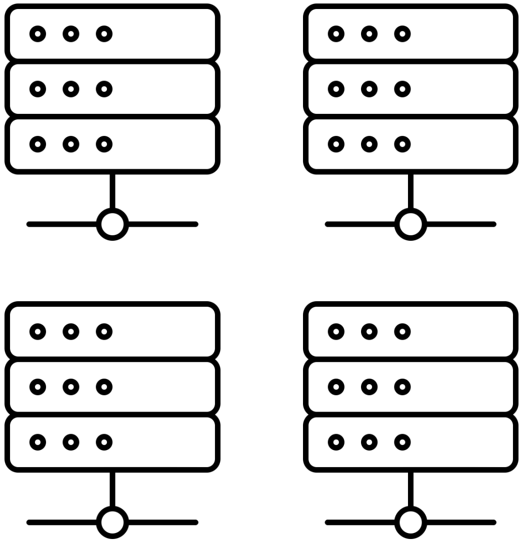
Outline

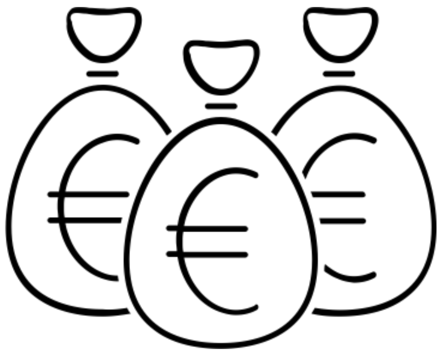
- Introduction
- 3 remarkable results with RDMA:
    - Consensus with crash faults
    - Broadcast with Byzantine faults
    - Fast memory replication

On Frugality

Number of replicas in the system

A system with $n = 3f + 1$ replicas has 33–50% more hardware than a system with $n = 2f + 1$, where f is the number of Byzantine replicas

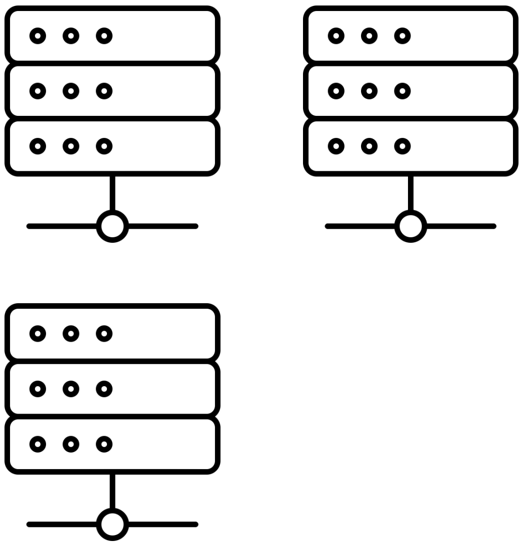


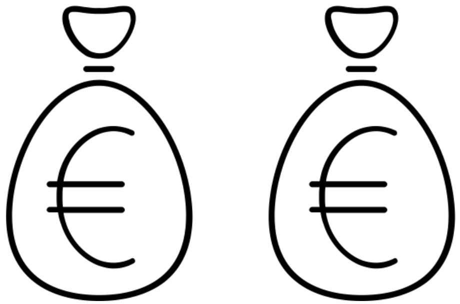


On Frugality

Number of replicas in the system

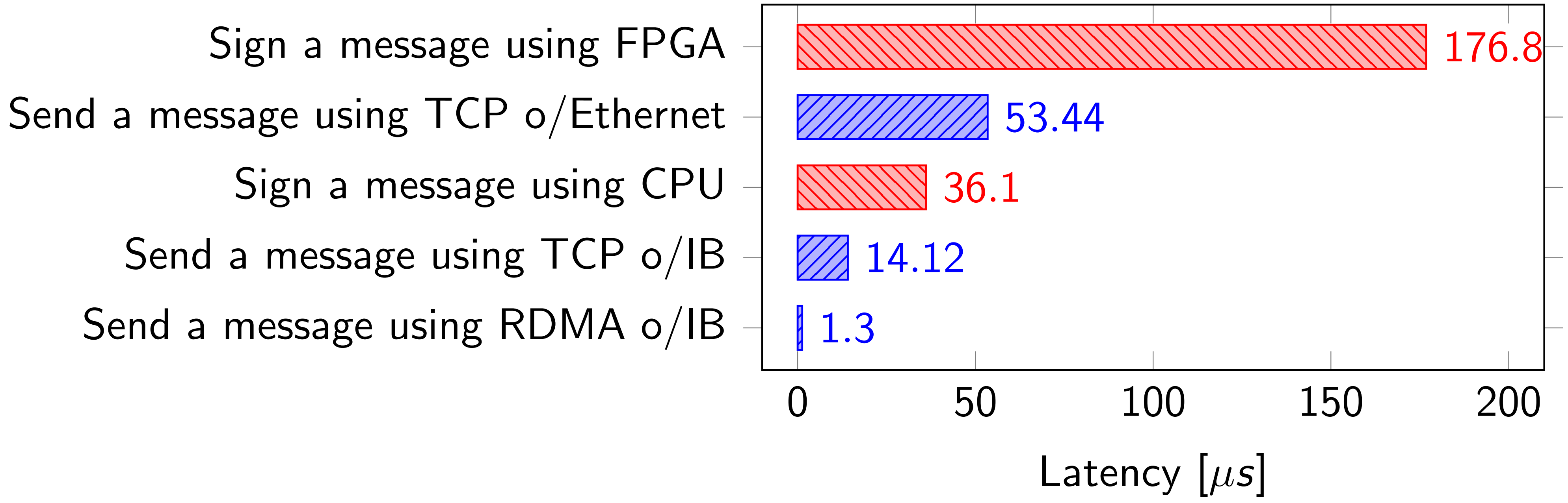
A system with $n = 3f + 1$ replicas has 33–50% more hardware than a system with $n = 2f + 1$, where f is the number of Byzantine replicas





On Frugality

Number of digital signatures



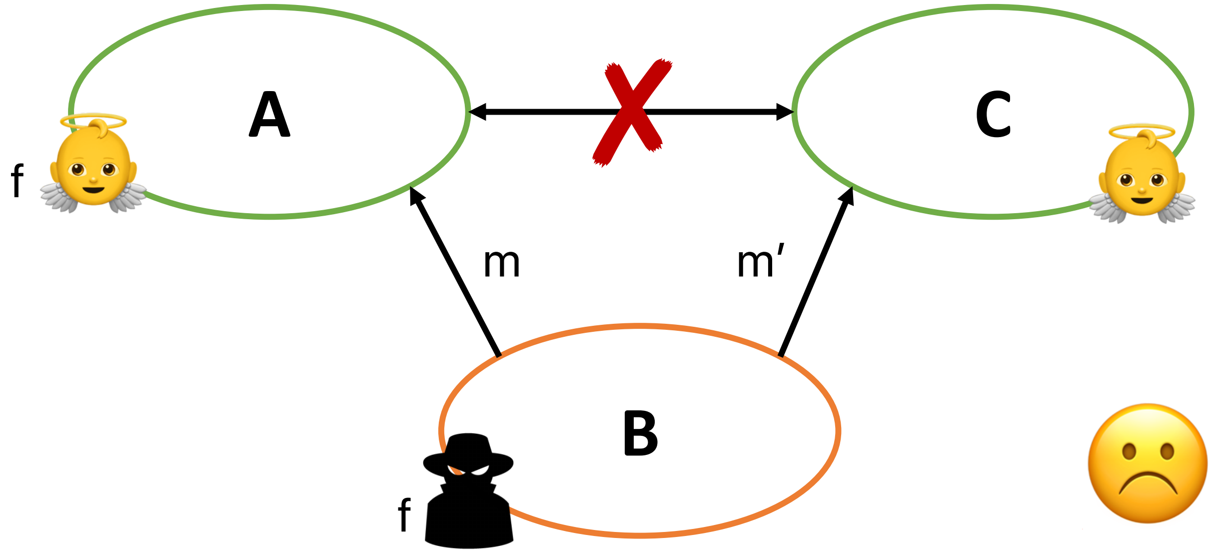
Goal

Address traditional distributed computing problems subject to Byzantine failures with few processes, $n = 2f + 1$, and few signatures

Equivocation

Preventing Equivocations in Message Passing

- Requires n=3f+1, where n is the total number of processes and up to f processes can be Byzantine
- Intuition:



Adversary can prevent correct processes from communicating

Byzantine fault-tolerance

Non-equivocation and digital signatures improve the fault-tolerance from $3f + 1$ to $2f + 1$ for reaching agreement

Byzantine fault-tolerance

Non-equivocation and digital signatures improve the fault-tolerance from $3f + 1$ to $2f + 1$ for reaching agreement

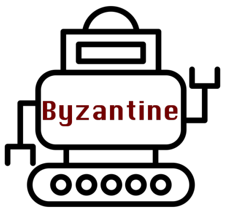
Shared memory provides non-equivocation capabilities:

Byzantine fault-tolerance

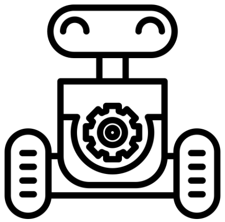
Non-equivocation and digital signatures improve the fault-tolerance from $3f + 1$ to $2f + 1$ for reaching agreement

Shared memory provides non-equivocation capabilities:

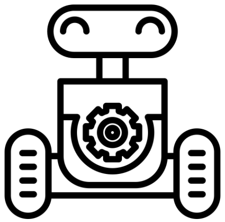
Shared memory M :



Process p_0



Process p_1

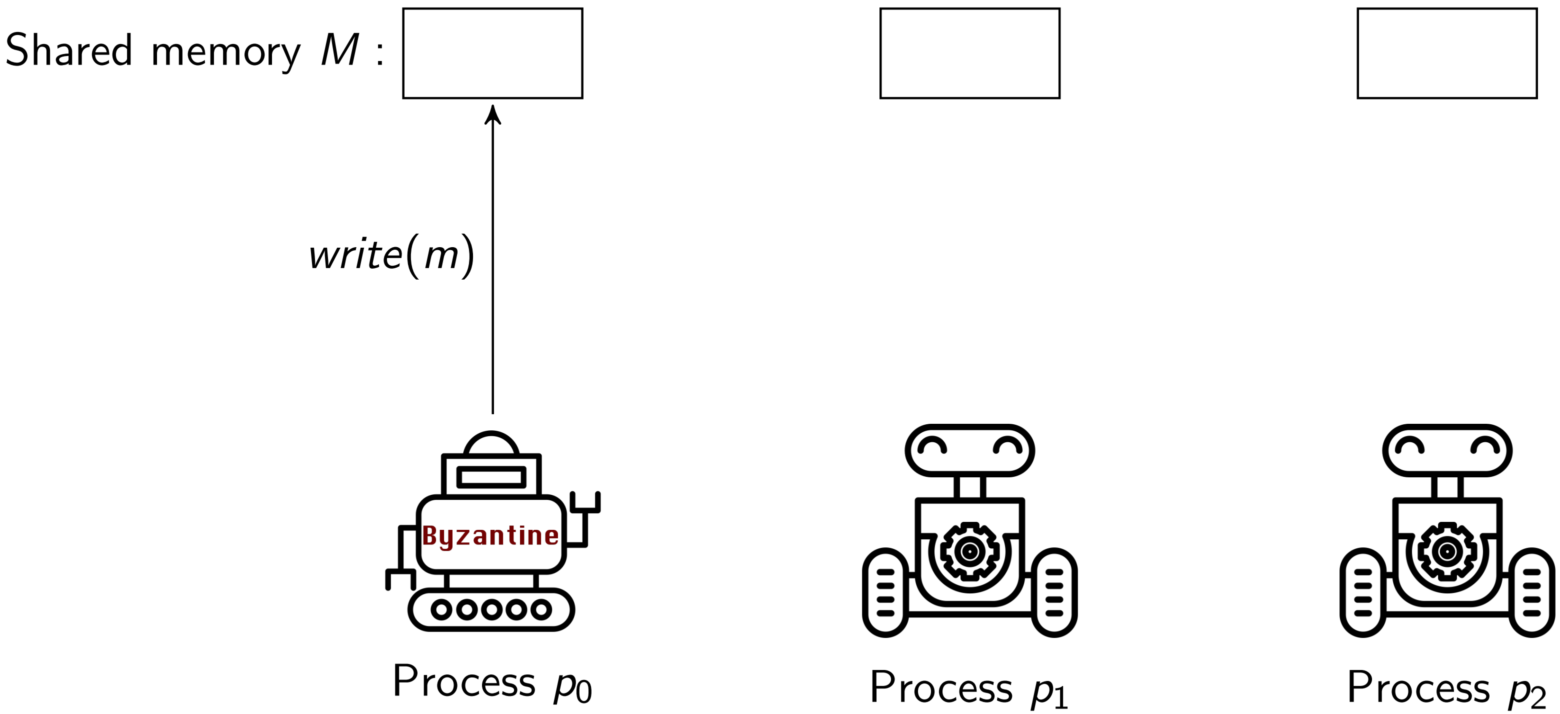


Process p_2

Byzantine fault-tolerance

Non-equivocation and digital signatures improve the fault-tolerance from $3f+1$ to $2f+1$ for reaching agreement

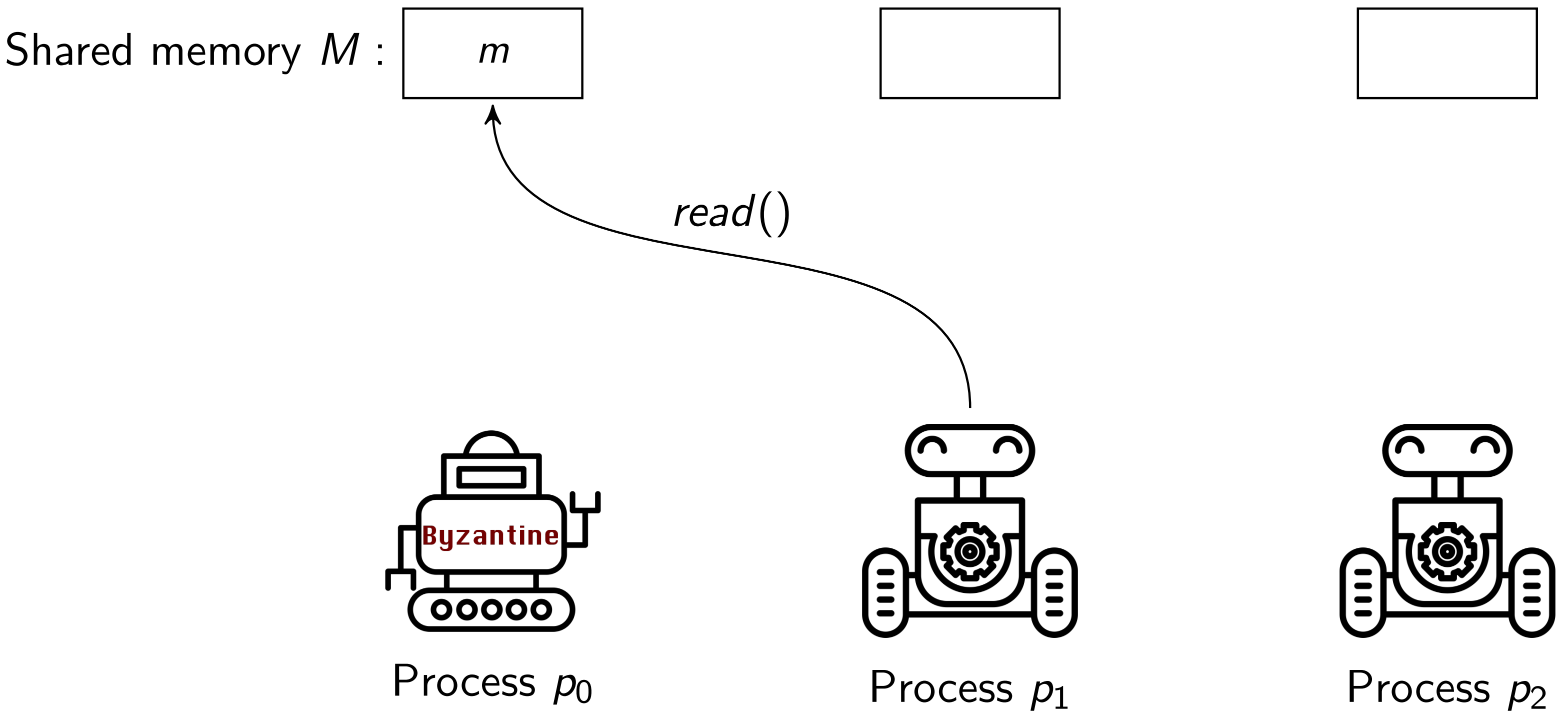
Shared memory provides non-equivocation capabilities:



Byzantine fault-tolerance

Non-equivocation and digital signatures improve the fault-tolerance from $3f+1$ to $2f+1$ for reaching agreement

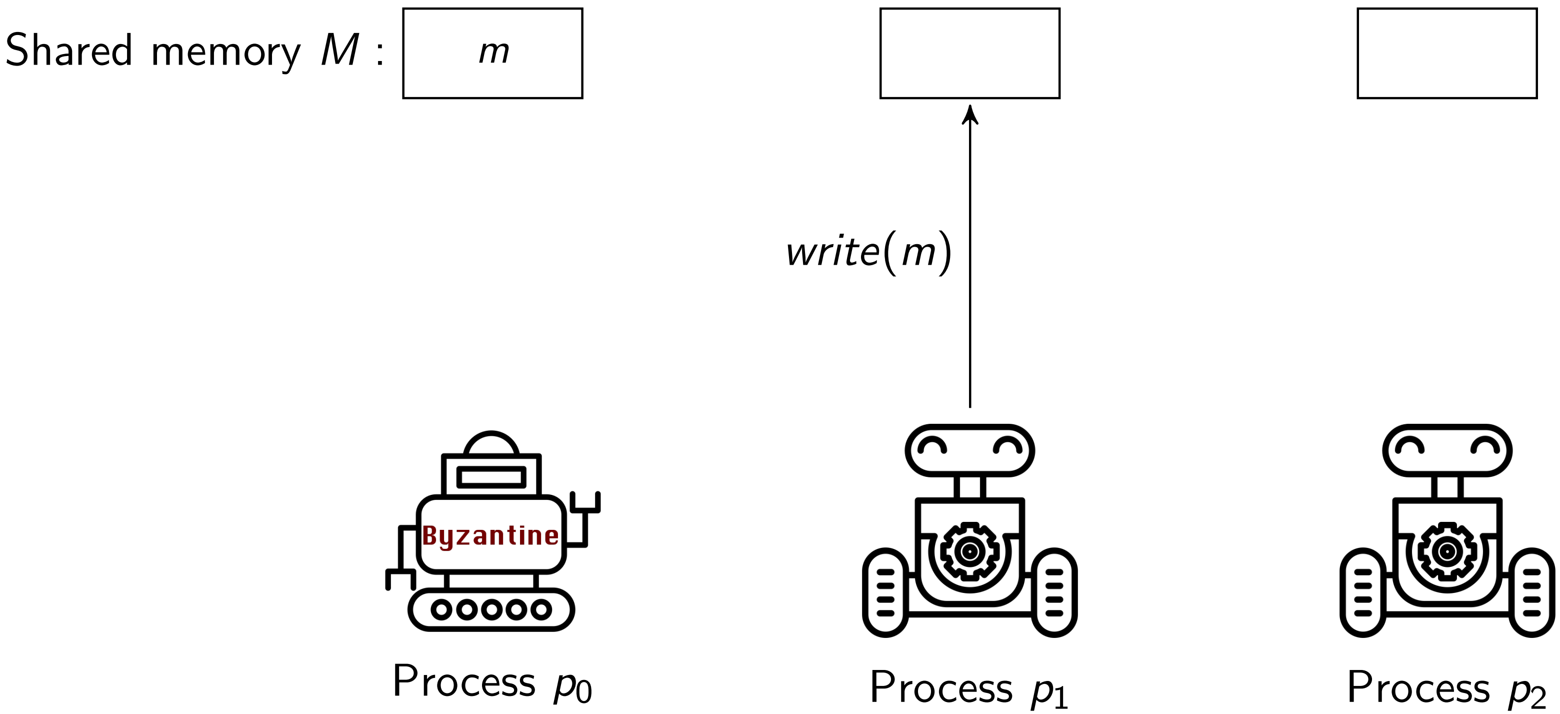
Shared memory provides non-equivocation capabilities:



Byzantine fault-tolerance

Non-equivocation and digital signatures improve the fault-tolerance from $3f+1$ to $2f+1$ for reaching agreement

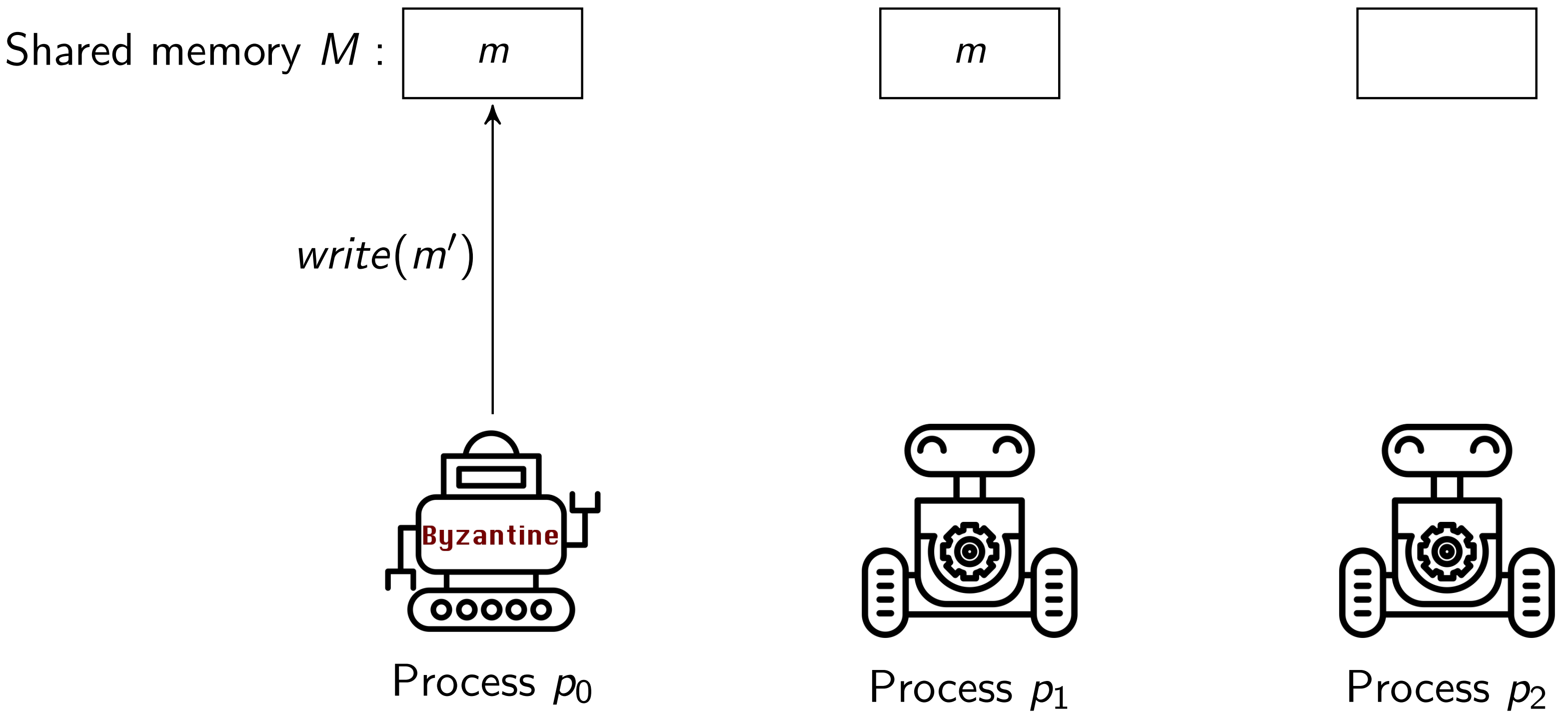
Shared memory provides non-equivocation capabilities:



Byzantine fault-tolerance

Non-equivocation and digital signatures improve the fault-tolerance from $3f+1$ to $2f+1$ for reaching agreement

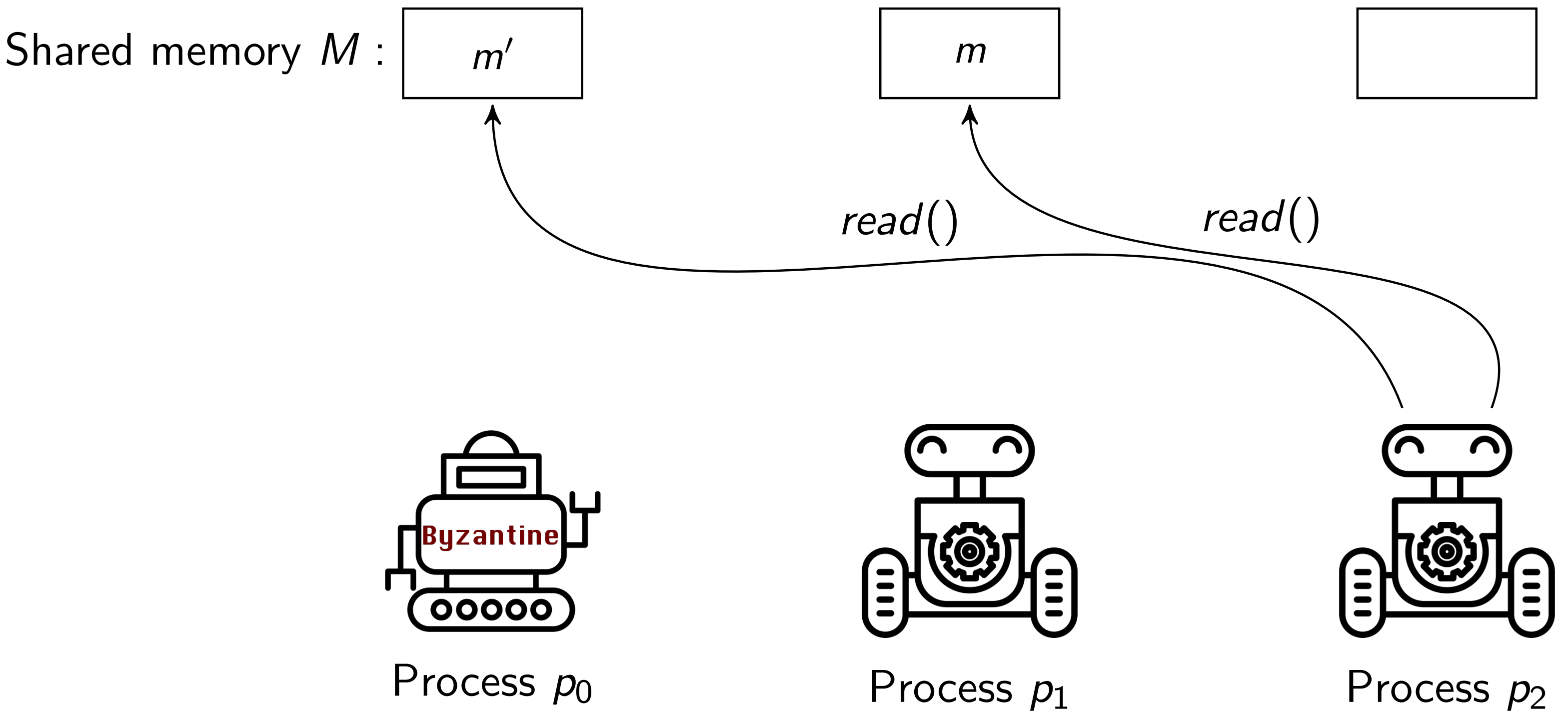
Shared memory provides non-equivocation capabilities:



Byzantine fault-tolerance

Non-equivocation and digital signatures improve the fault-tolerance from $3f+1$ to $2f+1$ for reaching agreement

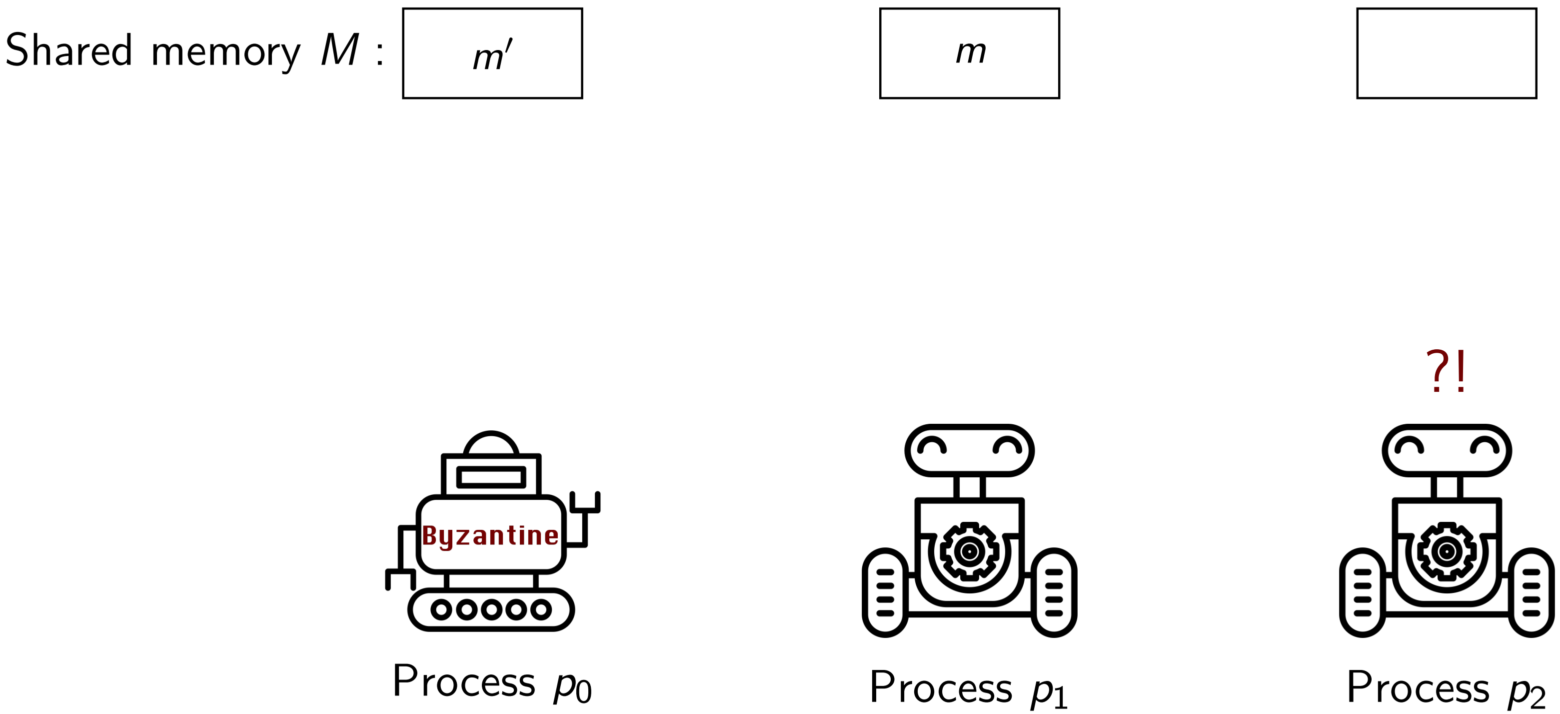
Shared memory provides non-equivocation capabilities:



Byzantine fault-tolerance

Non-equivocation and digital signatures improve the fault-tolerance from $3f + 1$ to $2f + 1$ for reaching agreement

Shared memory provides non-equivocation capabilities:



Model

Message-and-memory (M&M) [ABCGPT18] - allows processes to both pass messages and share memory M:

- Single-Writer Multi-Reader (SWMR) atomic registers
- individual memory may only fail by crashing

Model

Message-and-memory (M&M) [ABCGPT18] - allows processes to both pass messages and share memory M:

- Single-Writer Multi-Reader (SWMR) atomic registers
- individual memory may only fail by crashing

Signatures - each process has access to the primitives *sign* and *verify*

Model

Message-and-memory (M&M) [ABCGPT18] - allows processes to both pass messages and share memory M:

- Single-Writer Multi-Reader (SWMR) atomic registers
- individual memory may only fail by crashing

Signatures - each process has access to the primitives *sign* and *verify*

Up to f Byzantine processes, where $n = 2f + 1$

- cannot write on a register that is not its own
- cannot forge the signature of a correct process

Outline

1. Algorithms for Consistent and Reliable Broadcast
   - Signature-free in well-behaved executions

Outline

1. Algorithms for Consistent and Reliable Broadcast
   - Signature-free in well-behaved executions

2. Lower bounds for Consistent and Reliable Broadcast

Consistent Broadcast	Reliable Broadcast
1	$O(n)$

Table: Total number of signatures created by correct processes

Outline

1. Algorithms for Consistent and Reliable Broadcast
   - Signature-free in well-behaved executions

2. Lower bounds for Consistent and Reliable Broadcast

Consistent Broadcast	Reliable Broadcast
1	$O(n)$

Table: Total number of signatures created by correct processes

3. Consensus protocol using Consistent Broadcast

Process roles

Primitives: *broadcast*(m) and *deliver*(m)

Process roles

Primitives: *broadcast*(m) and *deliver*(m)

Sender s - the process that invokes *broadcast*(m)

Process roles

Primitives: $broadcast(m)$ and $deliver(m)$

Sender s - the process that invokes $broadcast(m)$

Replicator r - the process that ensures broadcast properties are satisfied (e.g., replicates messages)

Process roles

Primitives: $broadcast(m)$ and $deliver(m)$

Sender s - the process that invokes $broadcast(m)$

Replicator r - the process that ensures broadcast properties are satisfied (e.g., replicates messages)

Receiver p - the process that invokes $deliver(m)$

Process roles

Primitives: *broadcast*(m) and *deliver*(m)

Sender s - the process that invokes *broadcast*(m)

Replicator r - the process that ensures broadcast properties are satisfied (e.g., replicates messages)

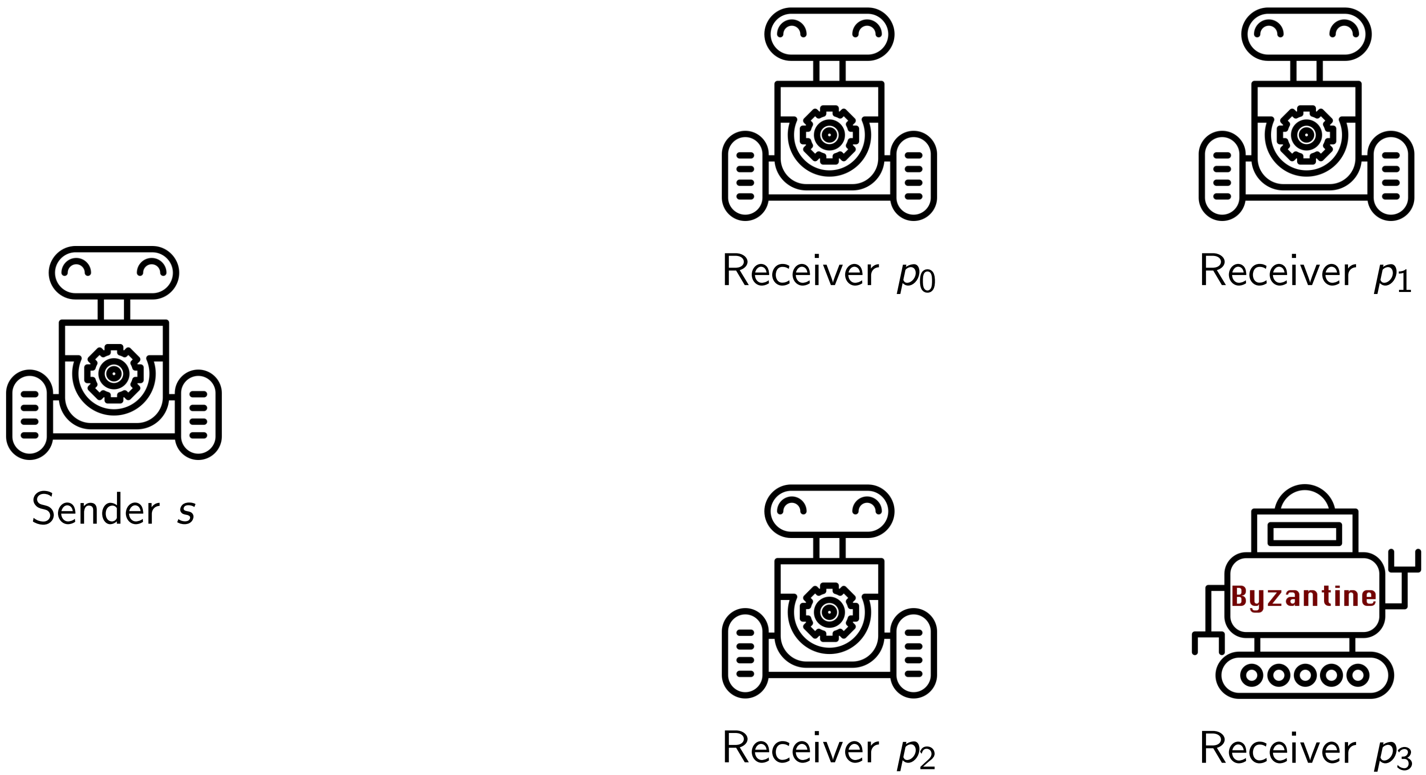
Receiver p - the process that invokes *deliver*(m)

n and f refer to the replicators

Consistent Broadcast

Validity

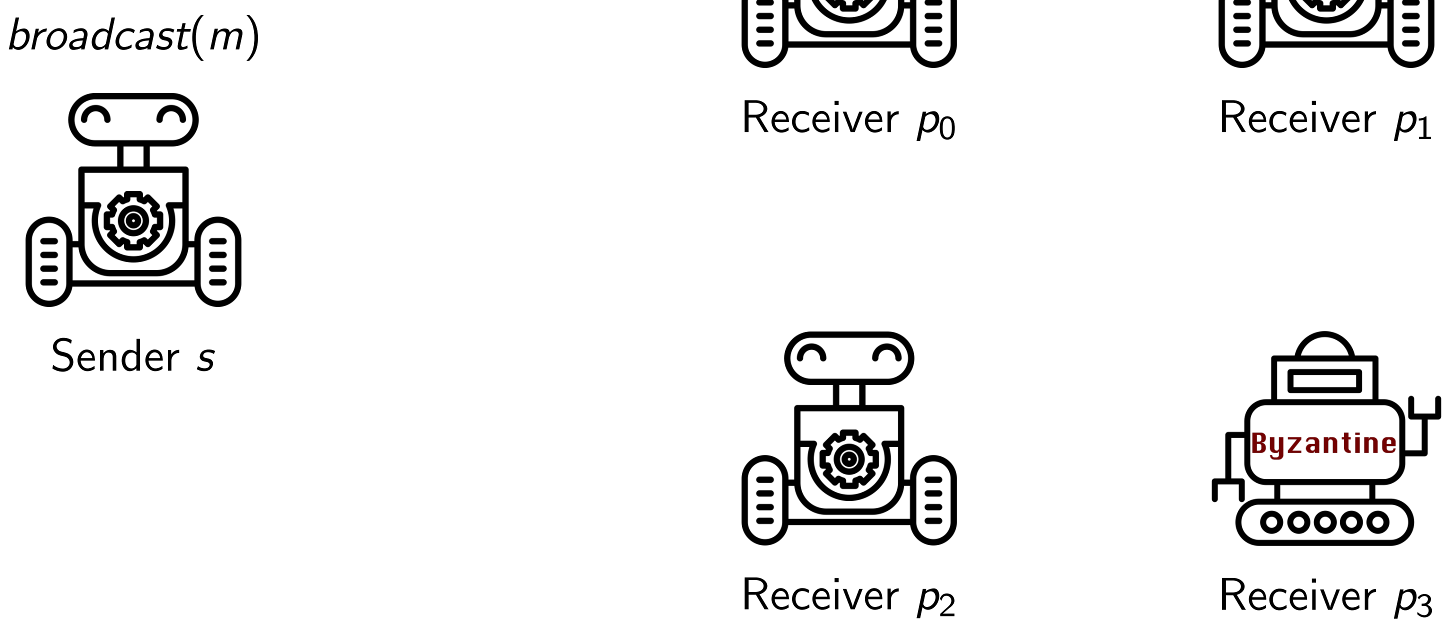
If a correct process s broadcasts m, then every correct process eventually delivers m



Consistent Broadcast

Validity

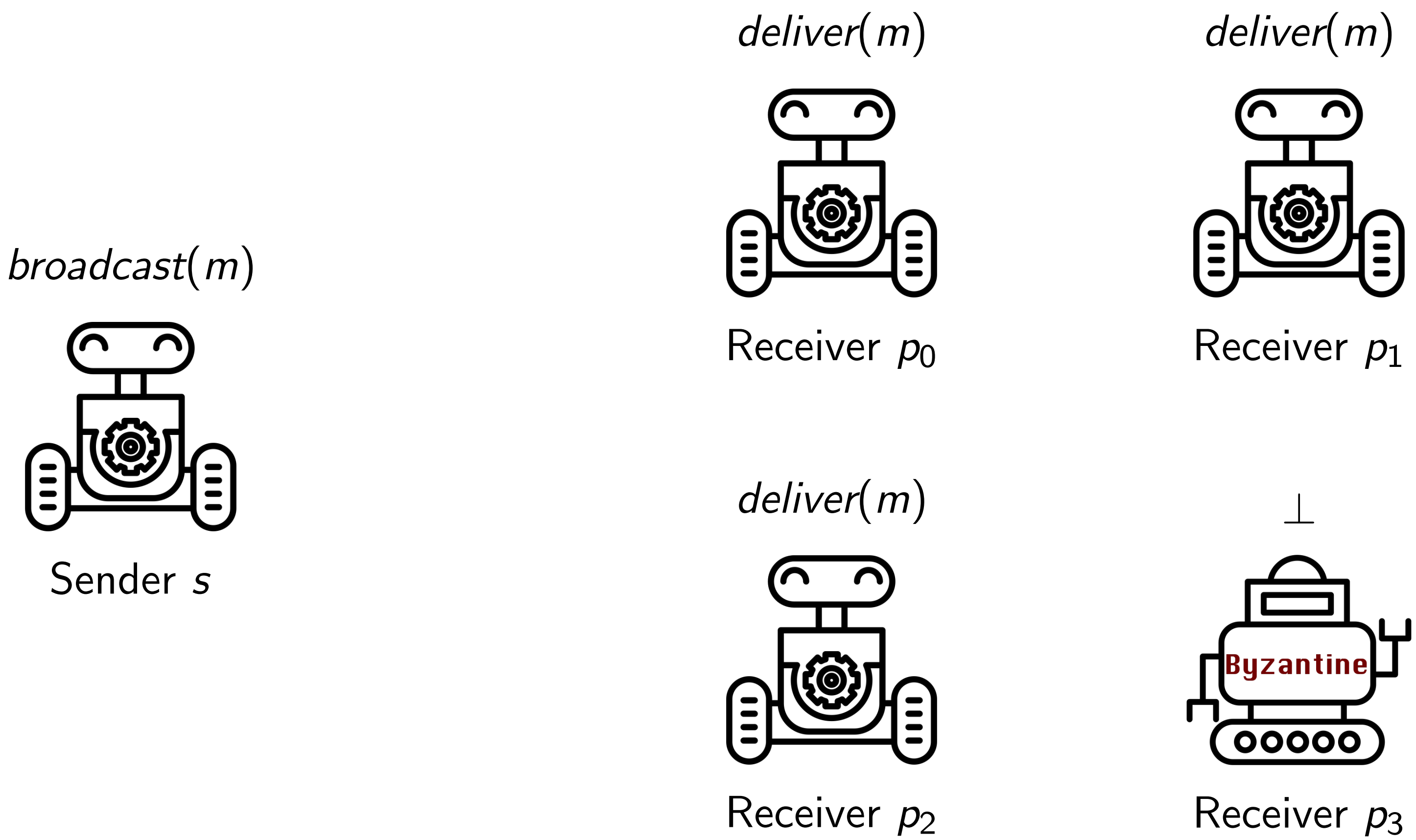
If a correct process s broadcasts m, then every correct process eventually delivers m



Consistent Broadcast

Validity

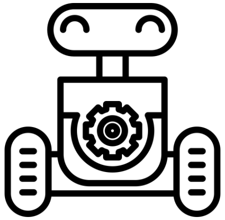
If a correct process s broadcasts m, then every correct process eventually delivers m



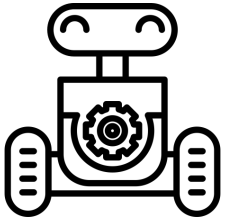
Consistent Broadcast

Consistency

If p and p' are correct processes, p delivers m, and p' delivers m', then $m{=}m'$

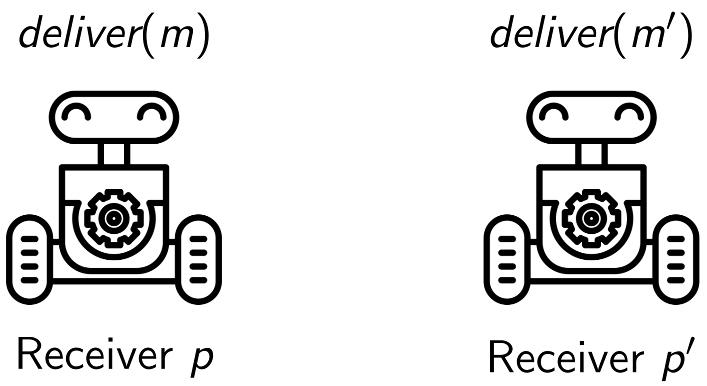


Receiver p



Receiver p'

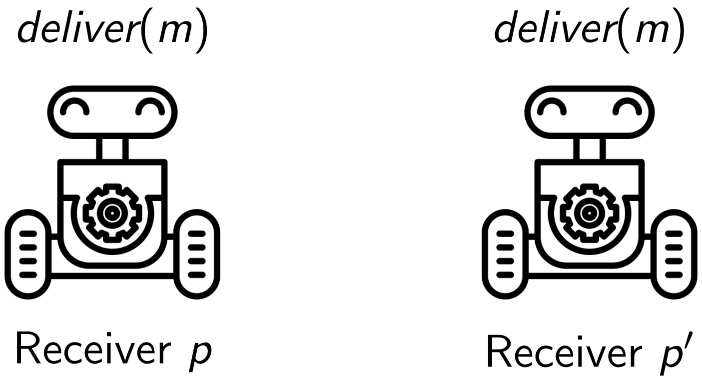
If p and p' are correct processes, p delivers m, and p' delivers m', then $m = m'$



Consistent Broadcast

Consistency

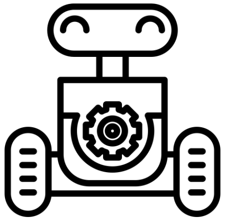
If p and p' are correct processes, p delivers m, and p' delivers m', then $m{=}m'$



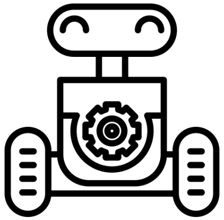
Consistent Broadcast

Integrity

If some correct process delivers m and s is correct, then s previously broadcast m



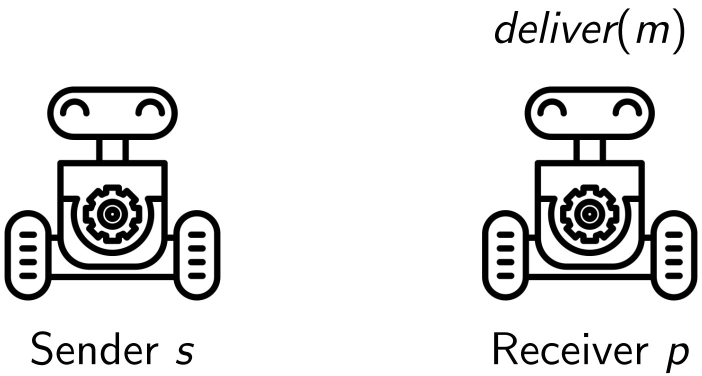
Sender s



Receiver p

Consistent Broadcast
Integrity

If some correct process delivers m and s is correct, then s previously broadcast m

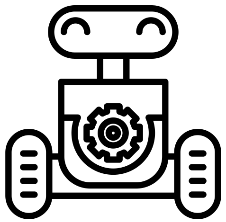


Consistent Broadcast

Integrity

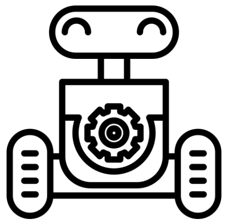
If some correct process delivers m and s is correct, then s previously broadcast m

broadcast(m)



Sender s

deliver(m)



Receiver p

Consistent Broadcast

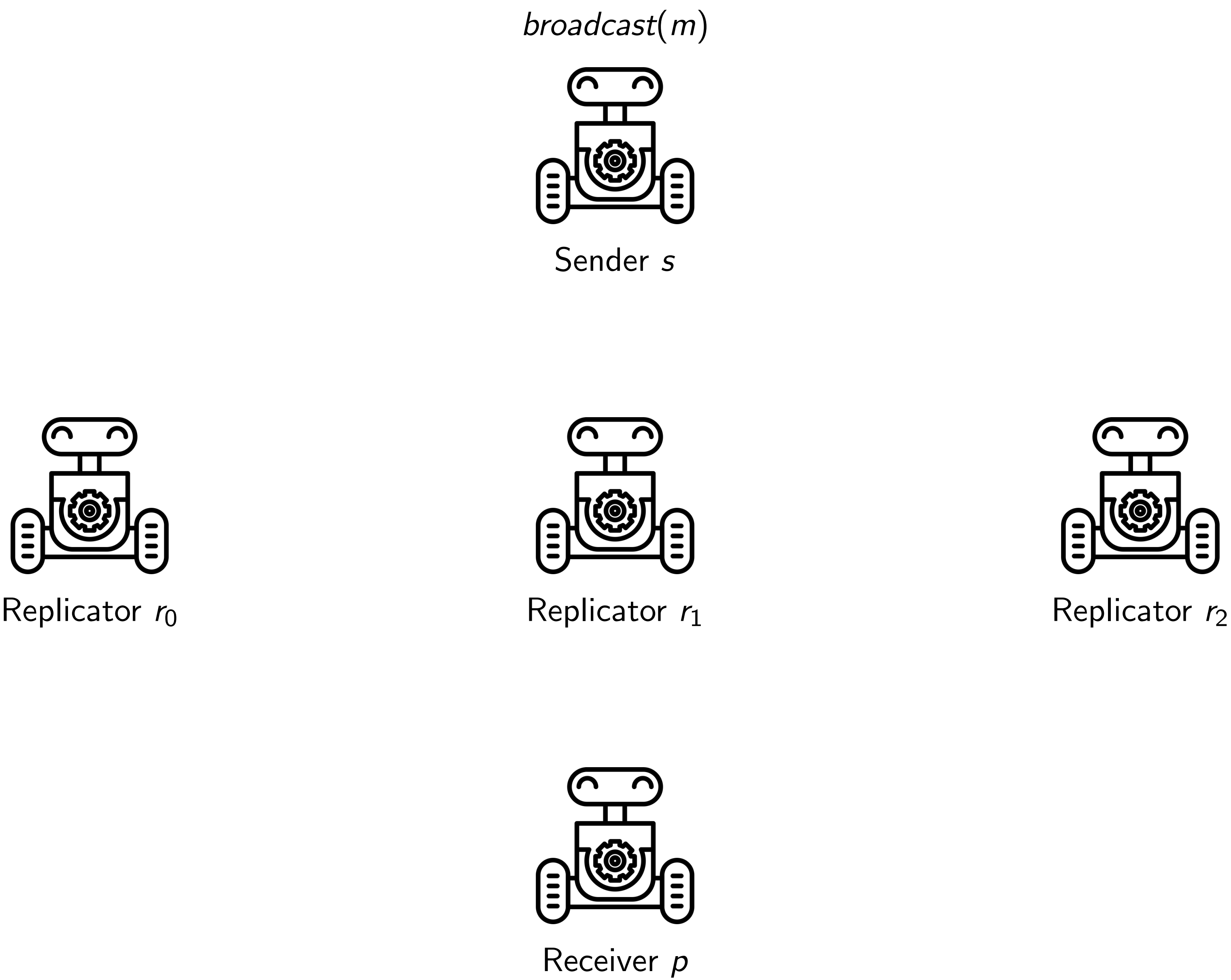
Validity - If a correct process s broadcasts m, then every correct process eventually delivers m

Consistency - If p and p' are correct processes, p delivers m, and p' delivers m', then $m{=}m'$

Integrity - If some correct process delivers m and s is correct, then s previously broadcast m

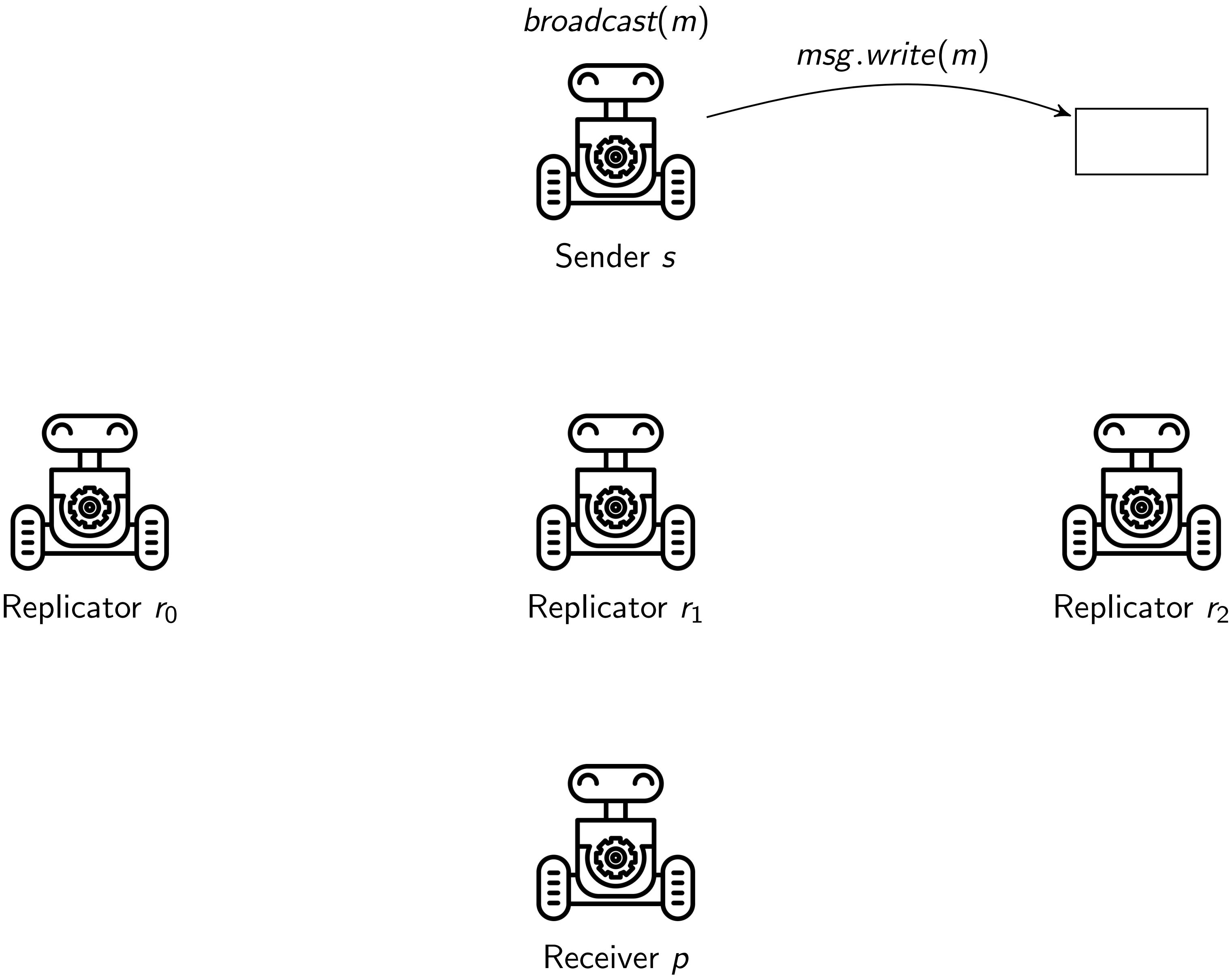
Consistent Broadcast

Algorithm sketch, $f = 1$. Fast path



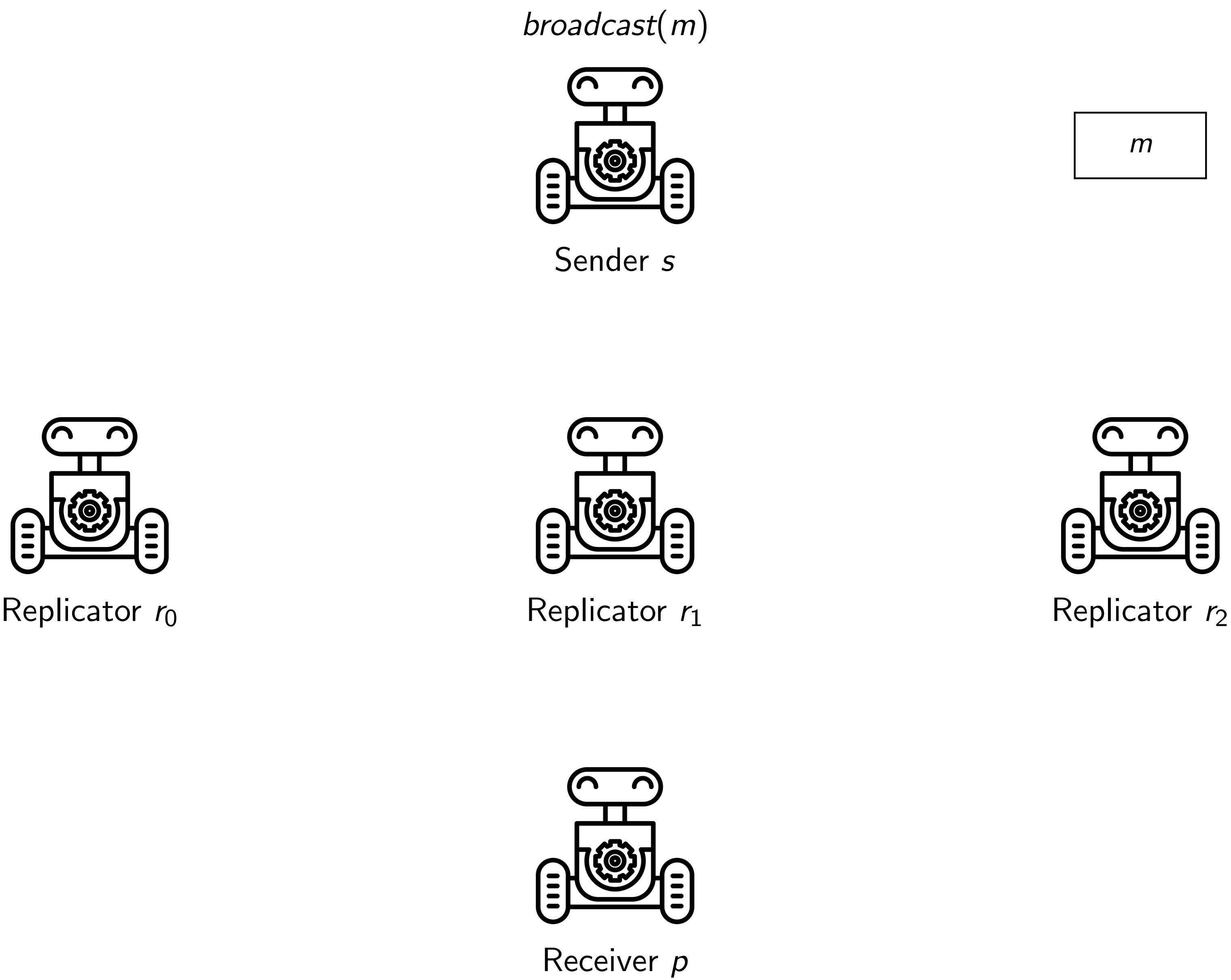
Consistent Broadcast

Algorithm sketch, $f = 1$. Fast path



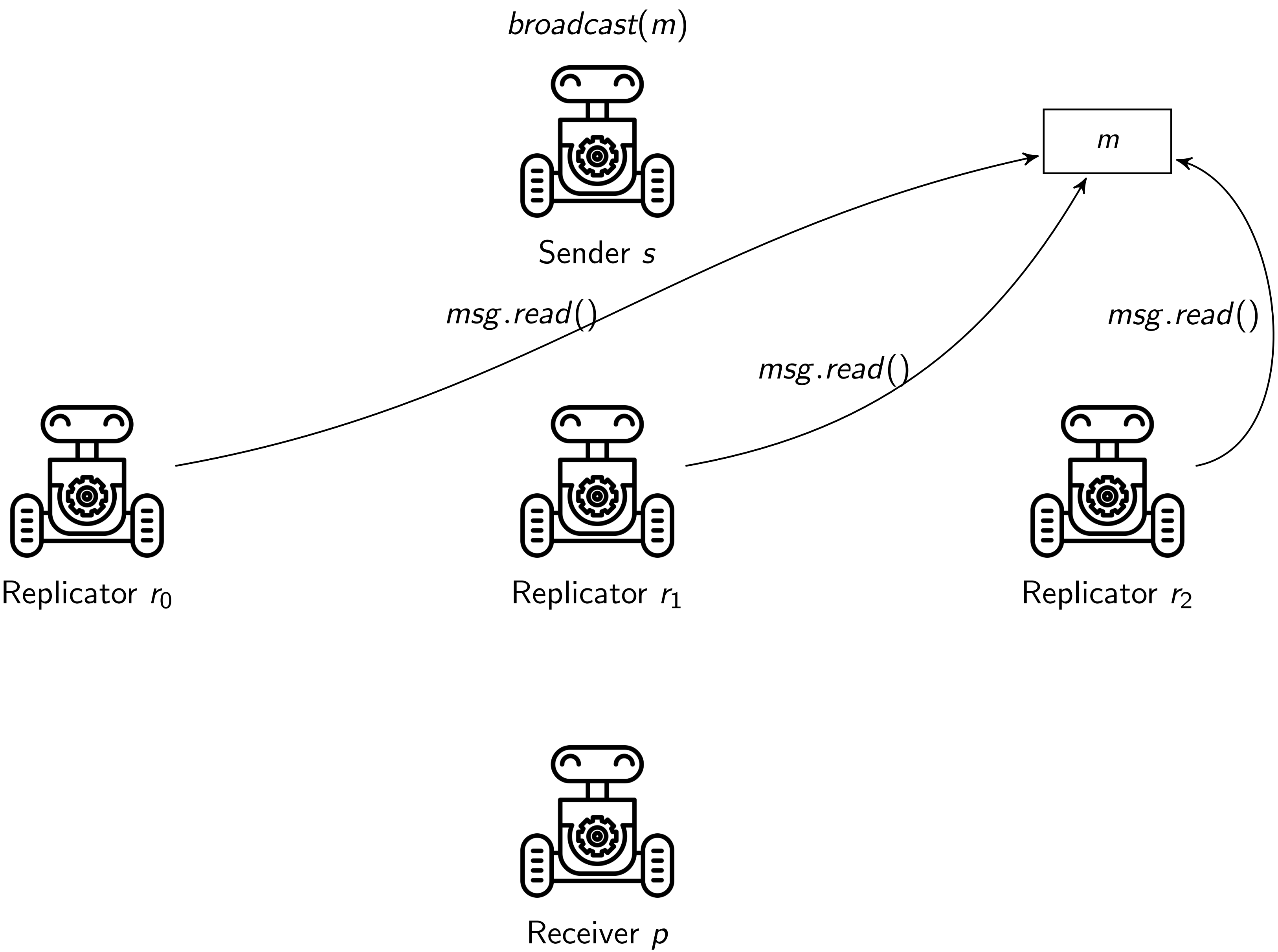
Consistent Broadcast

Algorithm sketch, $f = 1$. Fast path



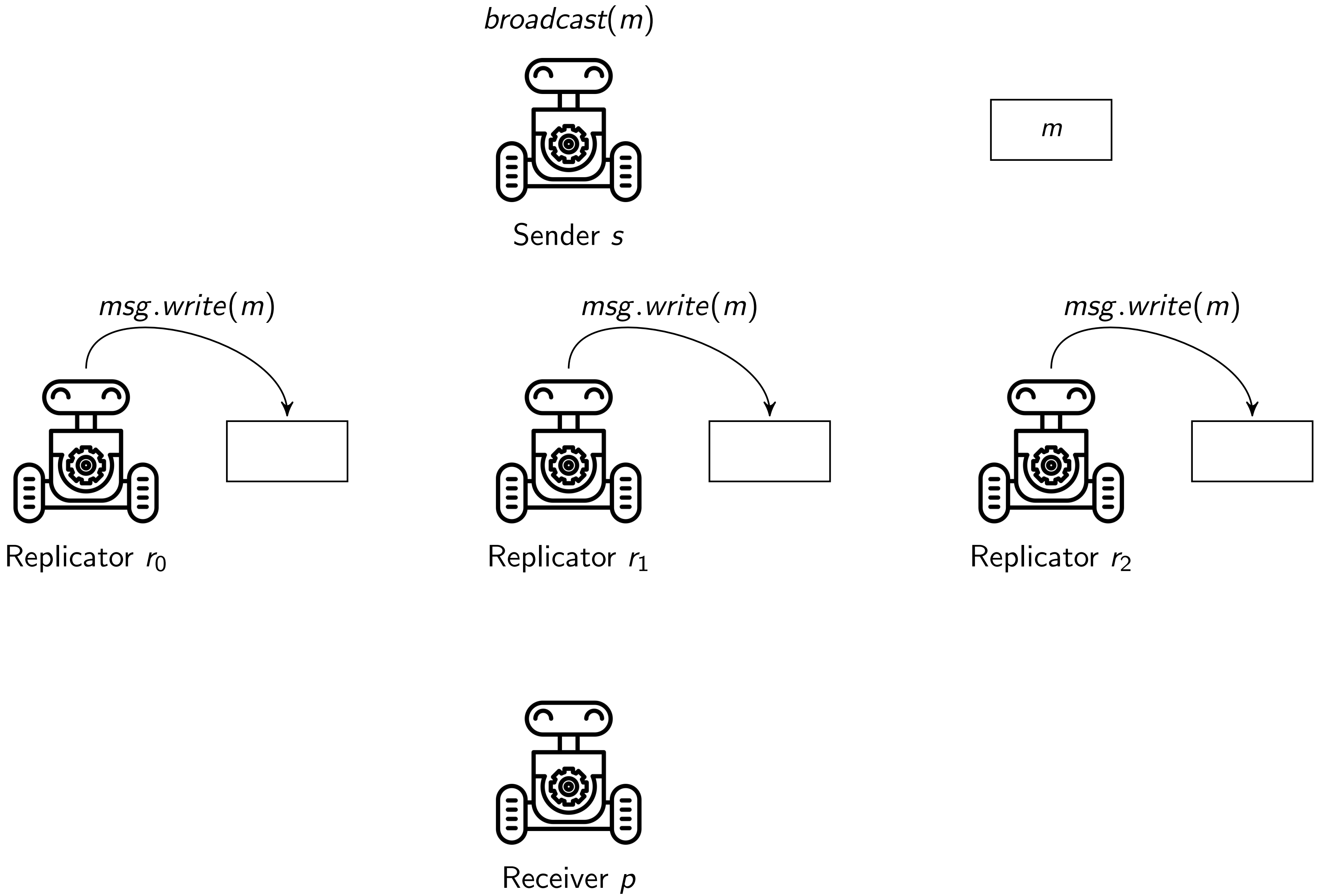
Consistent Broadcast

Algorithm sketch, $f = 1$. Fast path



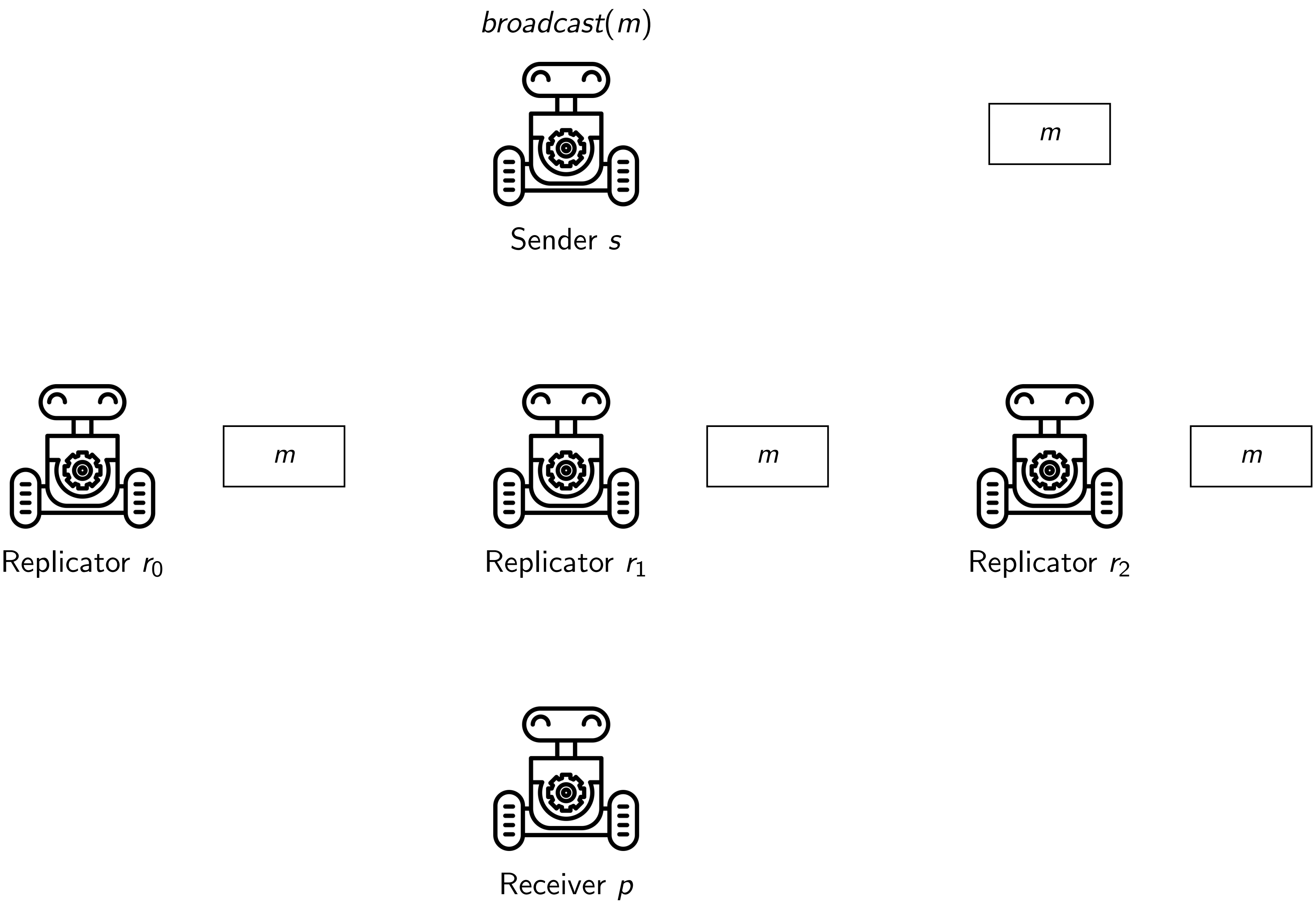
Consistent Broadcast

Algorithm sketch, $f = 1$. Fast path



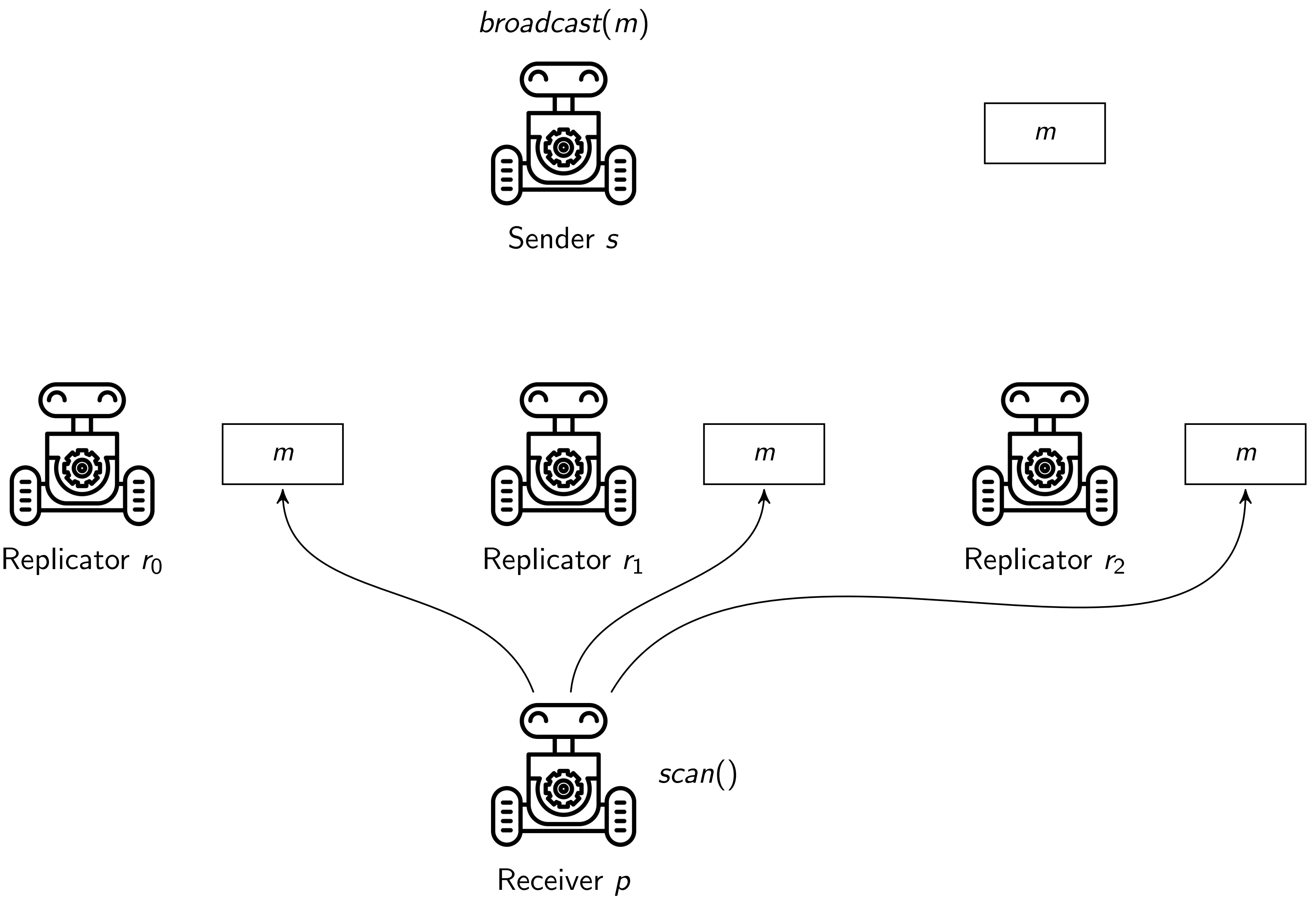
Consistent Broadcast

Algorithm sketch, $f = 1$. Fast path



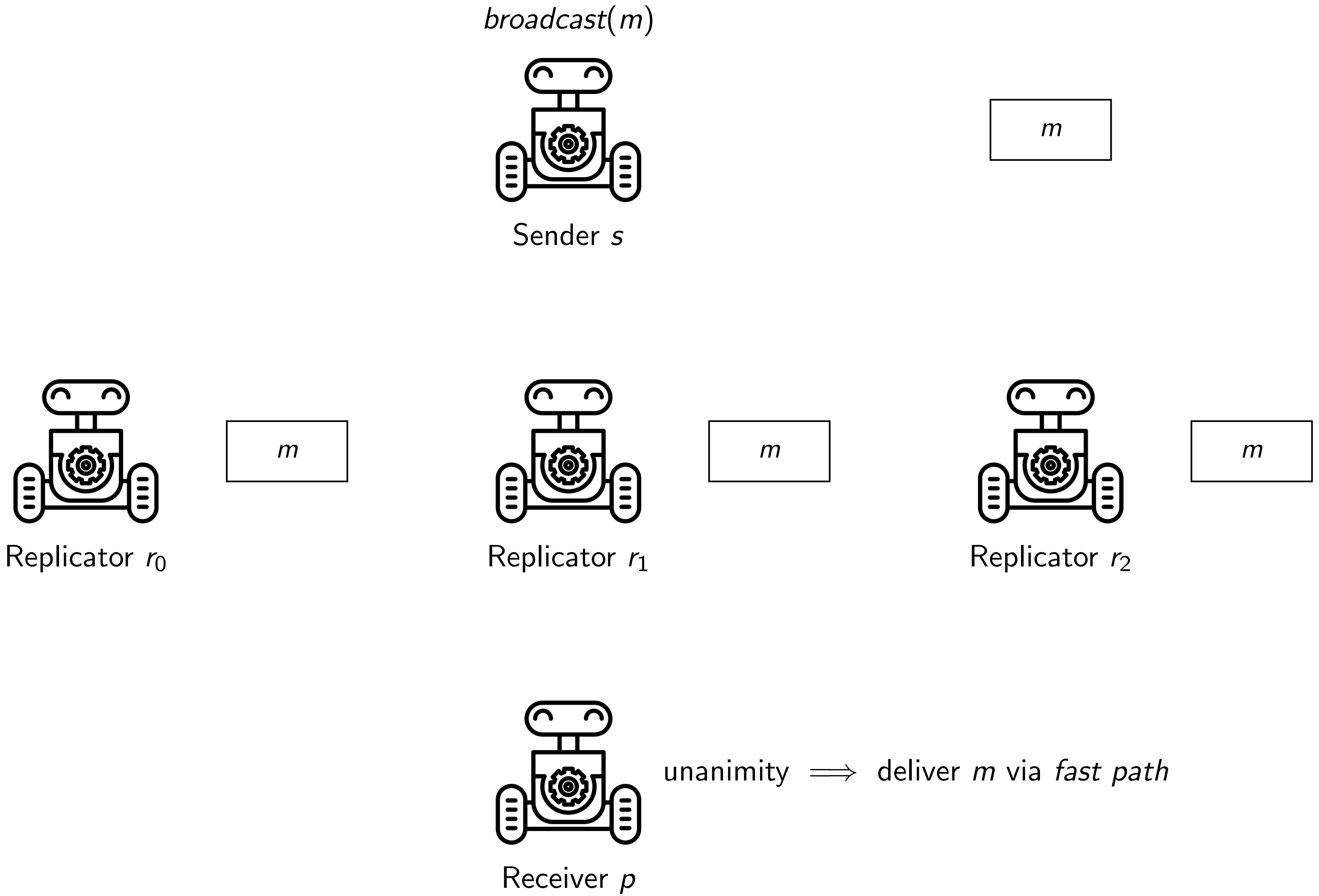
Consistent Broadcast

Algorithm sketch, $f = 1$. Fast path



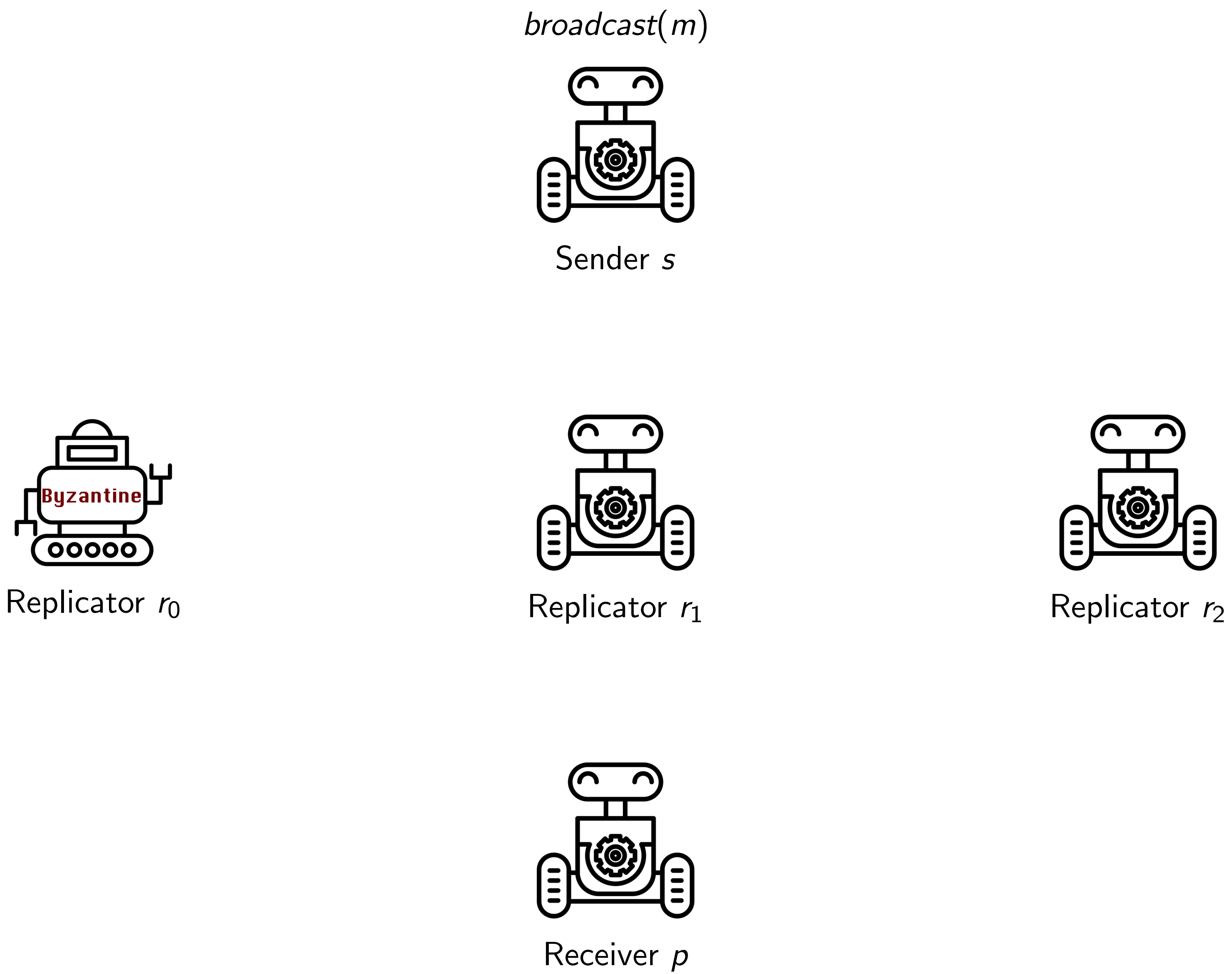
Consistent Broadcast

Algorithm sketch, $f = 1$. Fast path



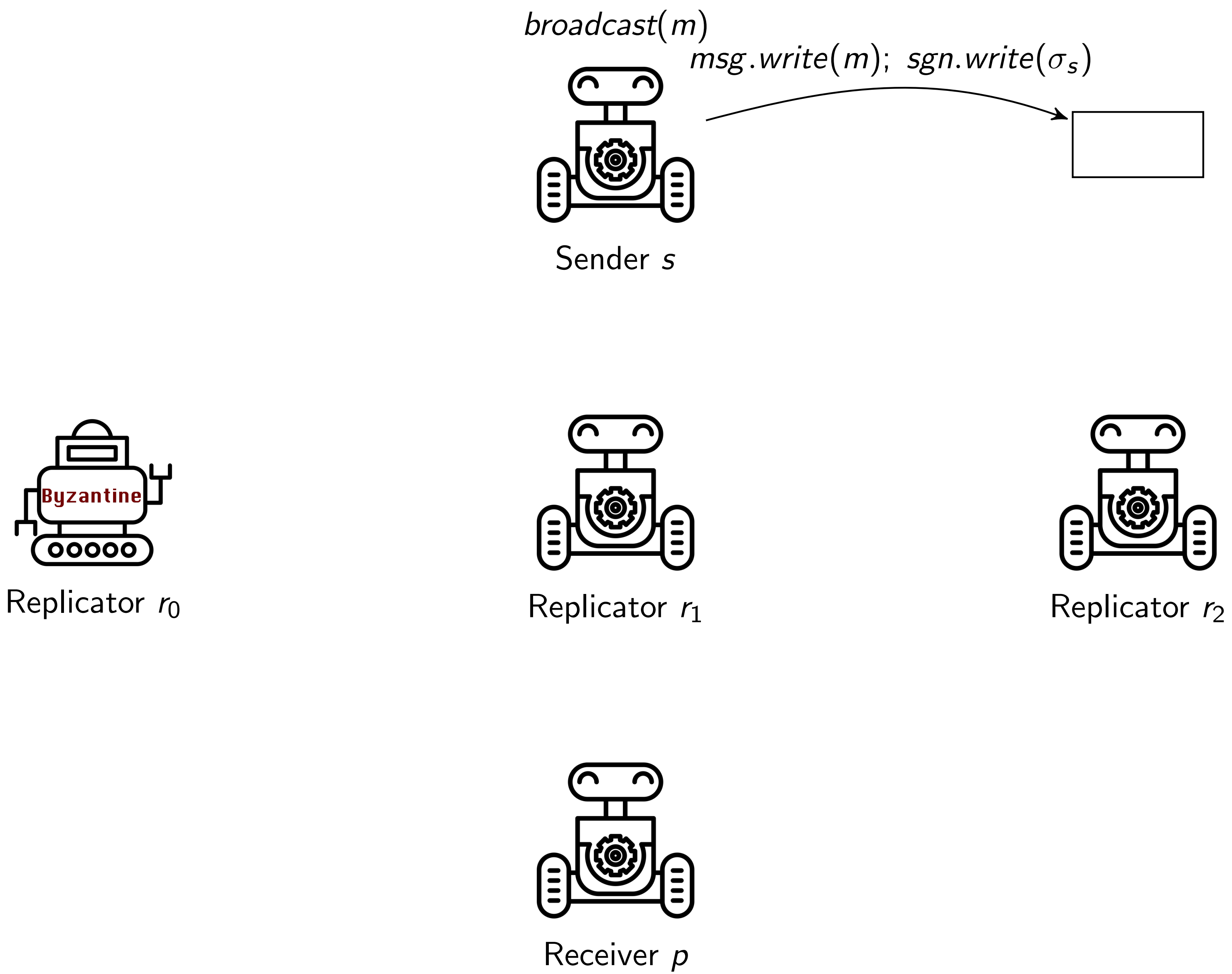
Consistent Broadcast

Algorithm sketch, $f = 1$



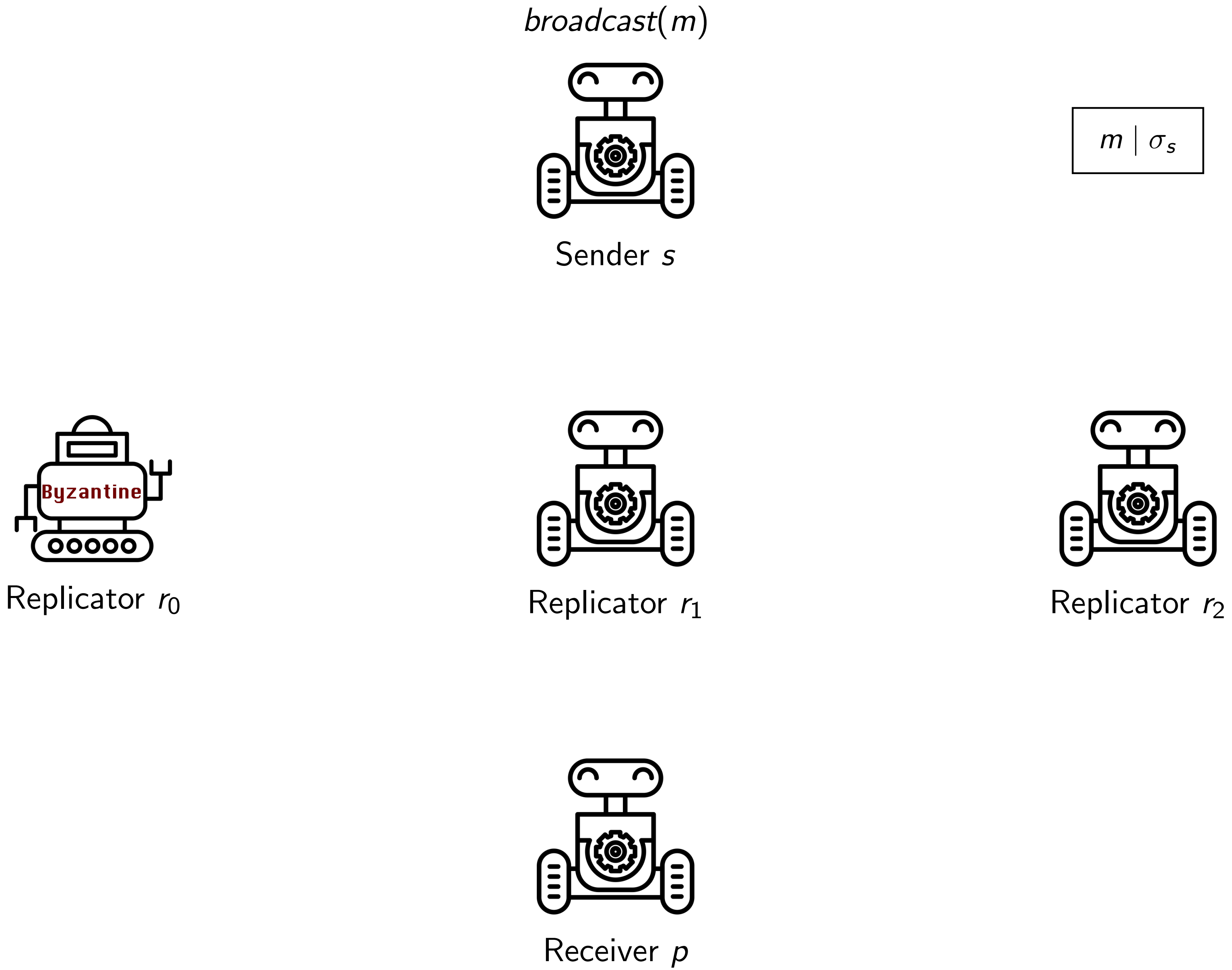
Consistent Broadcast

Algorithm sketch, $f = 1$



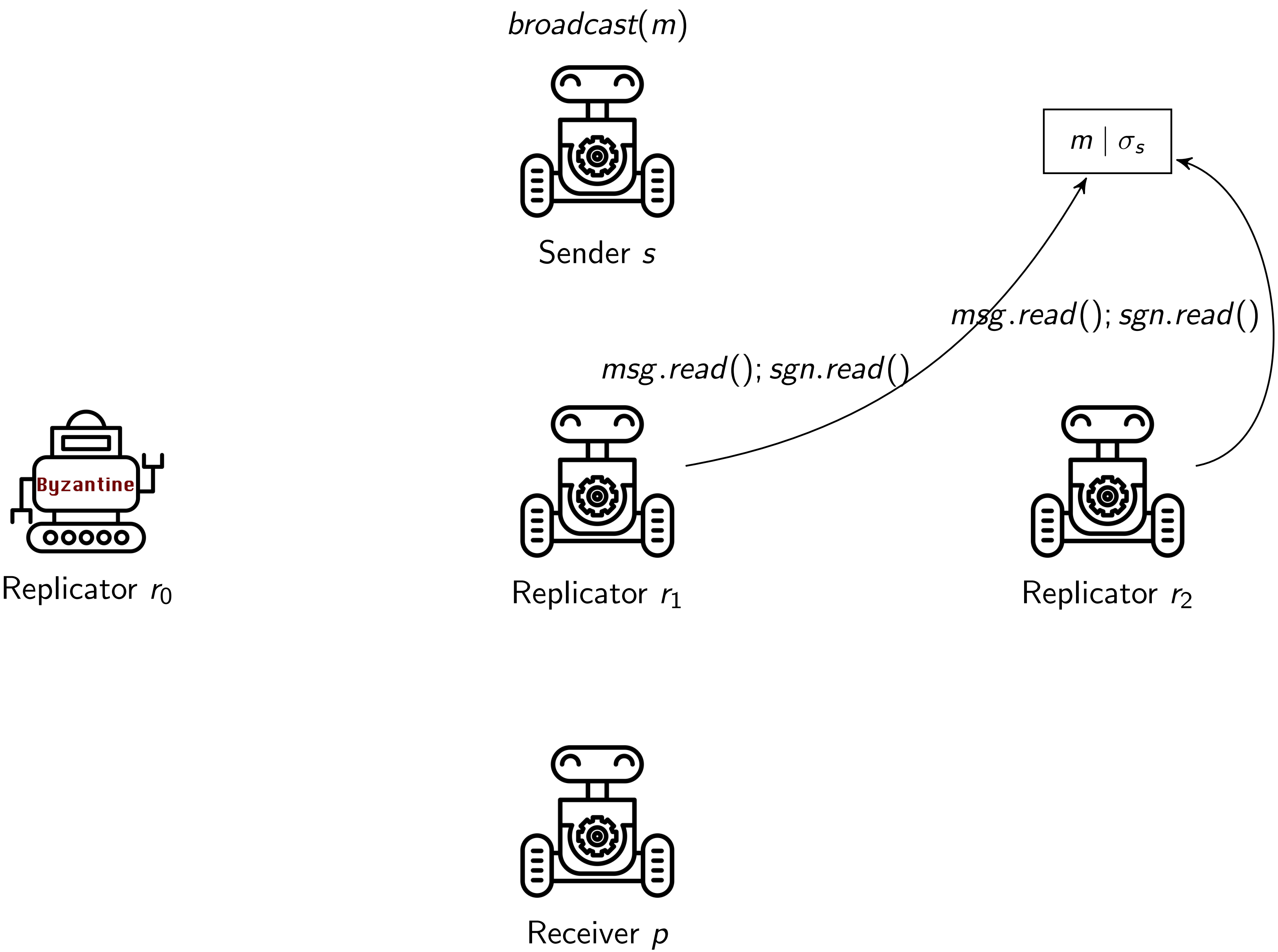
Consistent Broadcast

Algorithm sketch, $f = 1$



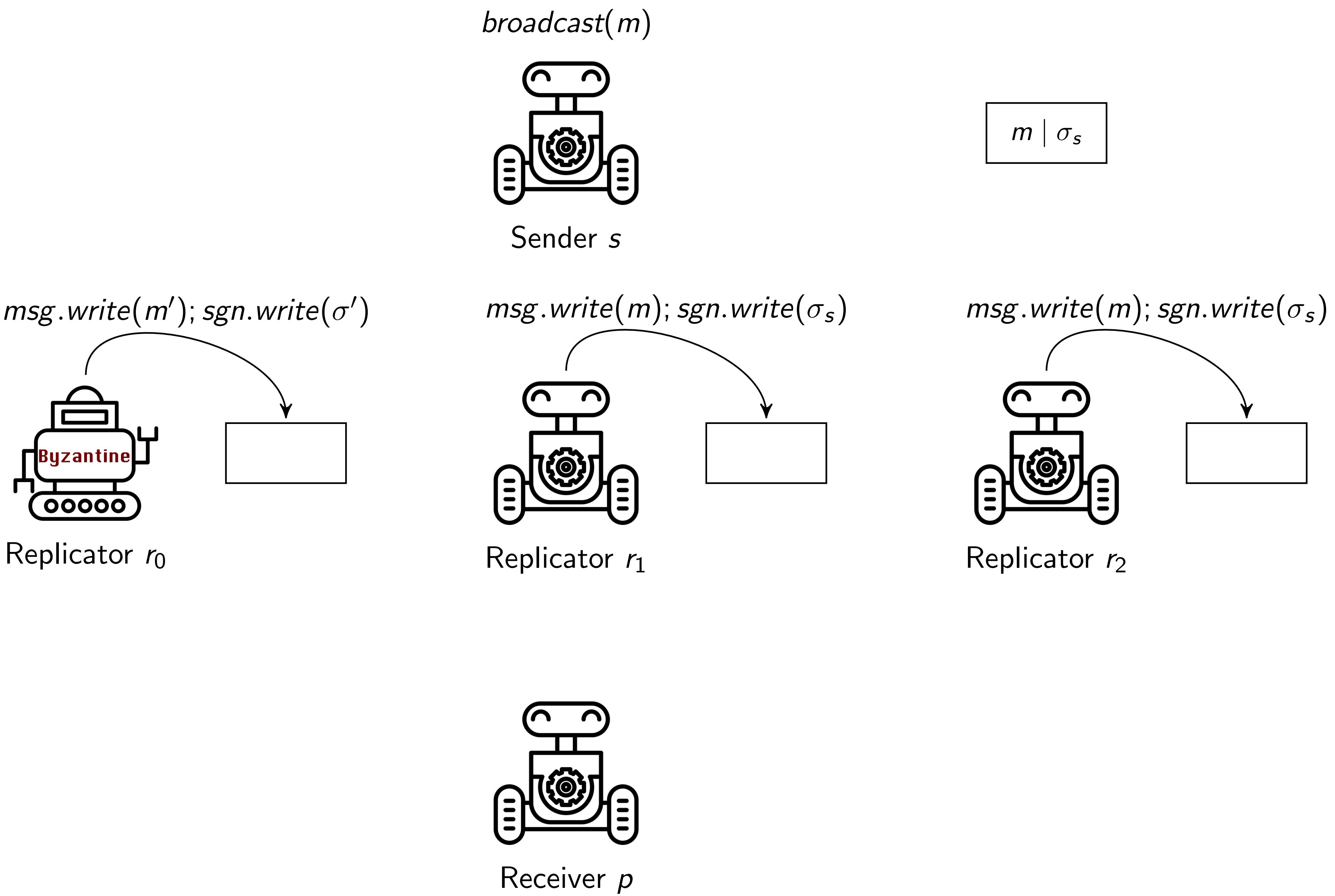
Consistent Broadcast

Algorithm sketch, $f = 1$



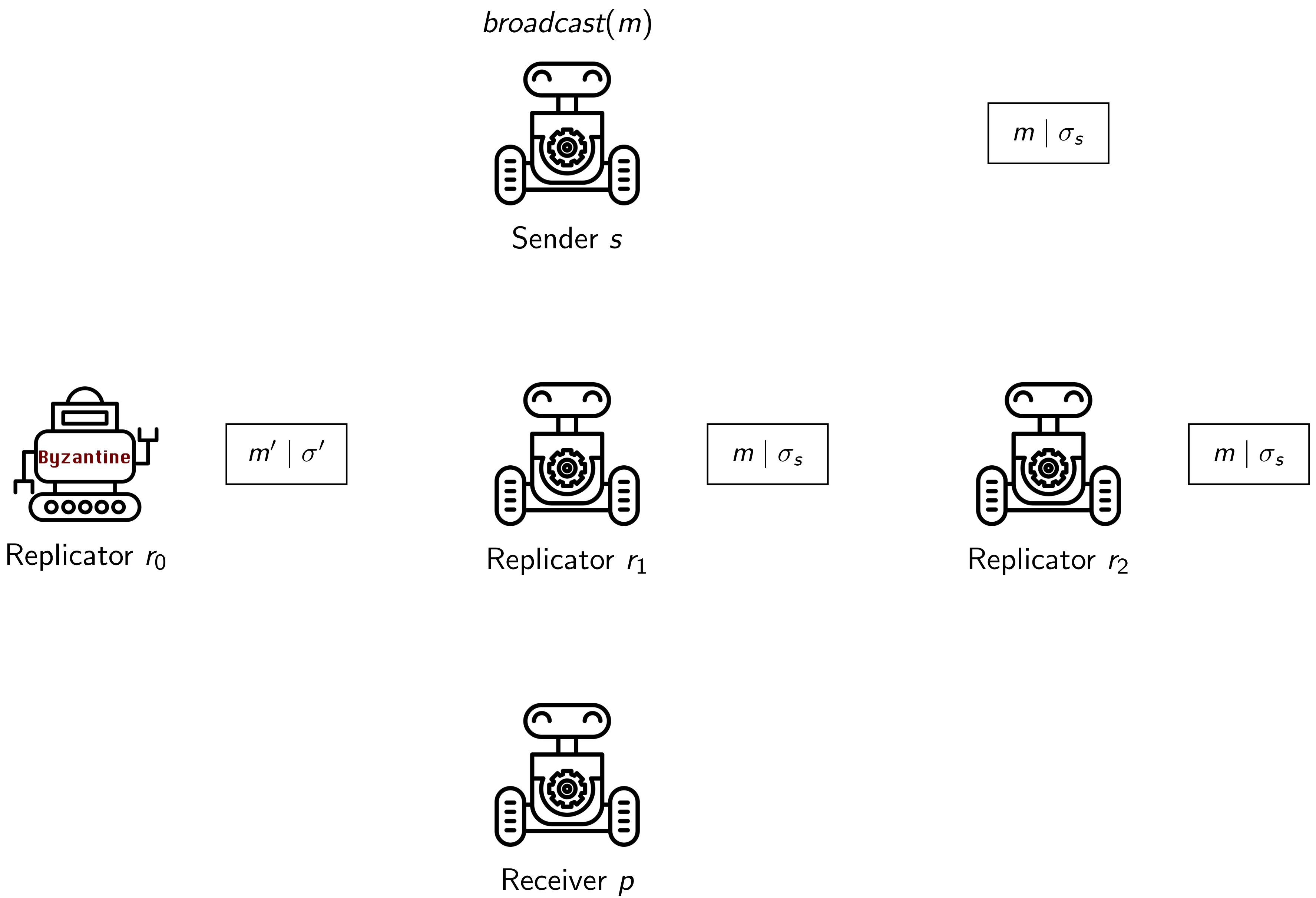
Consistent Broadcast

Algorithm sketch, $f = 1$



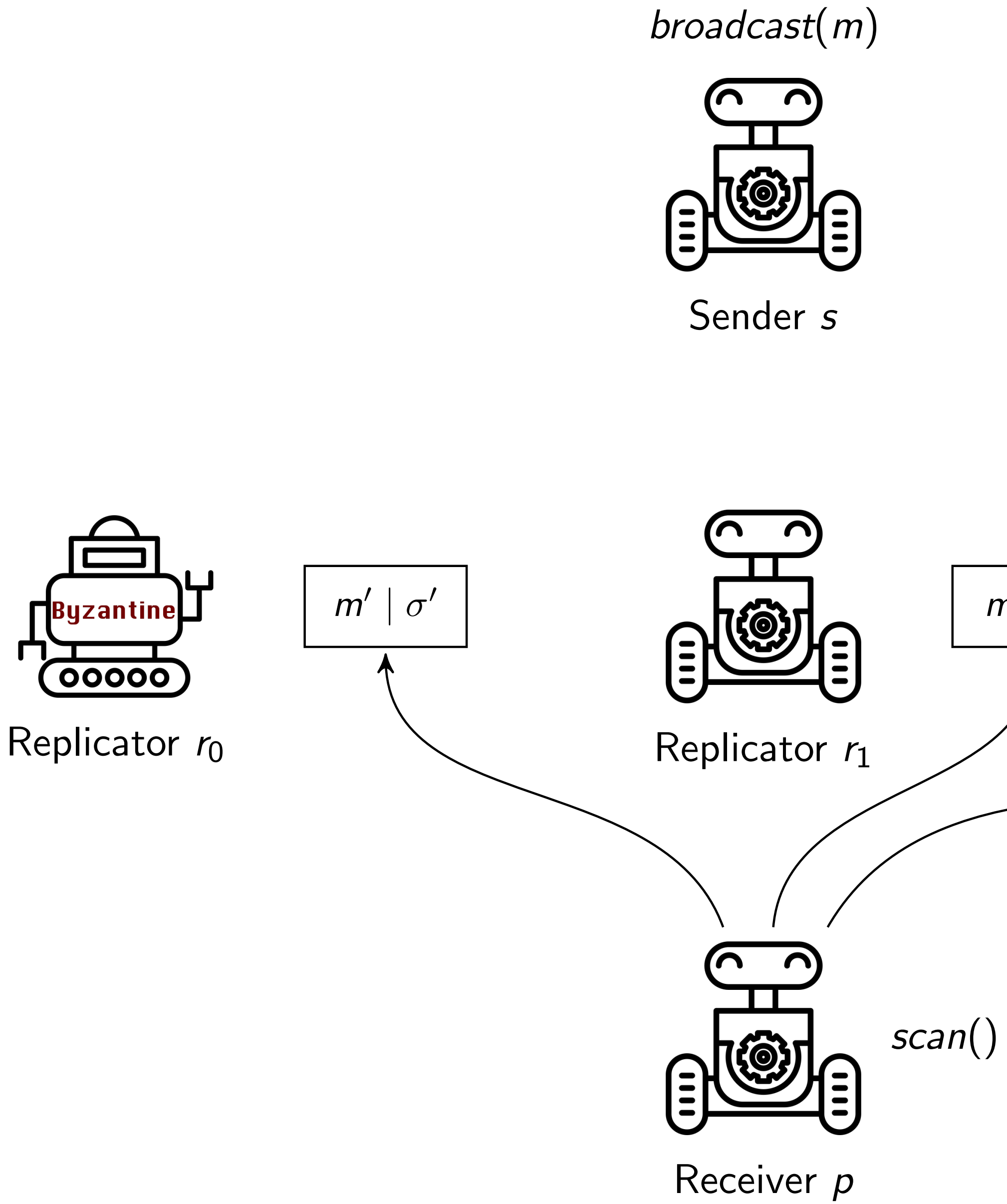
Consistent Broadcast

Algorithm sketch, $f = 1$



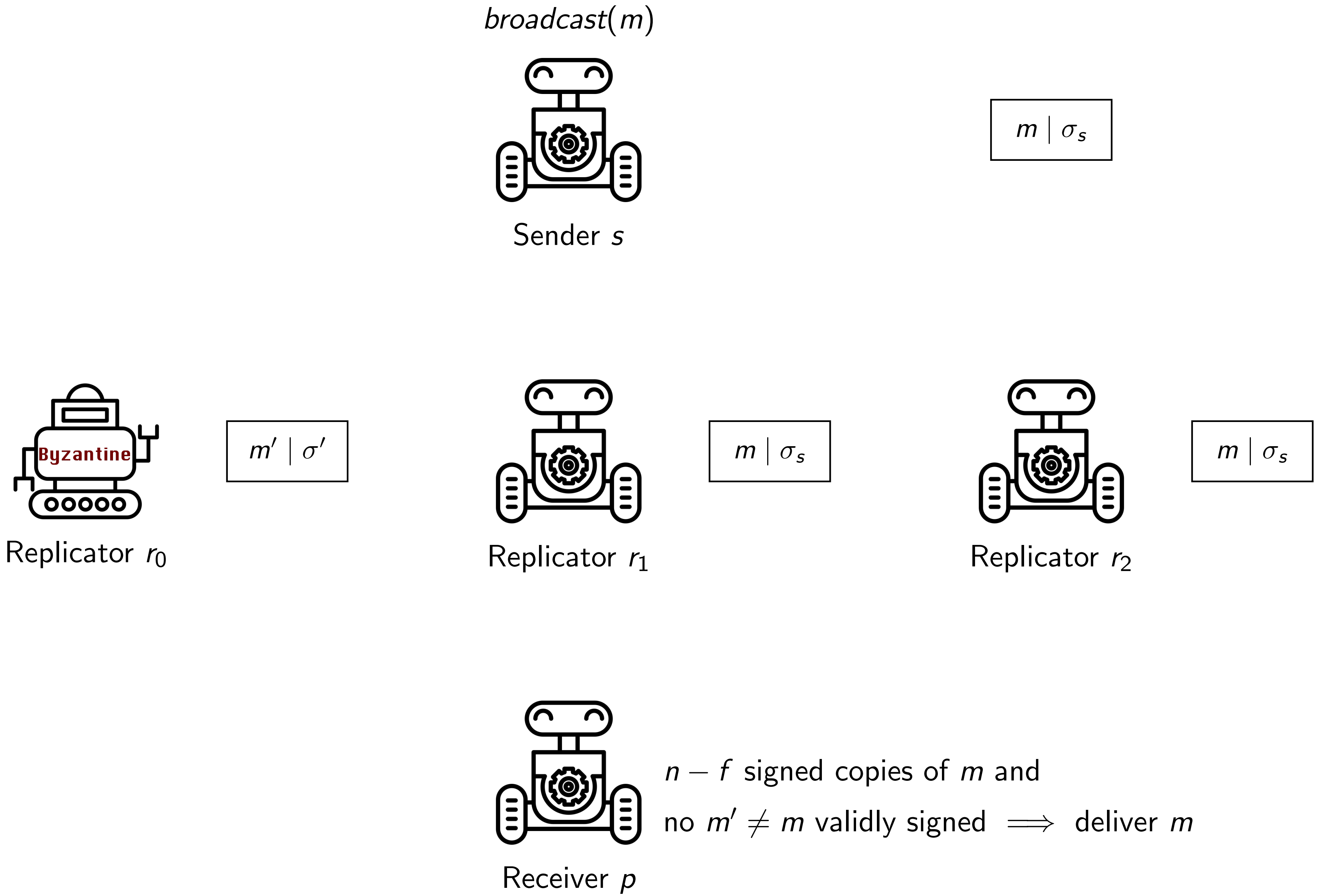
Consistent Broadcast

Algorithm sketch, $f = 1$



Consistent Broadcast

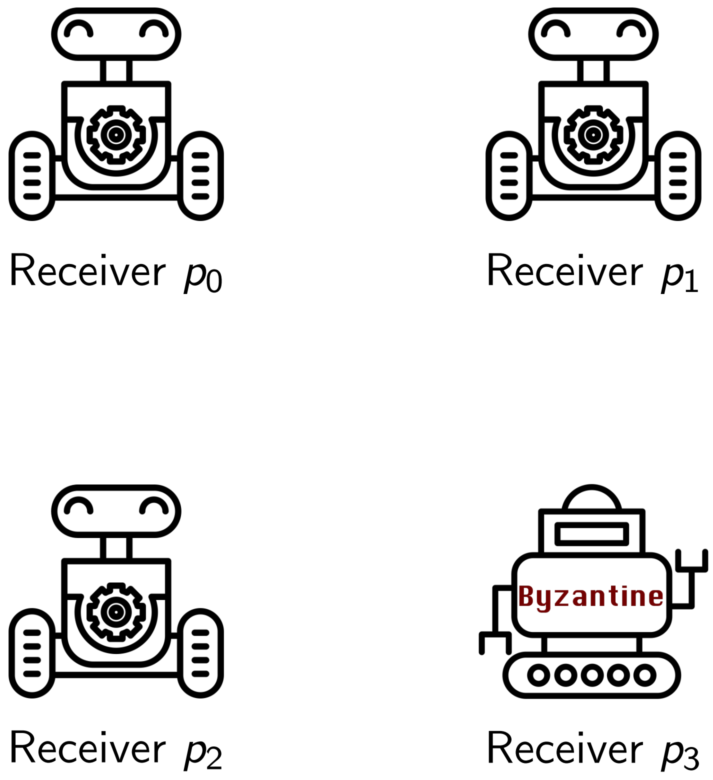
Algorithm sketch, $f = 1$



Reliable Broadcast

Same properties as Consistent Broadcast + Totality

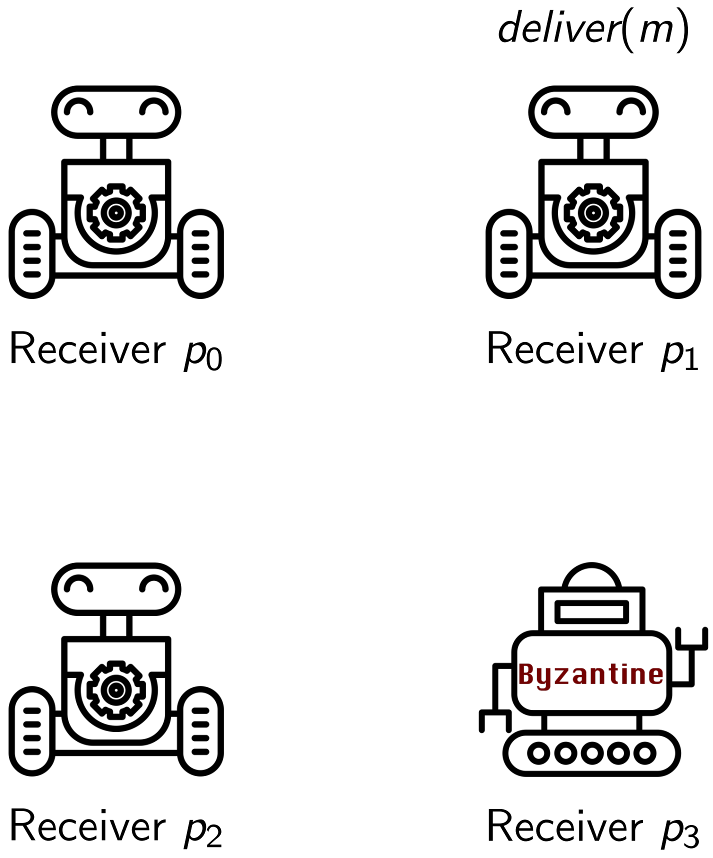
If some correct process delivers m, then every correct process eventually delivers a message



Reliable Broadcast

Same properties as Consistent Broadcast + Totality

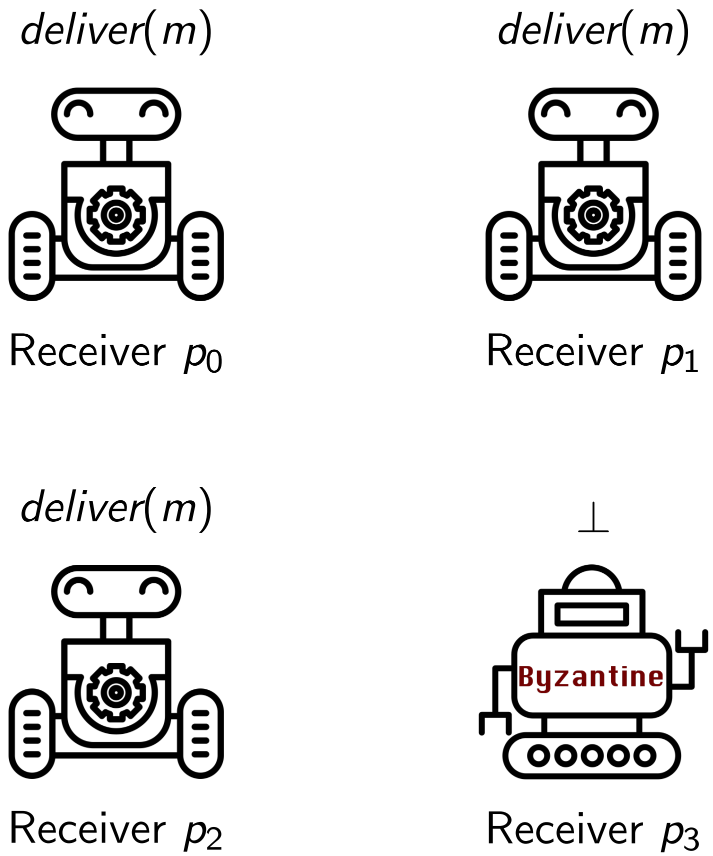
If some correct process delivers m, then every correct process eventually delivers a message



Reliable Broadcast

Same properties as Consistent Broadcast + Totality

If some correct process delivers m, then every correct process eventually delivers a message



Reliable Broadcast

Validity - If a correct process s broadcasts m, then every correct process eventually delivers m

Consistency - If p and p' are correct processes, p delivers m, and p' delivers m', then $m = m'$

Integrity - If some correct process delivers m and s is correct, then s previously broadcast m

Totality - If some correct process delivers m, then every correct process eventually delivers a message

Consistent Broadcast vs Reliable Broadcast

Consistent and Reliable Broadcast behave the same way when the sender s is correct (recall the *Validity* property)

Consistent Broadcast vs Reliable Broadcast

Consistent and Reliable Broadcast behave the same way when the sender s is correct (recall the *Validity* property)

Yet when the sender is faulty . . .

Consistent Broadcast vs Reliable Broadcast

Consistent and Reliable Broadcast behave the same way when the sender s is correct (recall the *Validity* property)

Yet when the sender is faulty ...

- Consistent Broadcast has no delivery guarantees: some correct processes may deliver a message, others may not

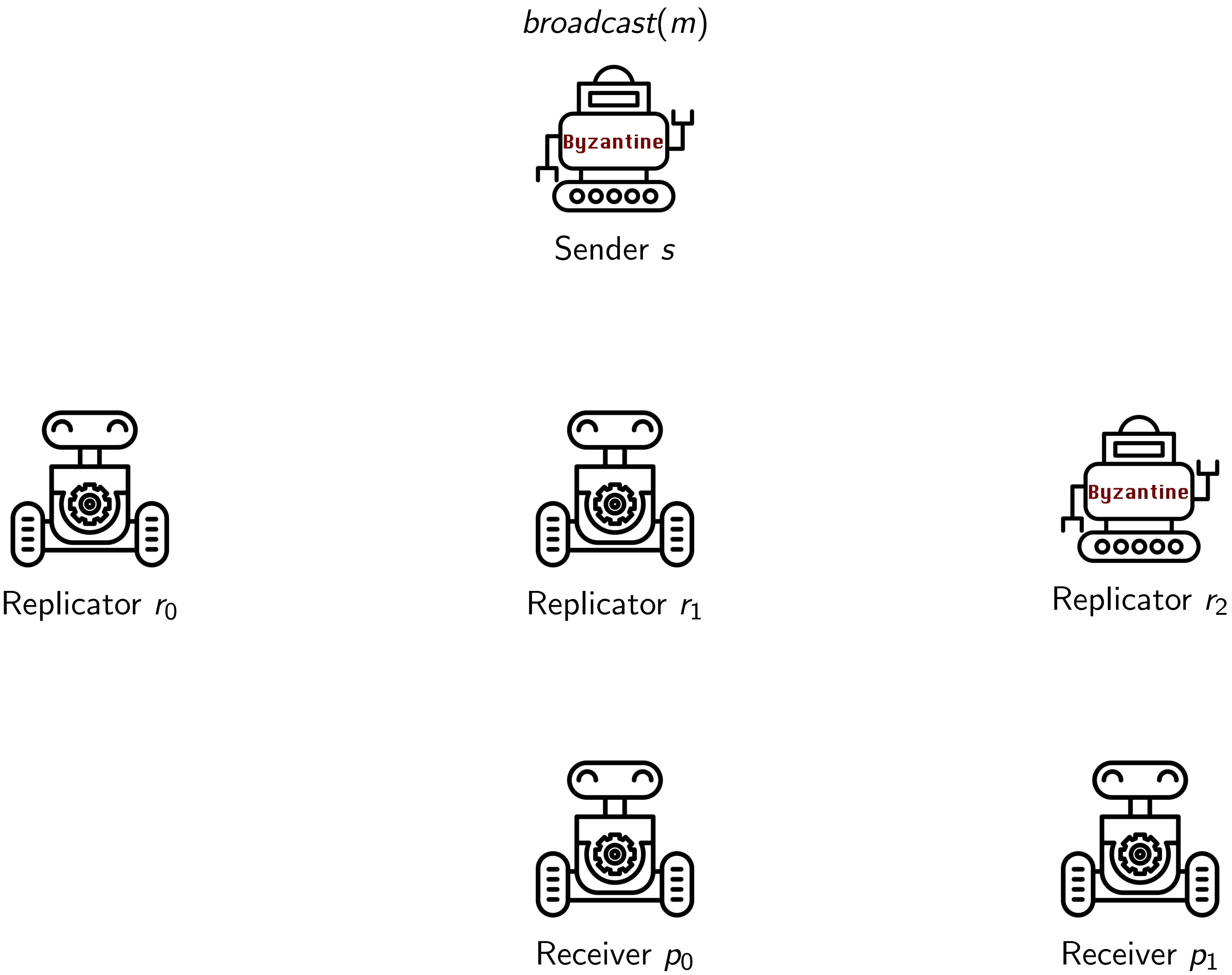
Consistent Broadcast vs Reliable Broadcast

Consistent and Reliable Broadcast behave the same way when the sender s is correct (recall the *Validity* property)

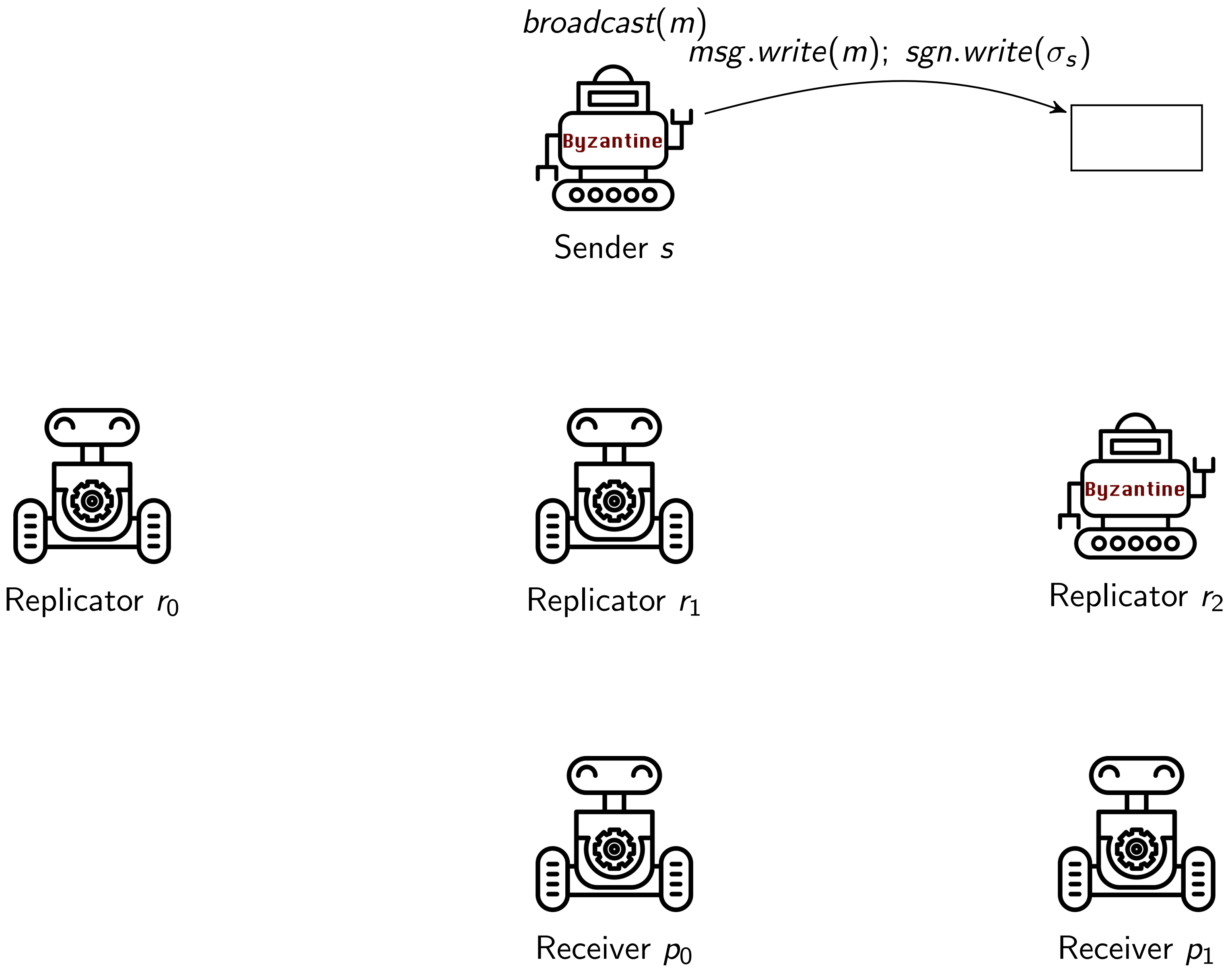
Yet when the sender is faulty . . .

- Consistent Broadcast has no delivery guarantees: some correct processes may deliver a message, others may not
- while Reliable Broadcast guarantees every correct process eventually delivers a message as soon as one correct process delivered

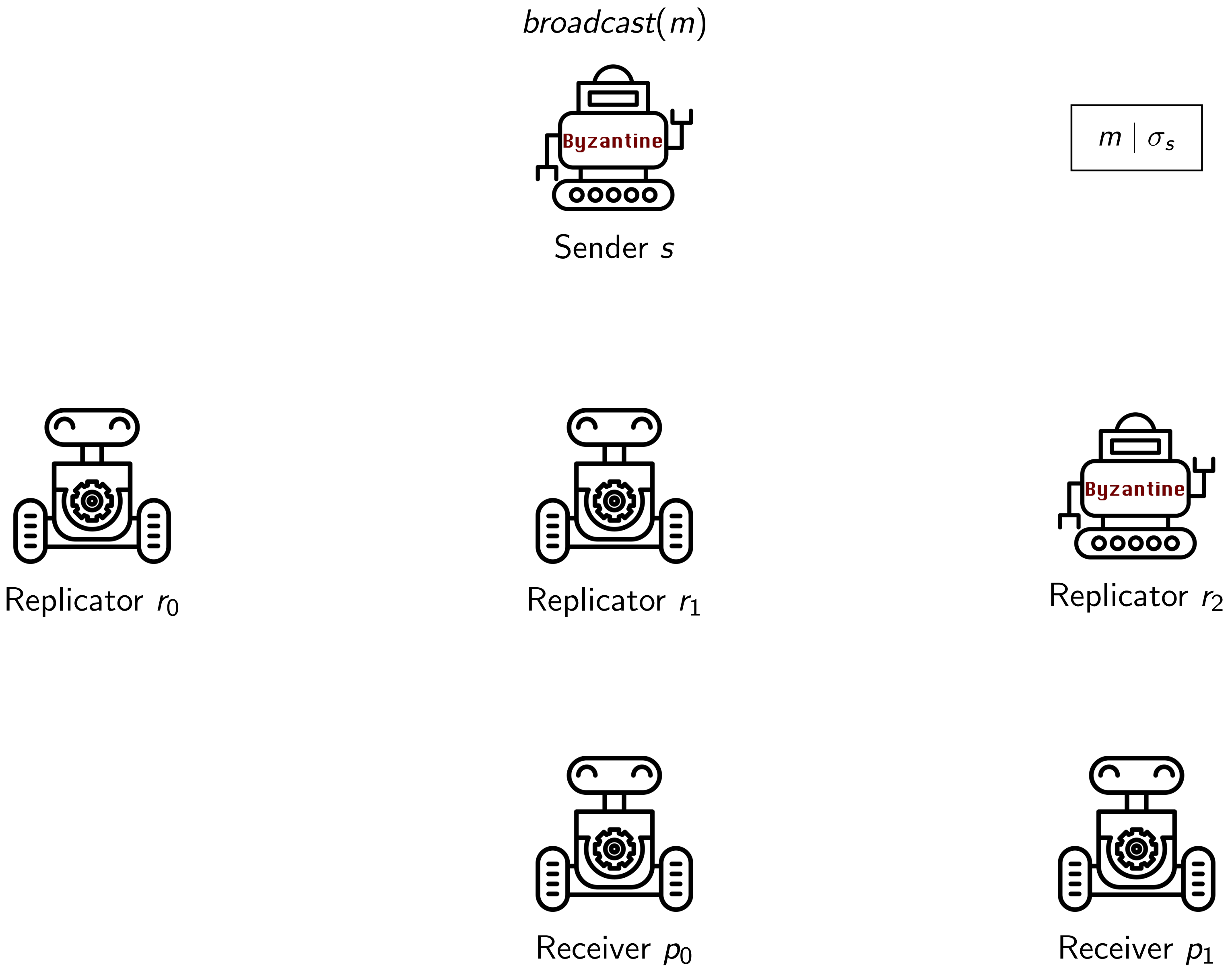
Consistent Broadcast algorithm $= \neg$ sufficient



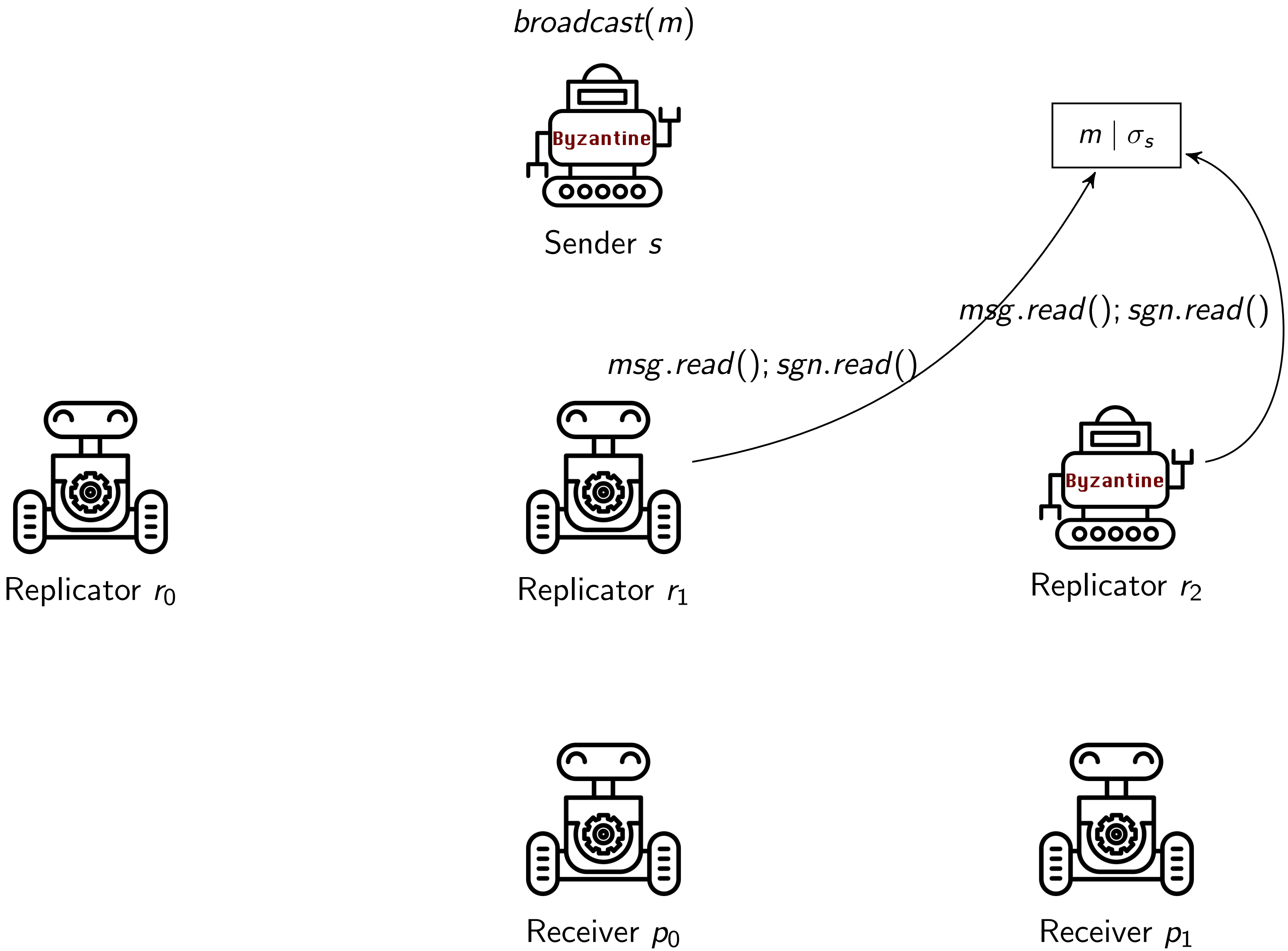
Consistent Broadcast algorithm $= \neg$ sufficient



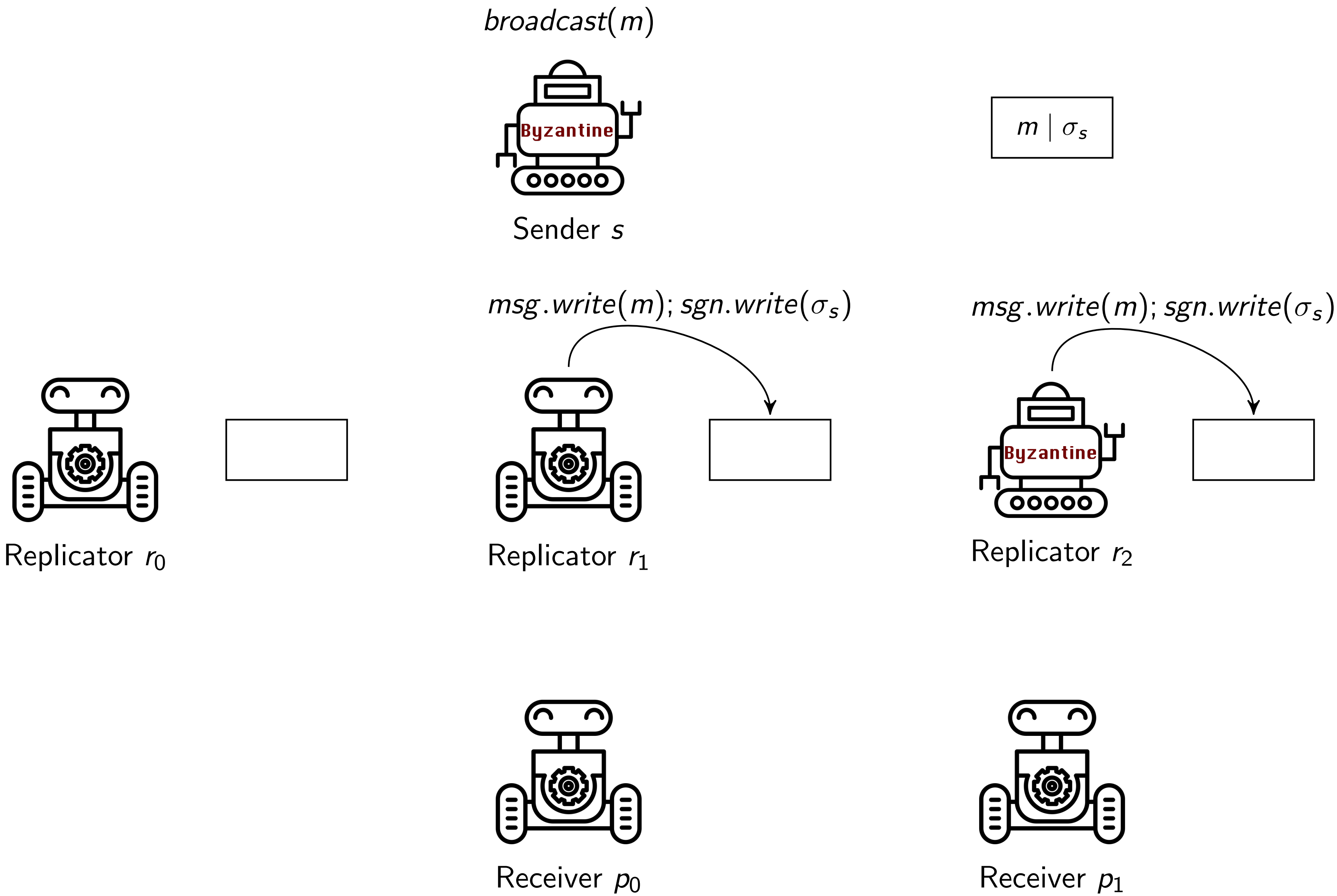
Consistent Broadcast algorithm $= \neg$ sufficient



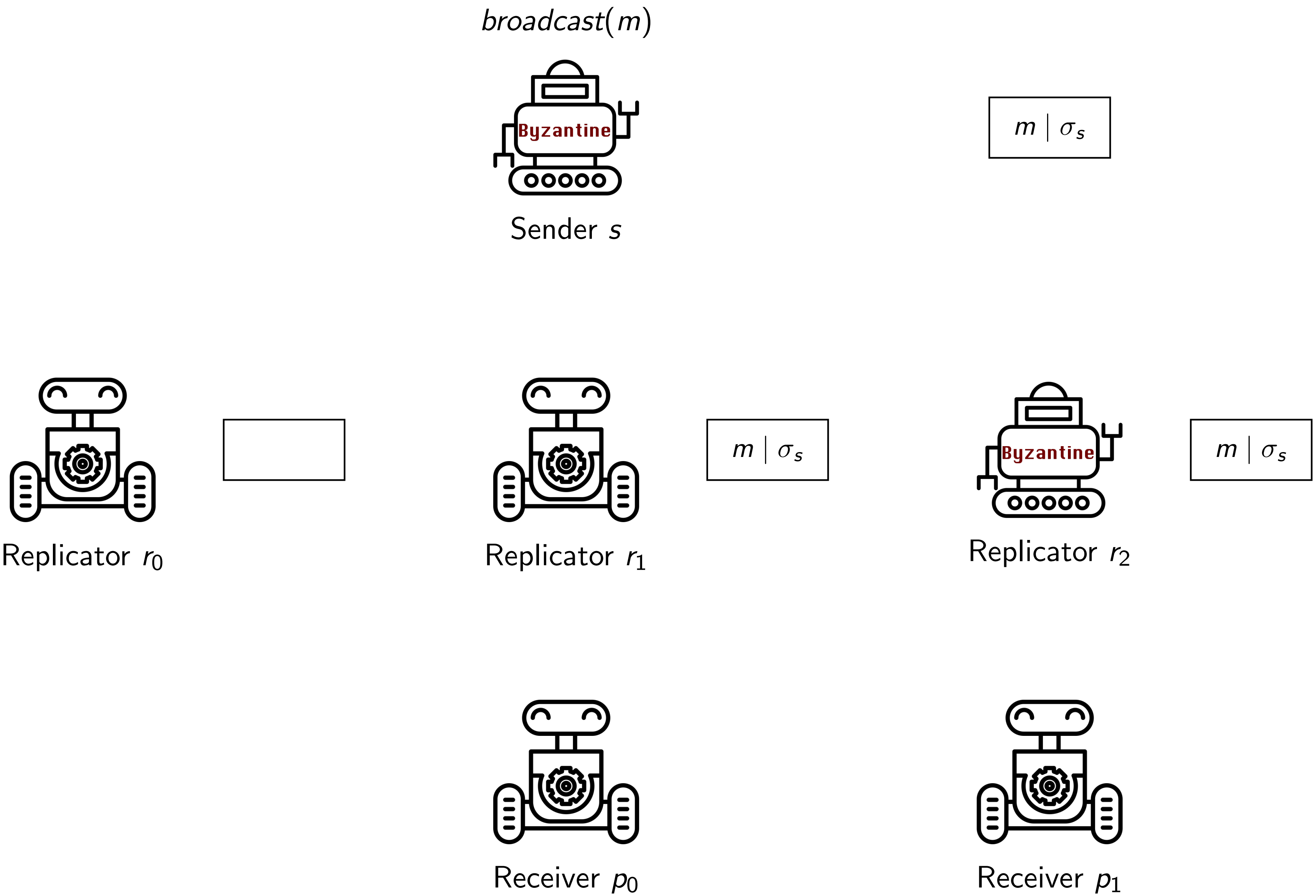
Consistent Broadcast algorithm $= \neg$ sufficient



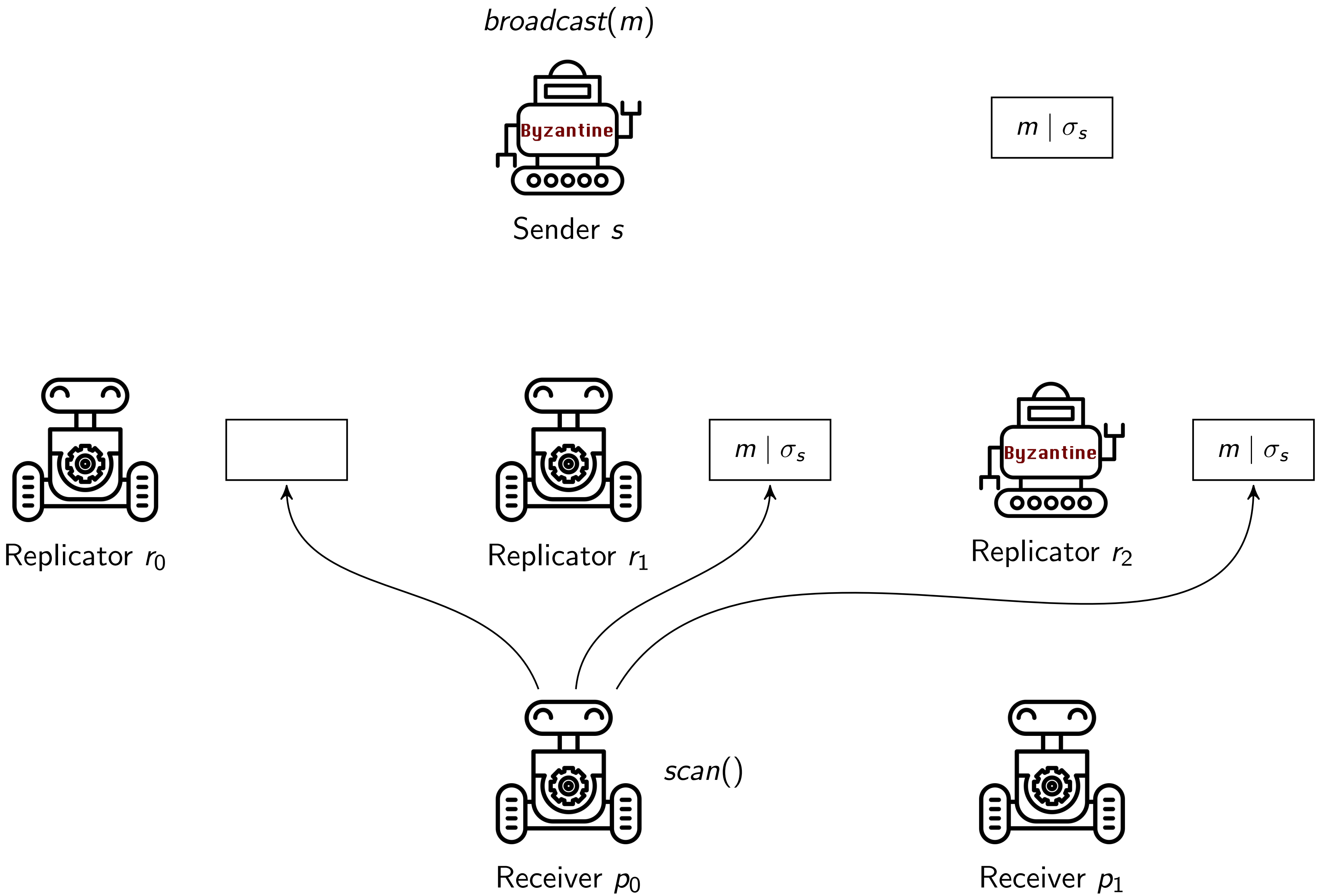
Consistent Broadcast algorithm $= \neg$ sufficient



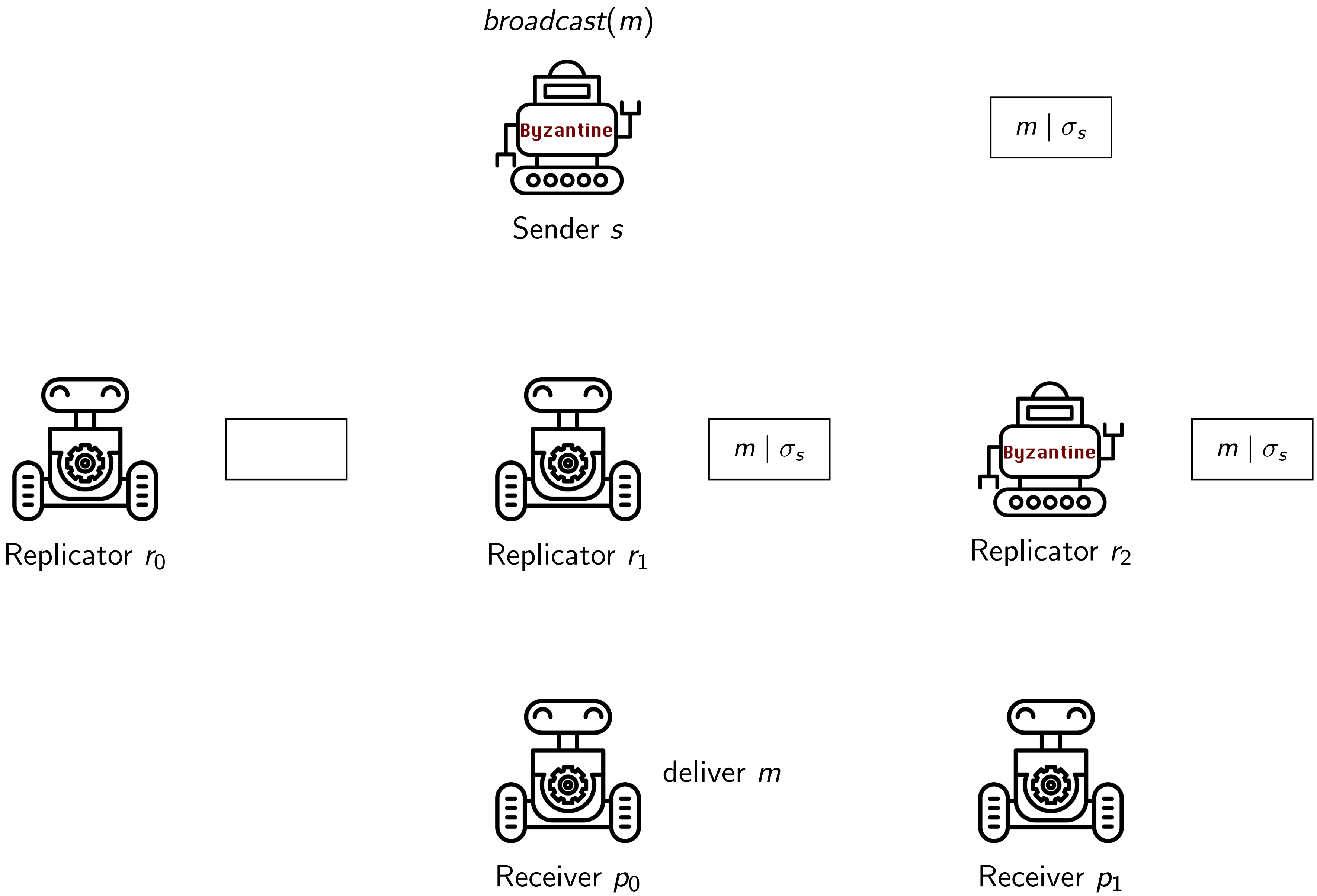
Consistent Broadcast algorithm $= \neg$ sufficient



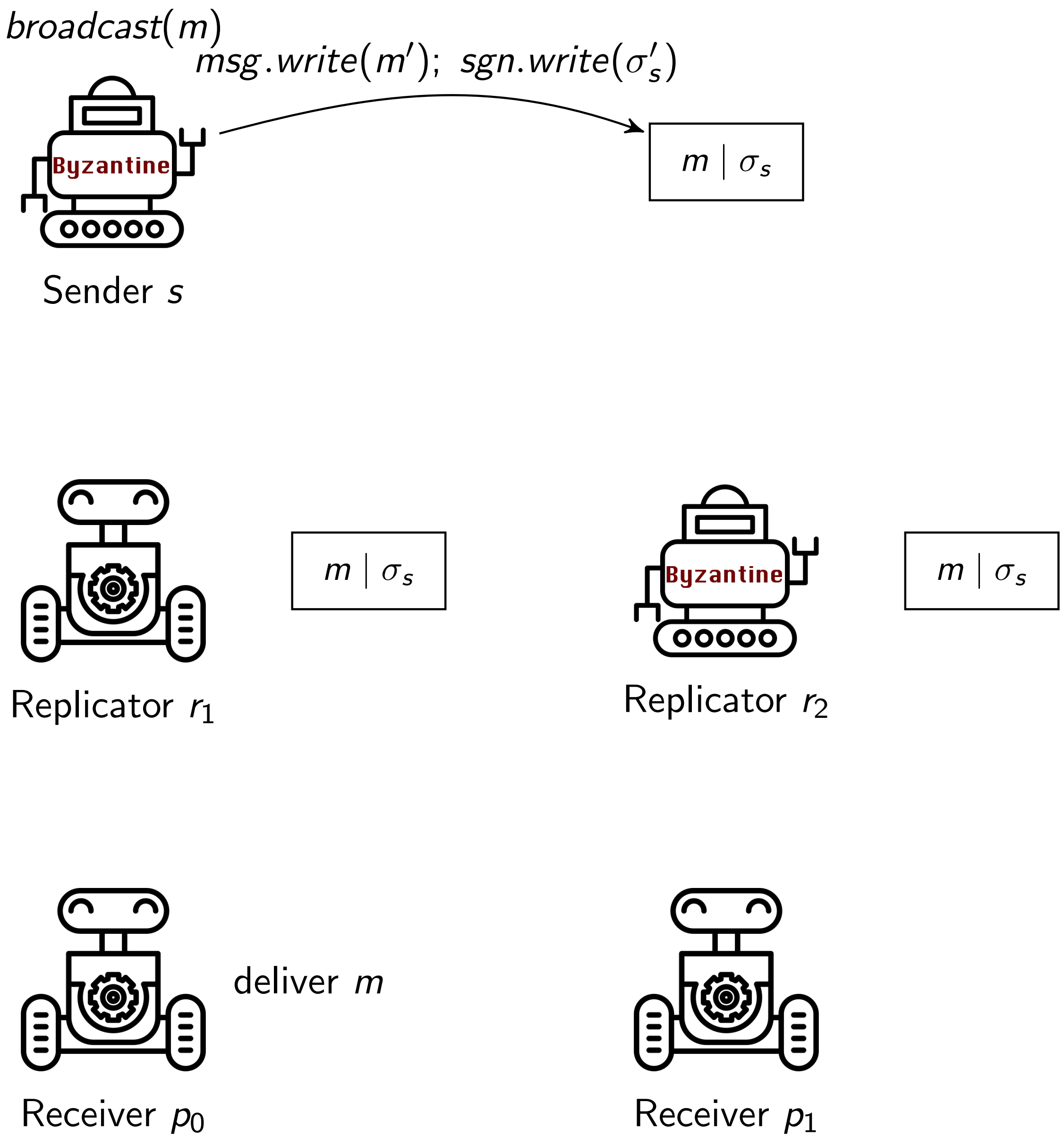
Consistent Broadcast algorithm = ¬ sufficient



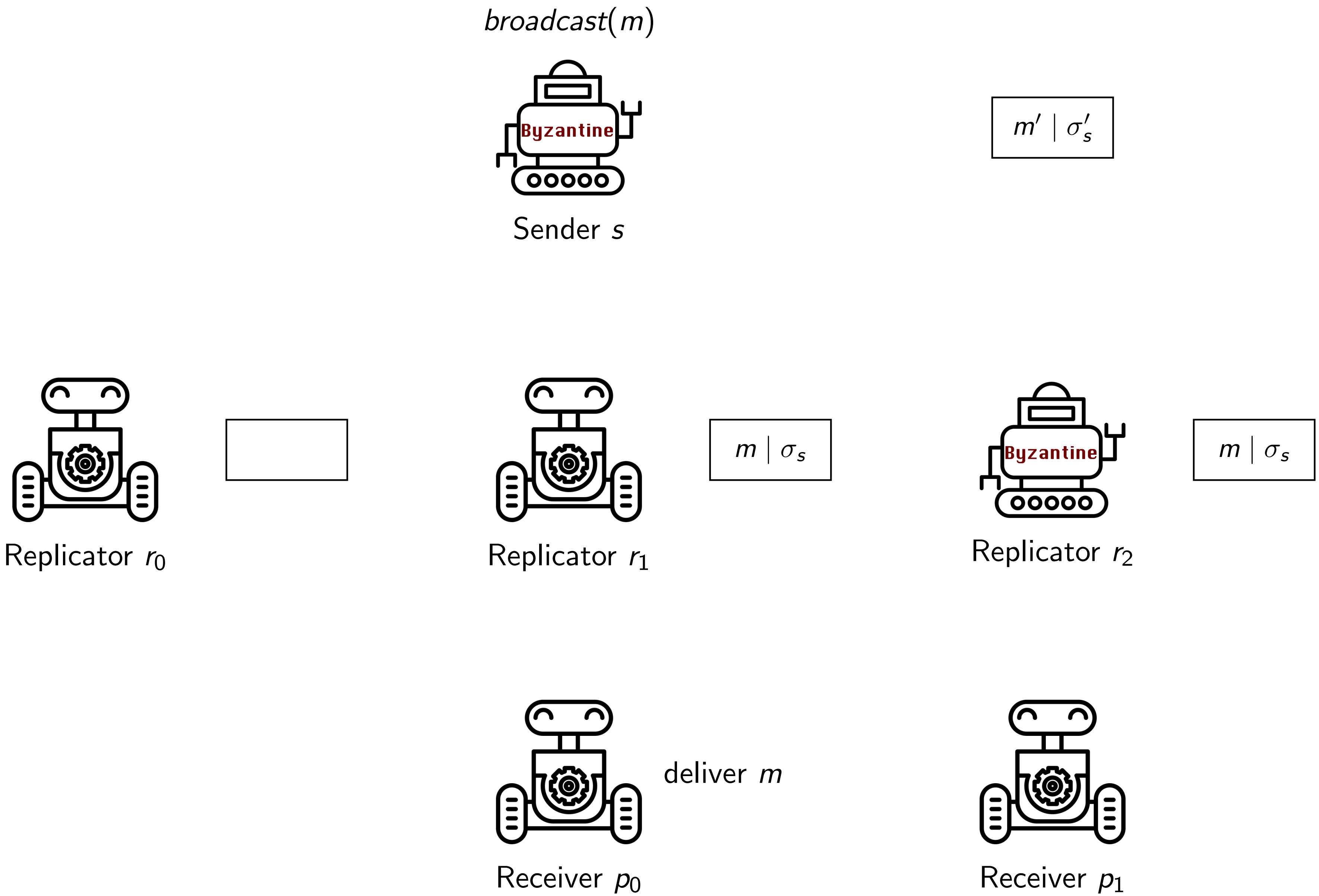
Consistent Broadcast algorithm = ¬ sufficient



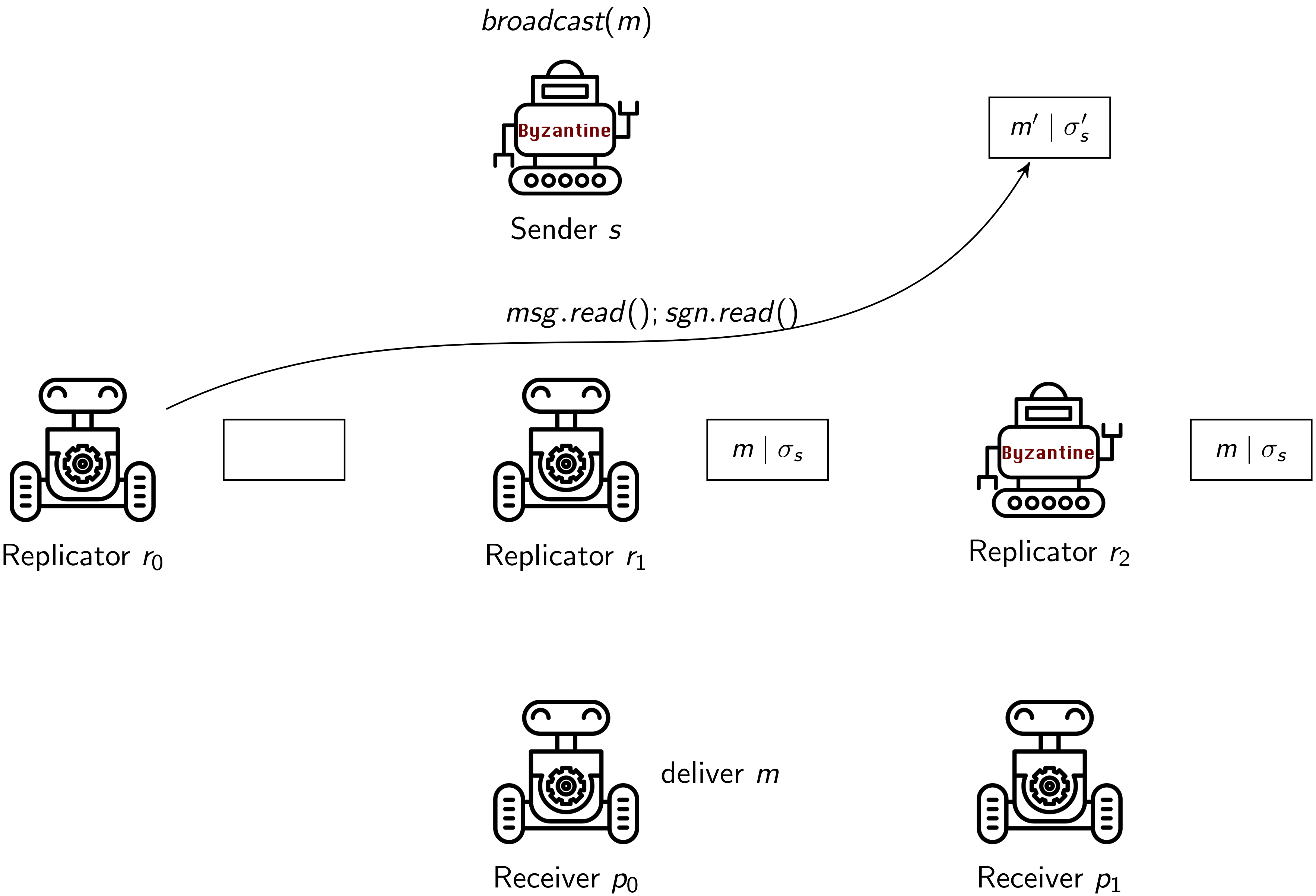
Consistent Broadcast algorithm $= \neg$ sufficient



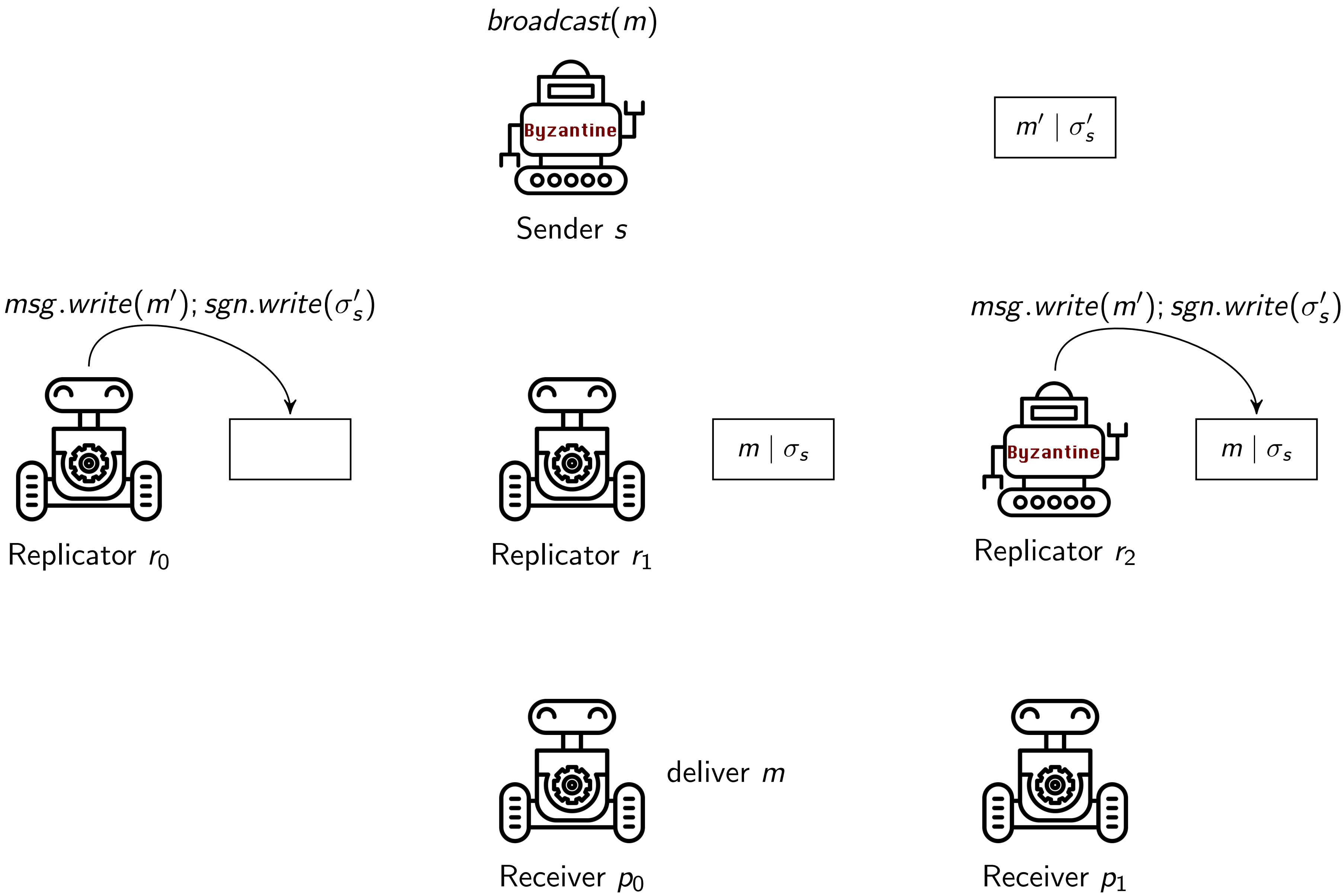
Consistent Broadcast algorithm $= \neg$ sufficient



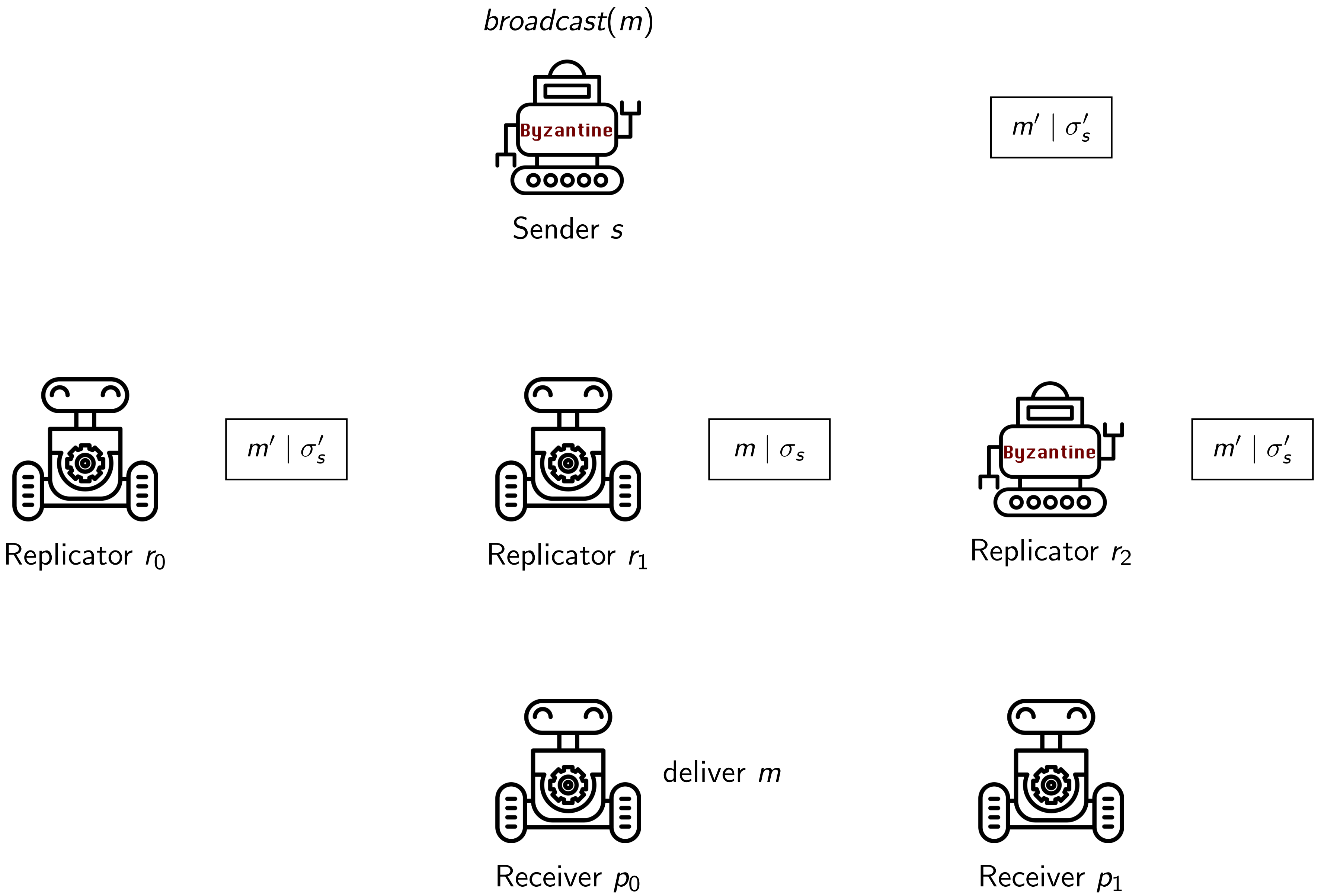
Consistent Broadcast algorithm $= \neg$ sufficient



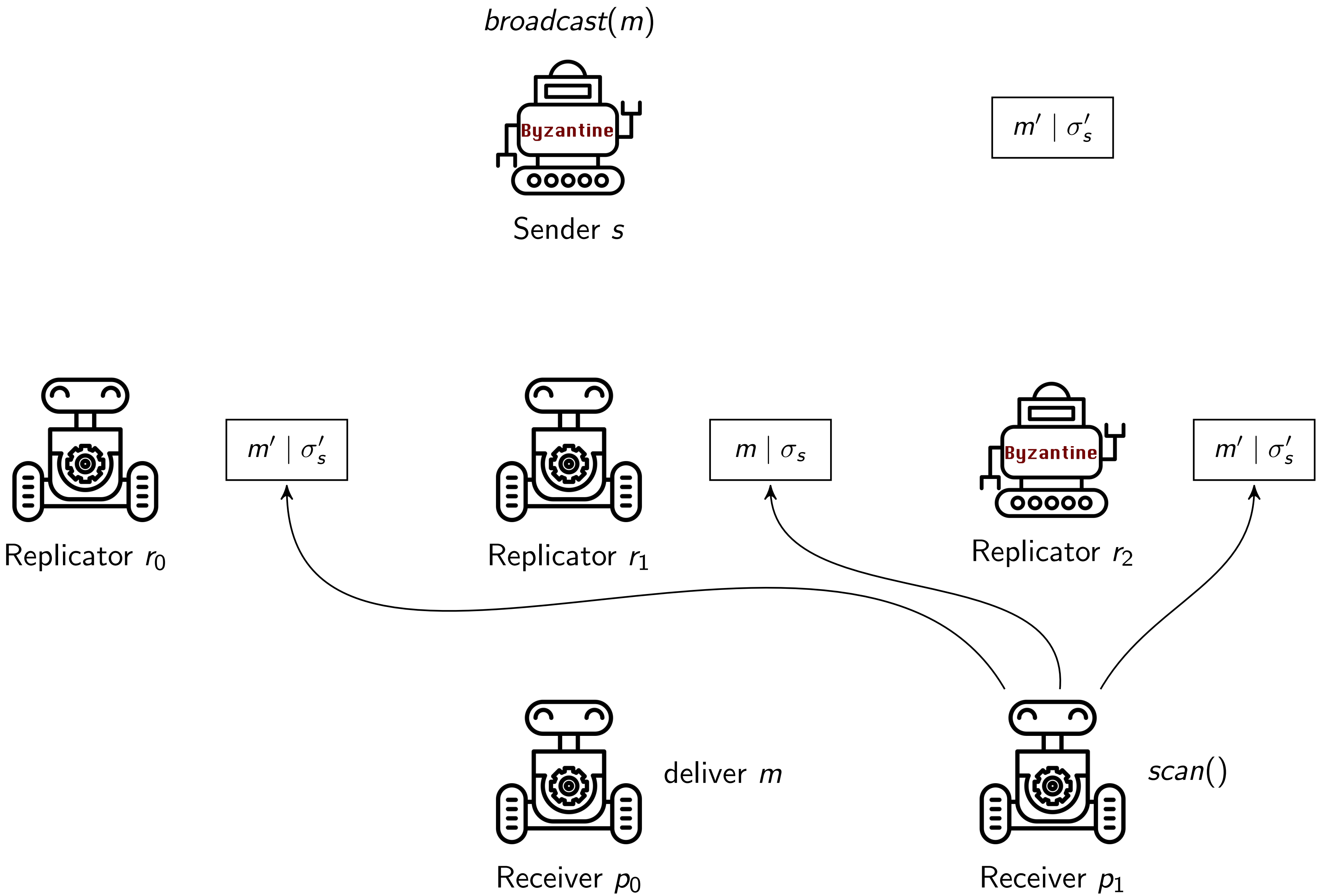
Consistent Broadcast algorithm $= \neg$ sufficient



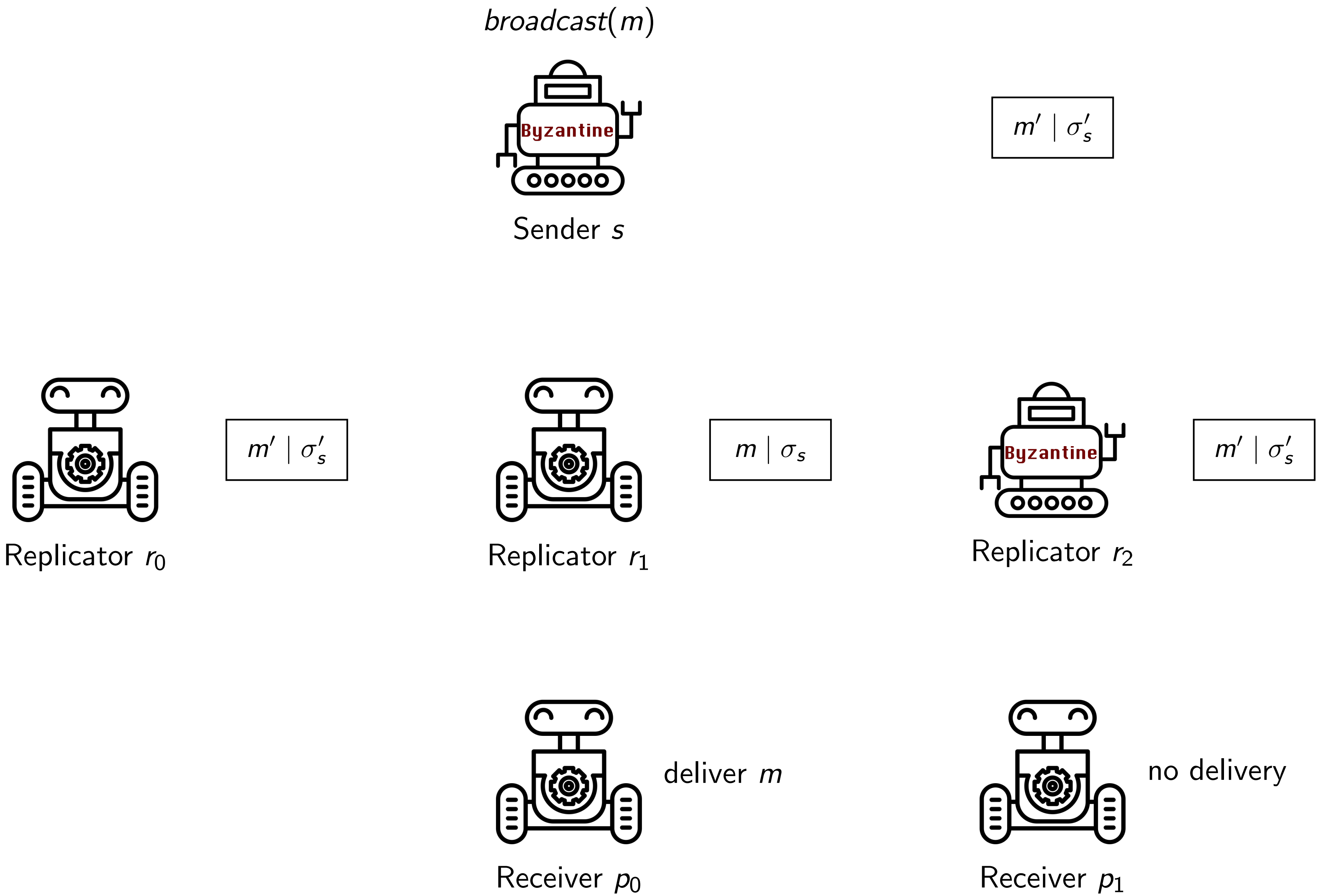
Consistent Broadcast algorithm = ¬ sufficient



Consistent Broadcast algorithm $= \neg$ sufficient



Consistent Broadcast algorithm $= \neg$ sufficient



Reliable Broadcast

Algorithm details

Init - Echo - Ready mechanism

Reliable Broadcast

Algorithm details

Init - Echo - Ready mechanism

Uses Consistent Broadcast

Reliable Broadcast

Algorithm details

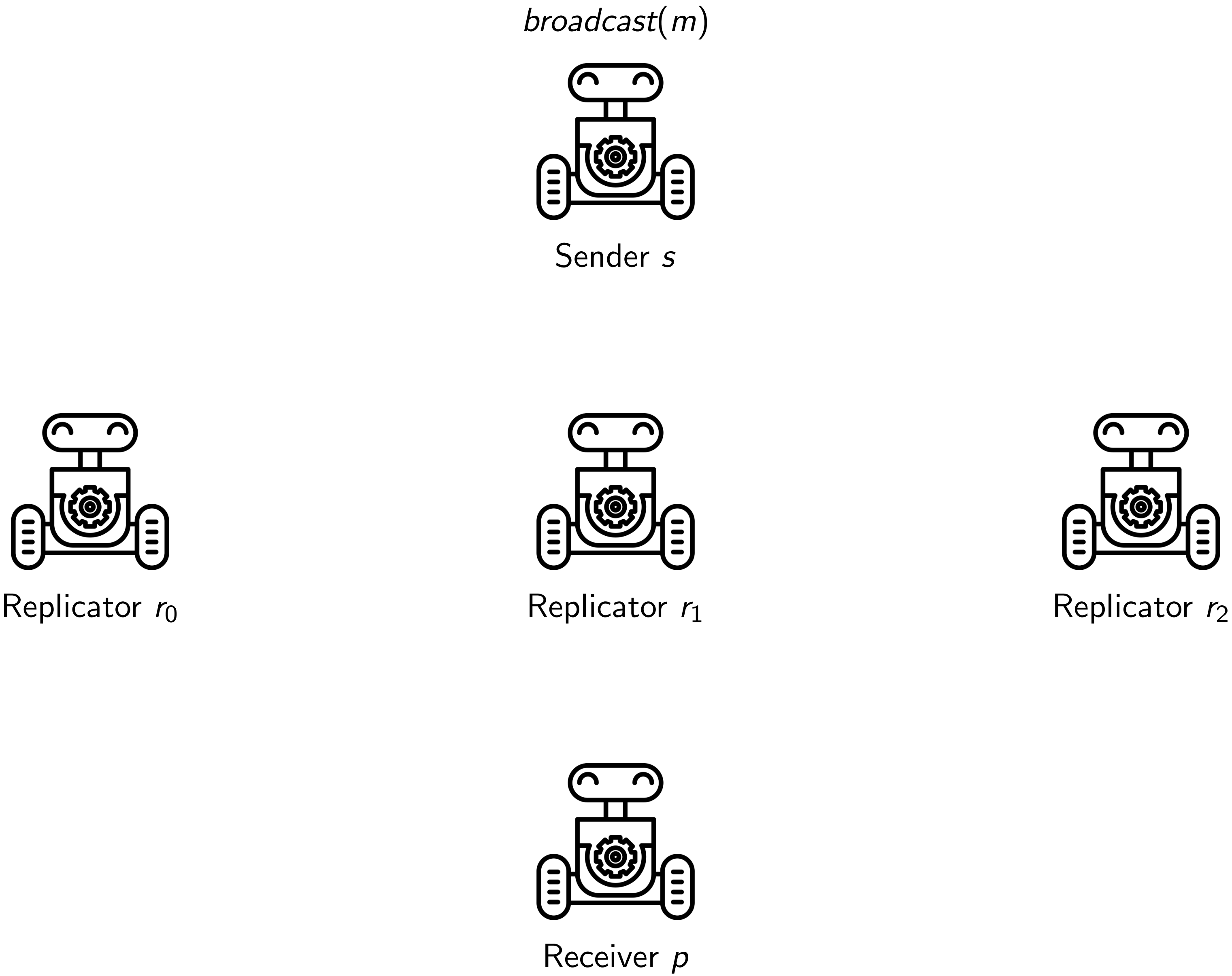
Init - Echo - Ready mechanism

Uses Consistent Broadcast

Similar delivery strategy to Consistent Broadcast: fast path, i.e., when there is unanimity and otherwise when $\exists\ n - f$ valid proof sets for m

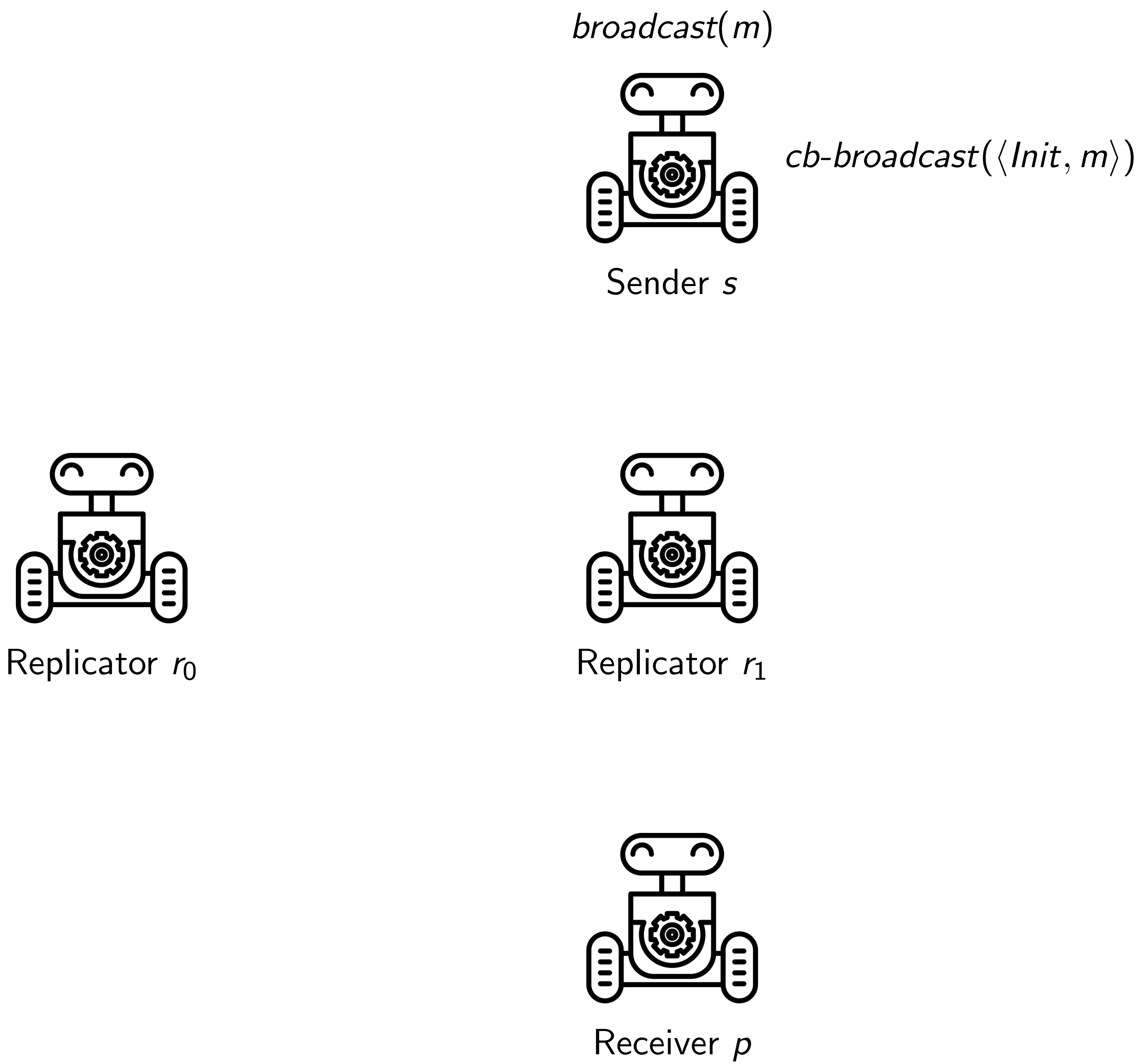
Reliable Broadcast

Algorithm sketch, $f = 1$. Fast path



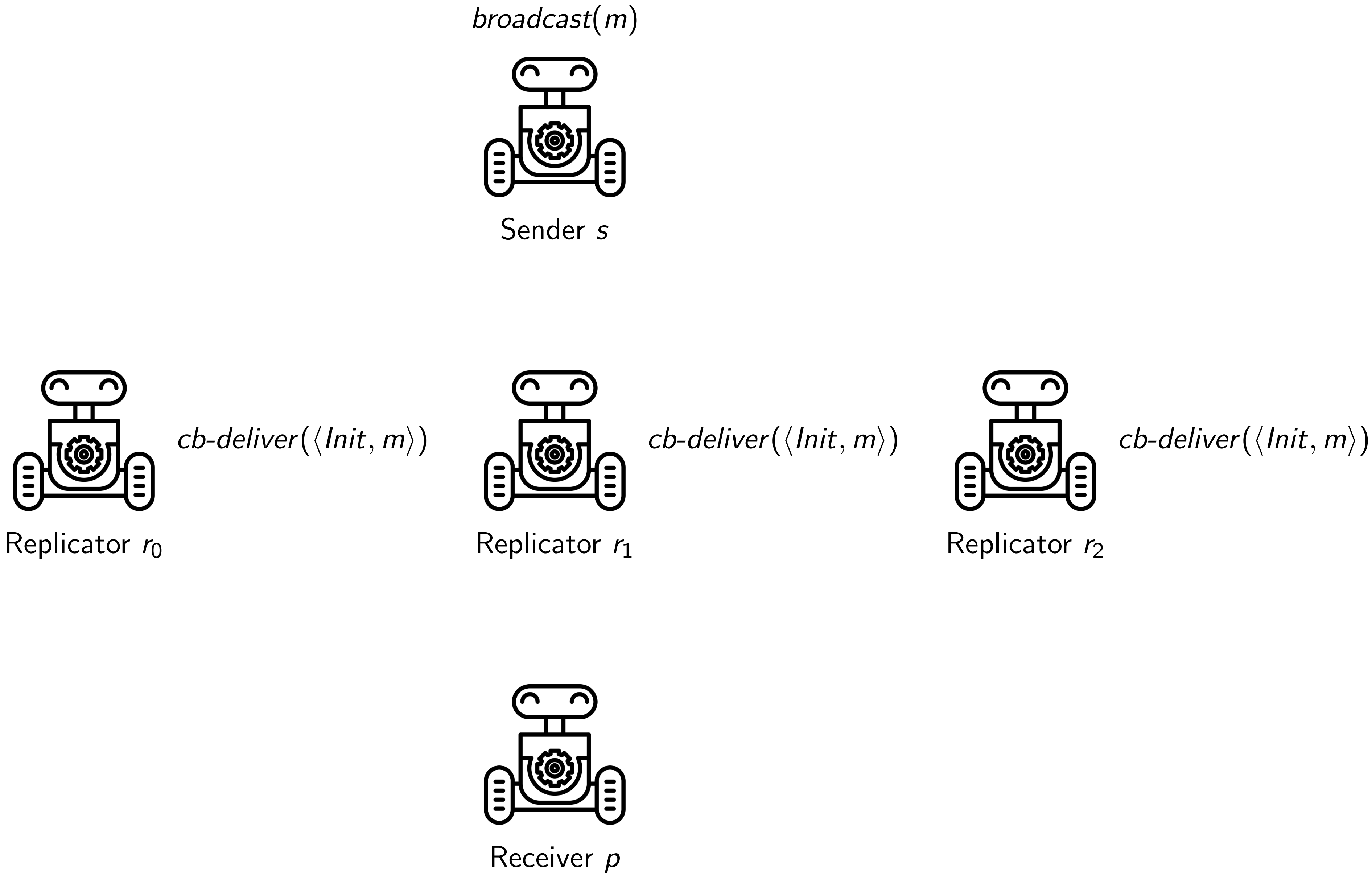
Reliable Broadcast

Algorithm sketch, $f = 1$. Fast path



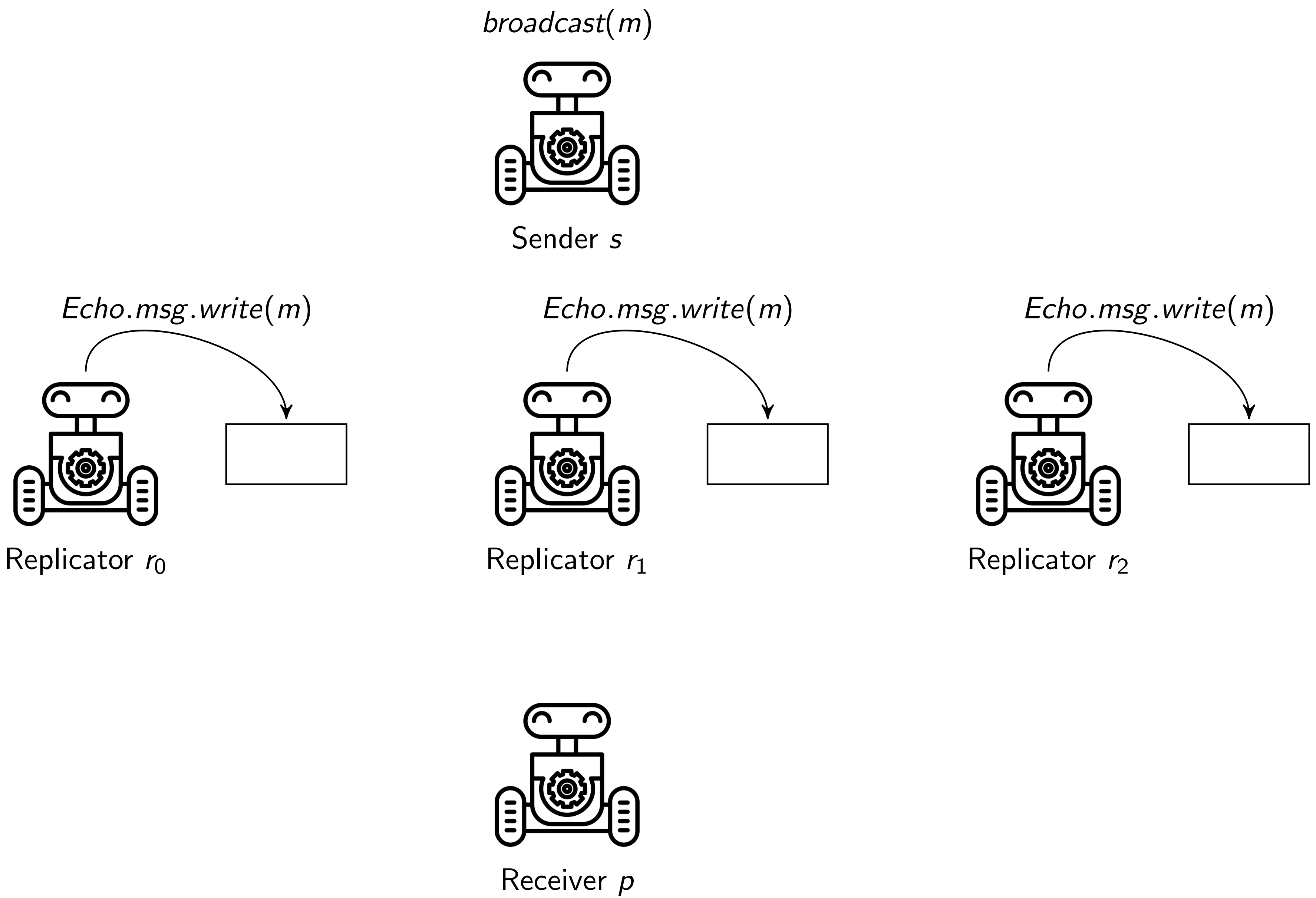
Reliable Broadcast

Algorithm sketch, $f = 1$. Fast path



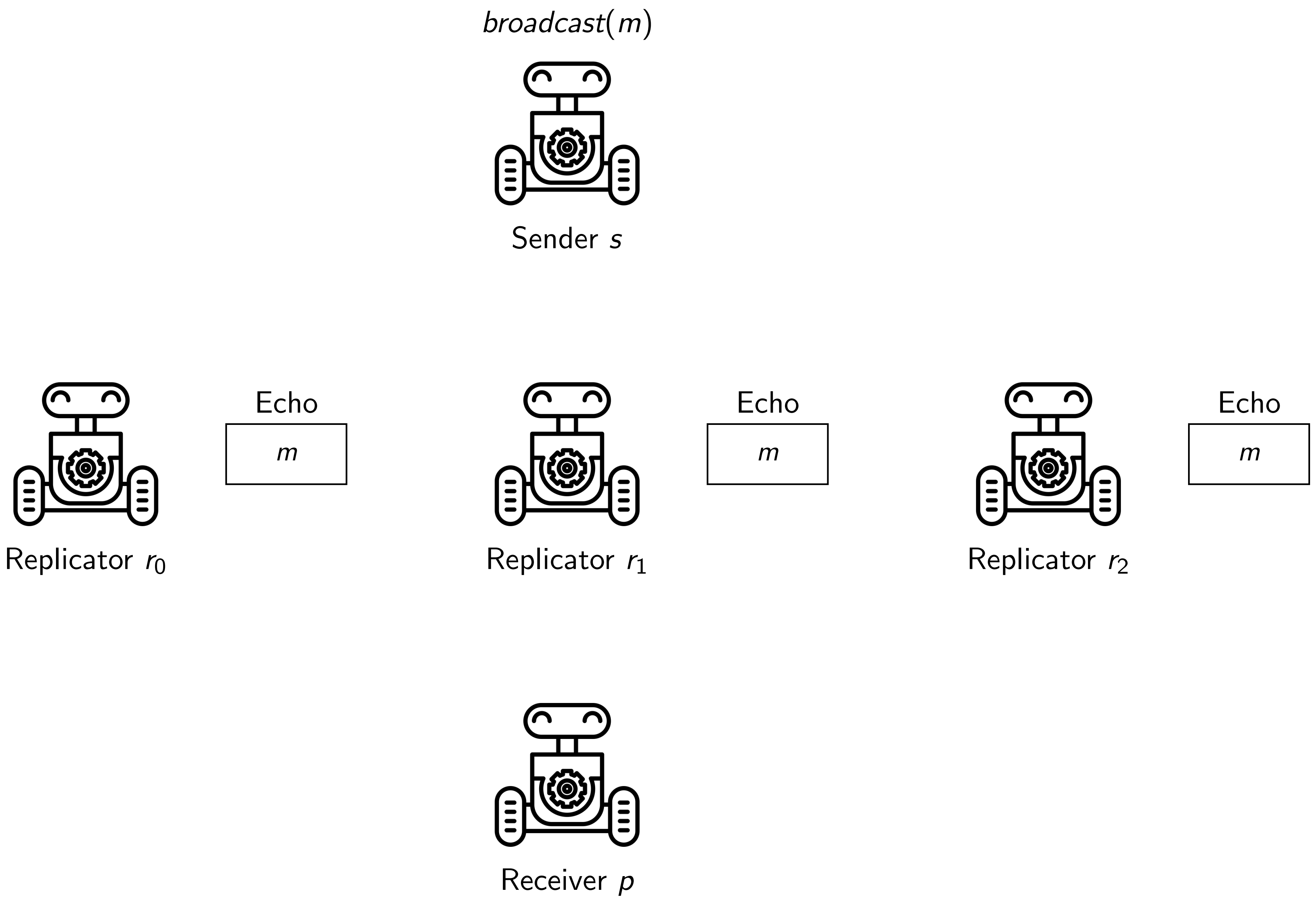
Reliable Broadcast

Algorithm sketch, $f = 1$. Fast path



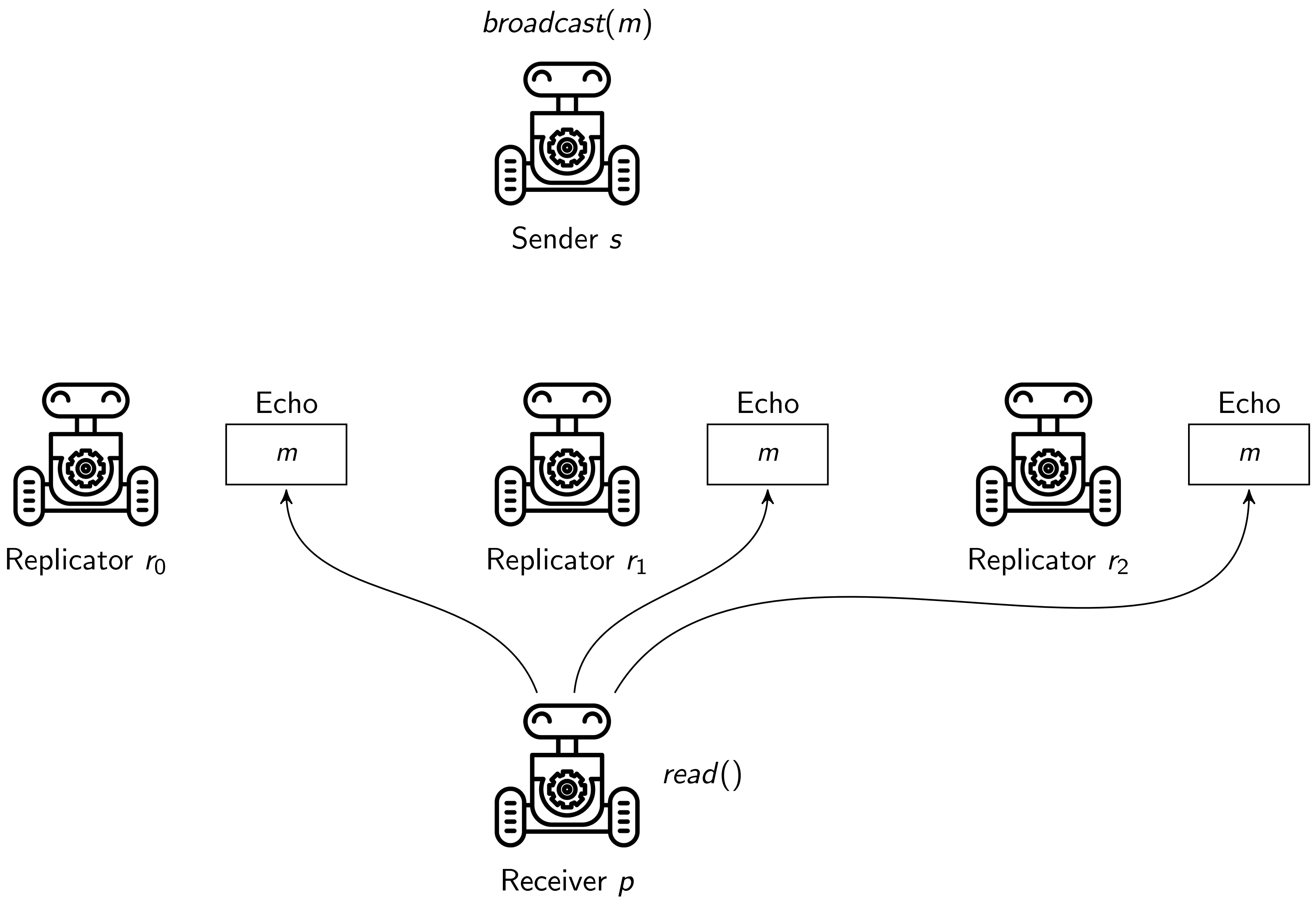
Reliable Broadcast

Algorithm sketch, $f = 1$. Fast path



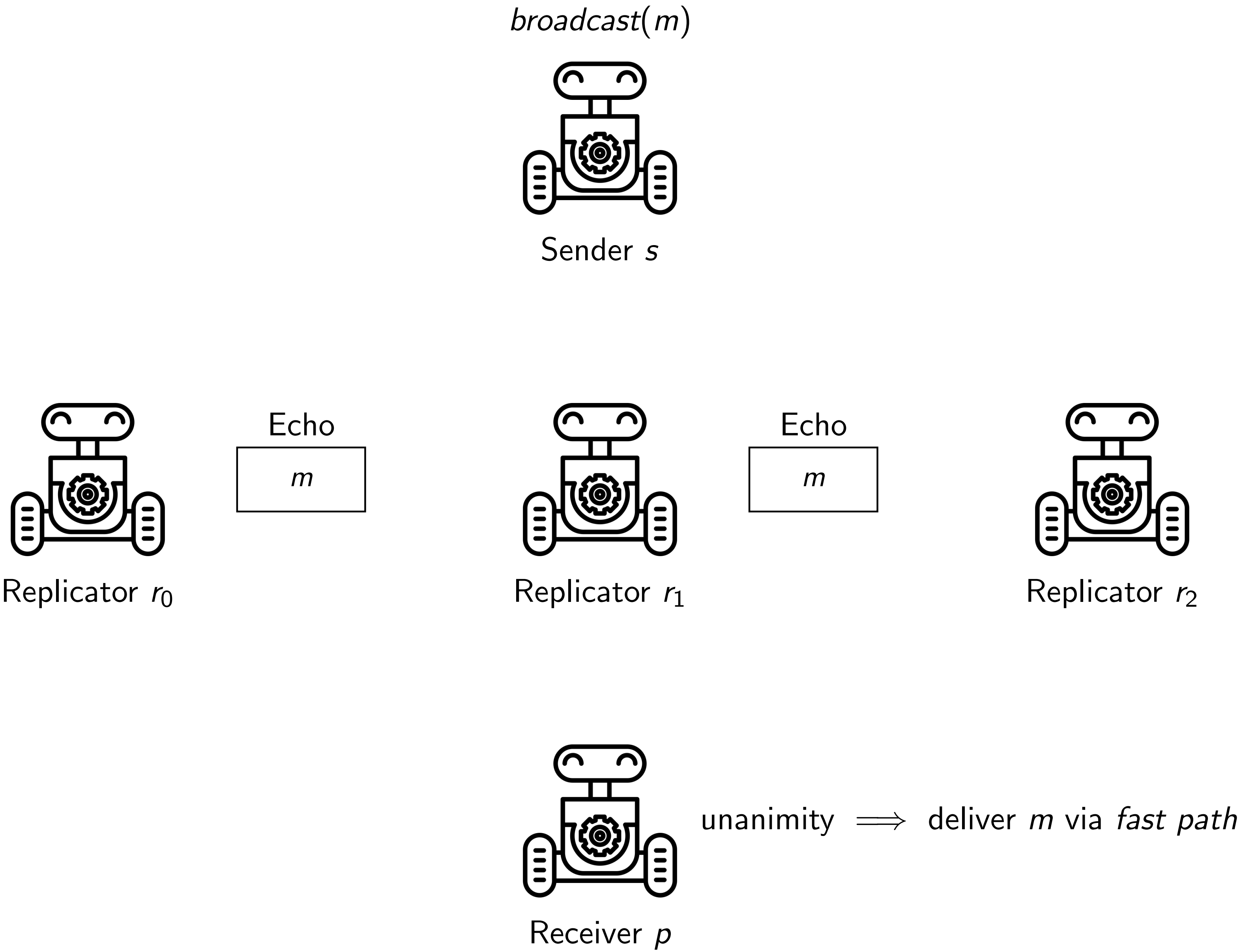
Reliable Broadcast

Algorithm sketch, $f = 1$. Fast path



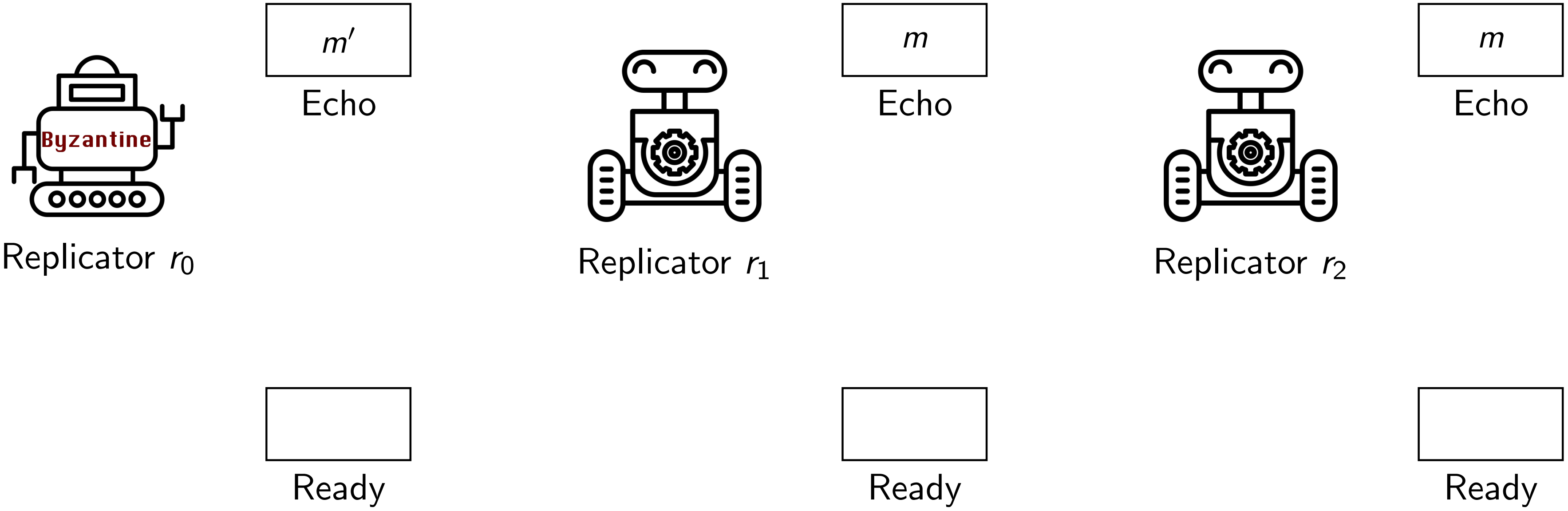
Reliable Broadcast

Algorithm sketch, $f = 1$. Fast path



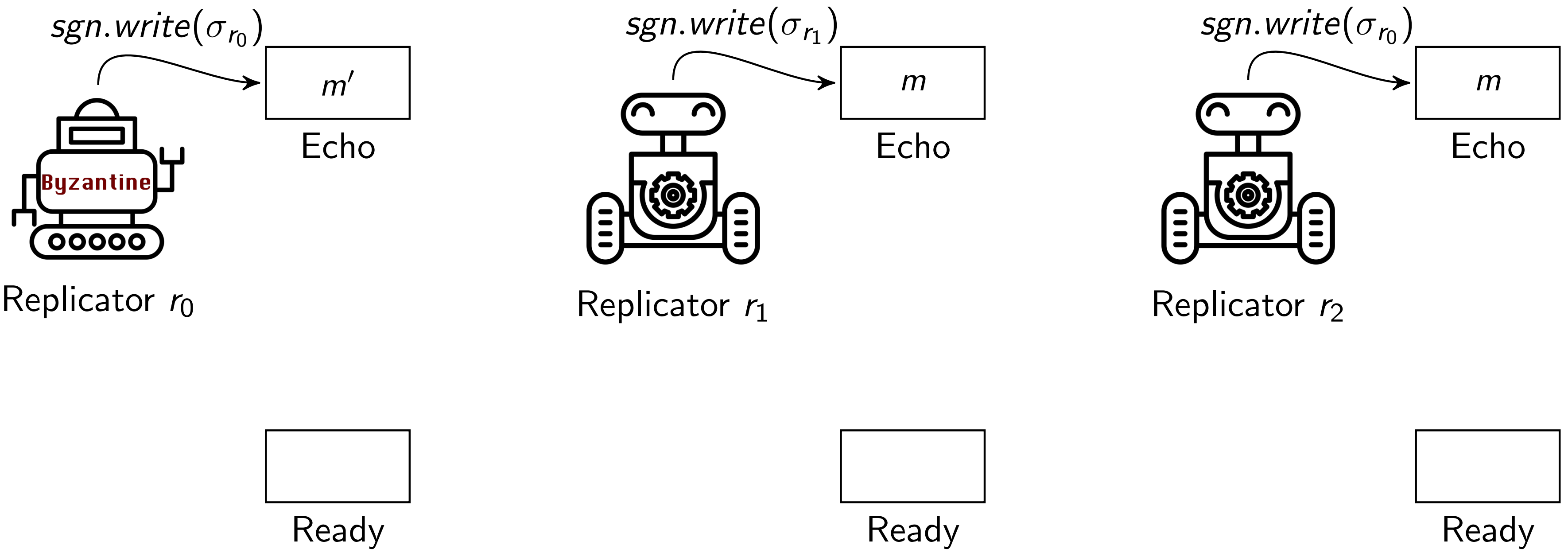
Reliable Broadcast

Algorithm sketch, $f = 1$. Construction of *ReadySet*



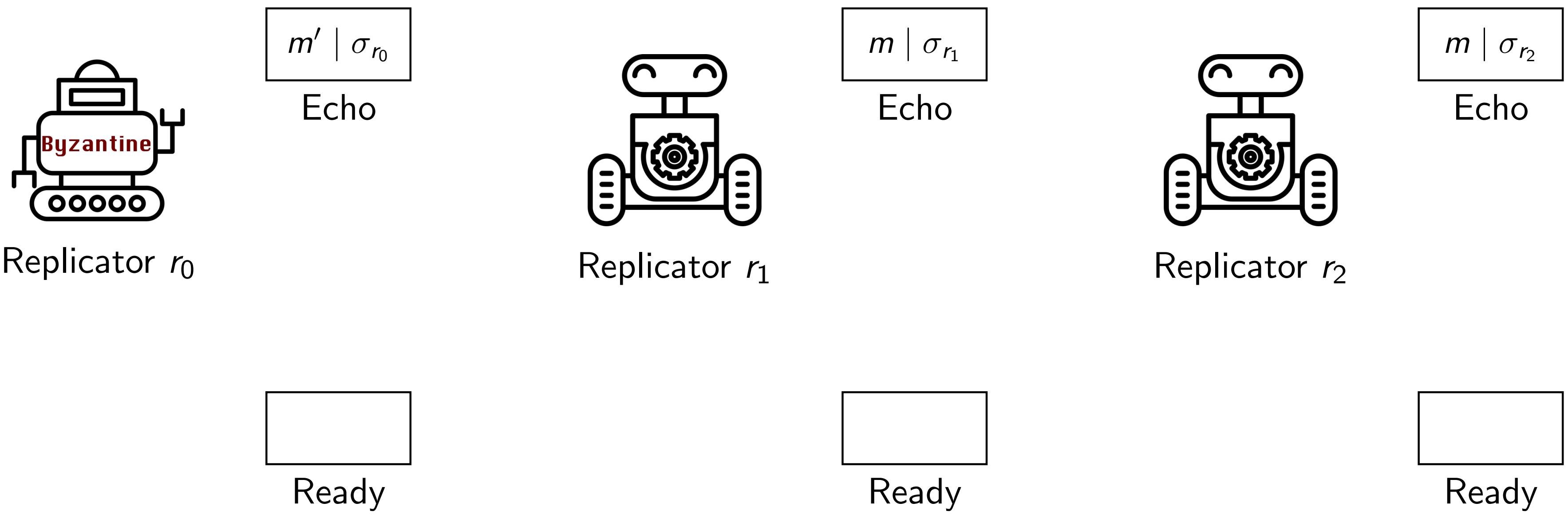
Reliable Broadcast

Algorithm sketch, $f = 1$. Construction of *ReadySet*



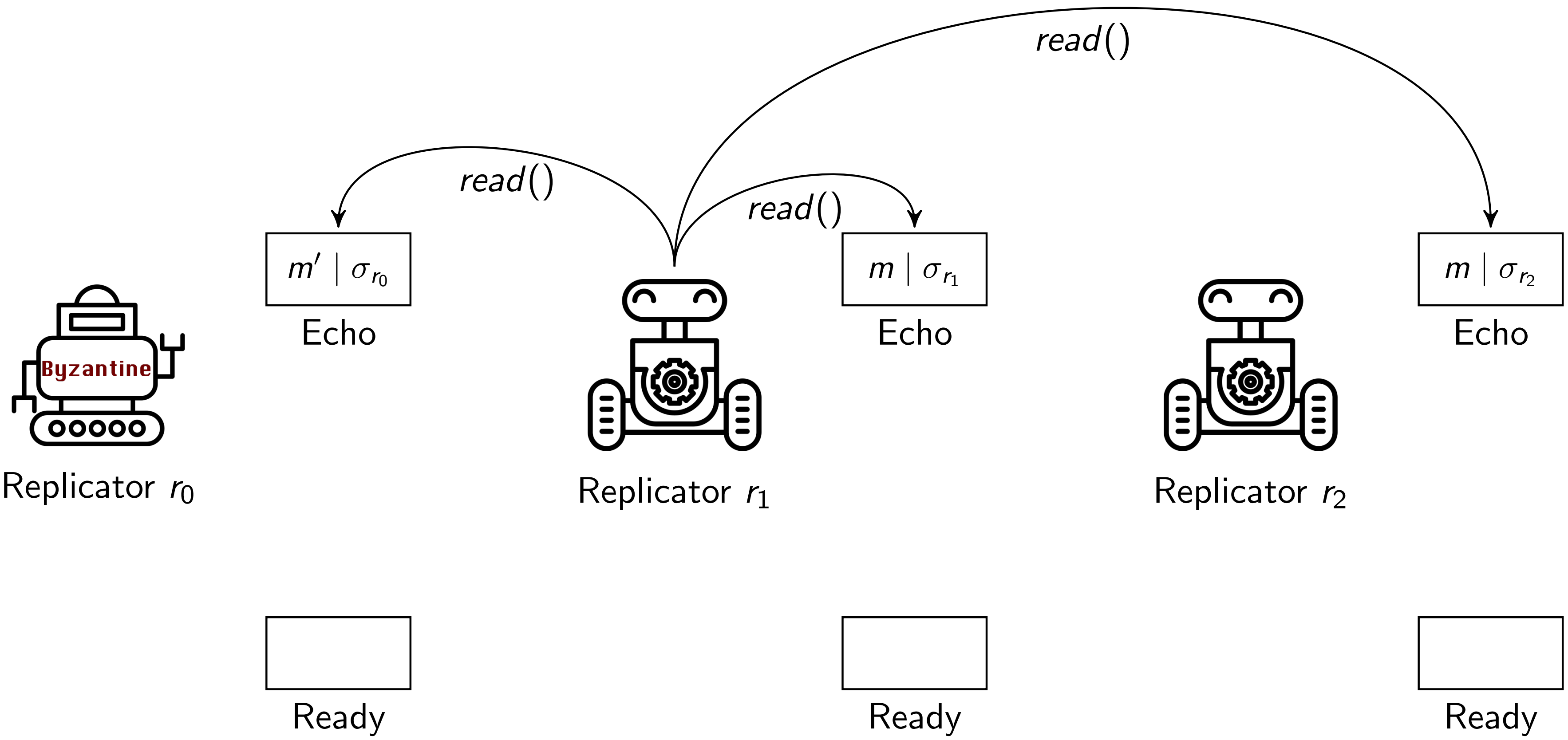
Reliable Broadcast

Algorithm sketch, $f = 1$. Construction of *ReadySet*



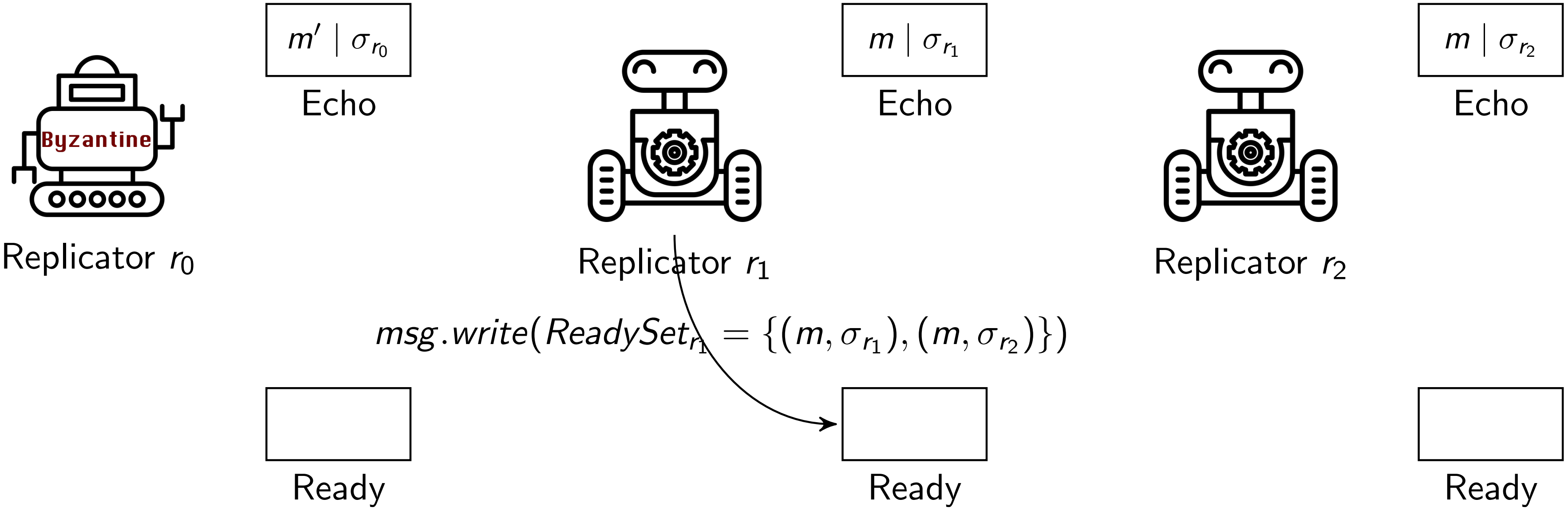
Reliable Broadcast

Algorithm sketch, $f = 1$. Construction of *ReadySet*



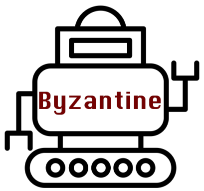
Reliable Broadcast

Algorithm sketch, $f = 1$. Construction of *ReadySet*



Reliable Broadcast

Algorithm sketch, $f = 1$. Construction of *ReadySet*

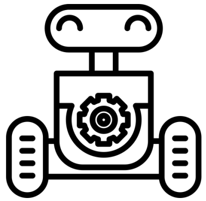


Replicator r_0

$m' \mid \sigma_{r_0}$

Echo

Ready



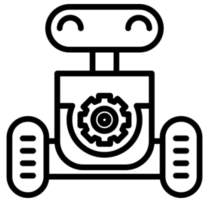
Replicator r_1

$m \mid \sigma_{r_1}$

Echo

$ReadySet_{r_1}$

Ready



Replicator r_2

$m \mid \sigma_{r_2}$

Echo

Ready

Icon credits

From the Noun Project[1]:

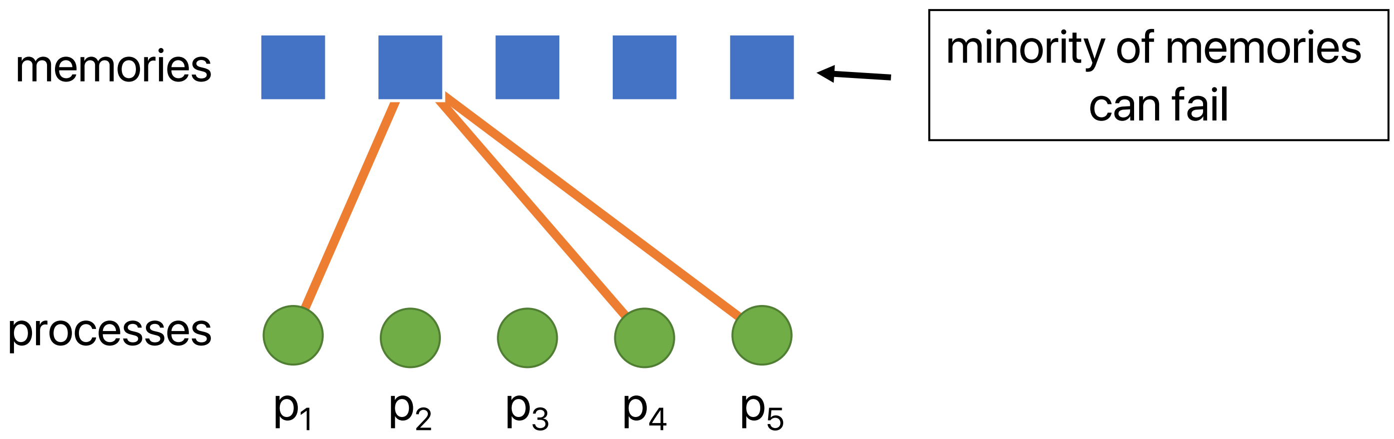
- Server by *nauraicon*
- Money bag by *Mello*
- Robots by *iconcheese*

[1] `https://thenounproject.com/`

Outline

- Introduction
- 3 remarkable results with RDMA:
  - Consensus with crash faults
  - Broadcast with Byzantine faults
  - Fast memory replication

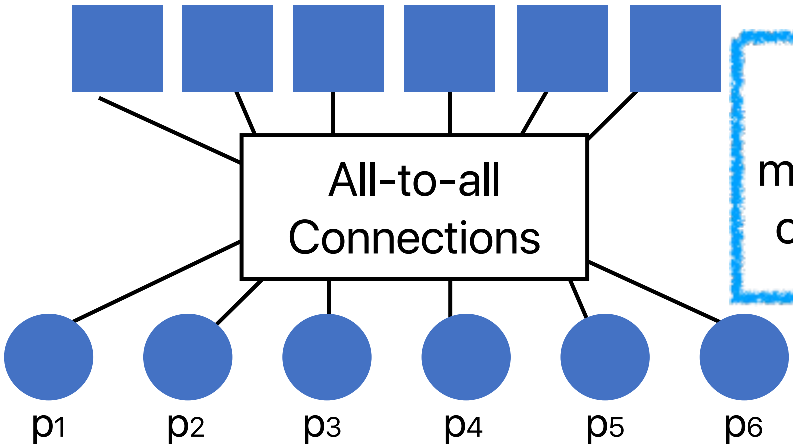
Recall: Disaggregated Memory



Handling Memory Failures

Replication: Treat all memories the same

Send all write/read requests to all memories, wait to hear acknowledgement from majority



Instead of many faulty memories, we can now think of one non-faulty memory!

Exercise Show that this implements a regular register, but not an atomic register!

Reliable MRMW Atomic Register

- We want to implement an atomic MRMW register on a set of unreliable (fault-prone) memories
- We want to minimize the number of round trips (RTTs) per operation.
- Proven: cannot be solved s.t. each operation always takes 1 RTT.
- But can it be done s.t. operations take 1 RTT most of the time?
- To simplify the problem, we assume each memory has plenty of **max registers.**

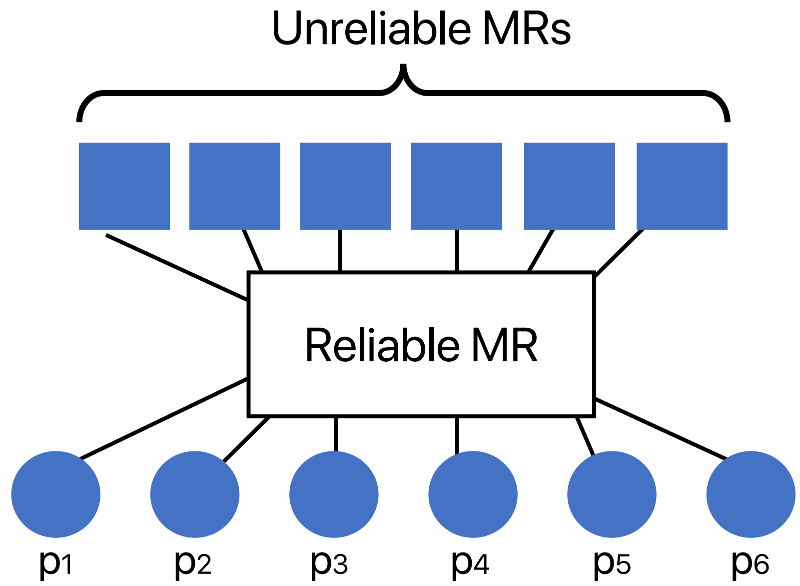
Max Registers

- Two operations: read and write
- Intuitively: read returns highest value written so far
- Formally:
  - **Validity**: If read R returns v, then either (a) $v = \perp$, or (b) some operation write(v) was invoked before R returns.
  - **Read-read monotonicity**: If a read returns value y and a preceding read returns value x, then $x \leq y$.
  - **Write-read monotonicity**: If a read returns value y and a preceding write writes value x, then $x \leq y$.
  - **Liveness (wait-freedom):** Every invoked operation eventually returns.

Step 1: Reliable Max Register

Exercise

- Implement a reliable max register from a set of unreliable max registers
- Writes should complete in 0-1 RTTs, reads should complete in 1-2 RTTs.
- Common case: both operations should take 1 RTT.
- Hint: use caching.



Step 2: Atomic MRMW Register

Classic Algorithm

M = Reliable max register. Each value is a tuple (timestamp, id, value). Lexicographic ordering.

```
def WRITE(v): // this block is not atomic
  fresh_ts = (M.READ().ts.i + 1, tid)
  M.WRITE((fresh_ts, v))

def READ():
  return M.READ().v
```

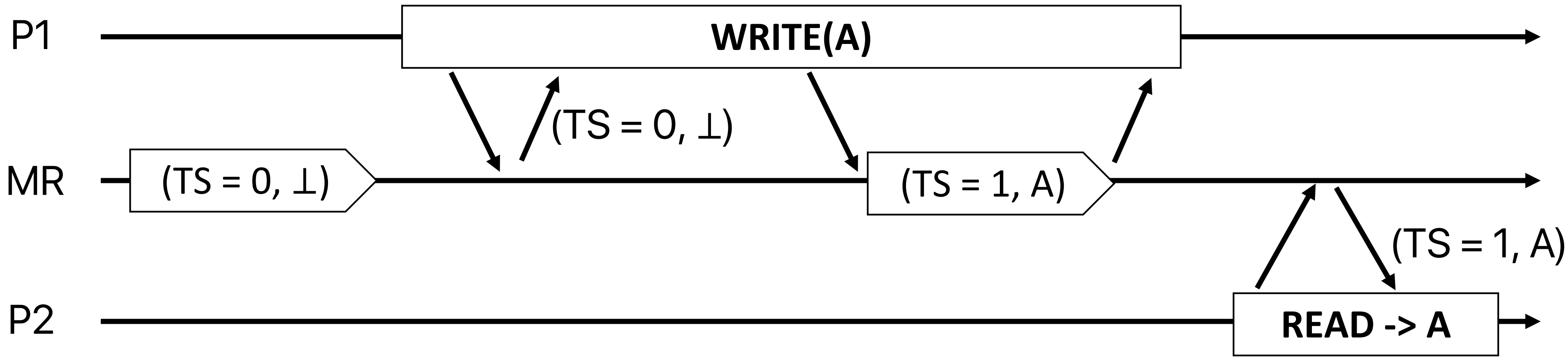
2 RTTs

1 RTT

Why 2 RTTs for WRITE? Can we do better?

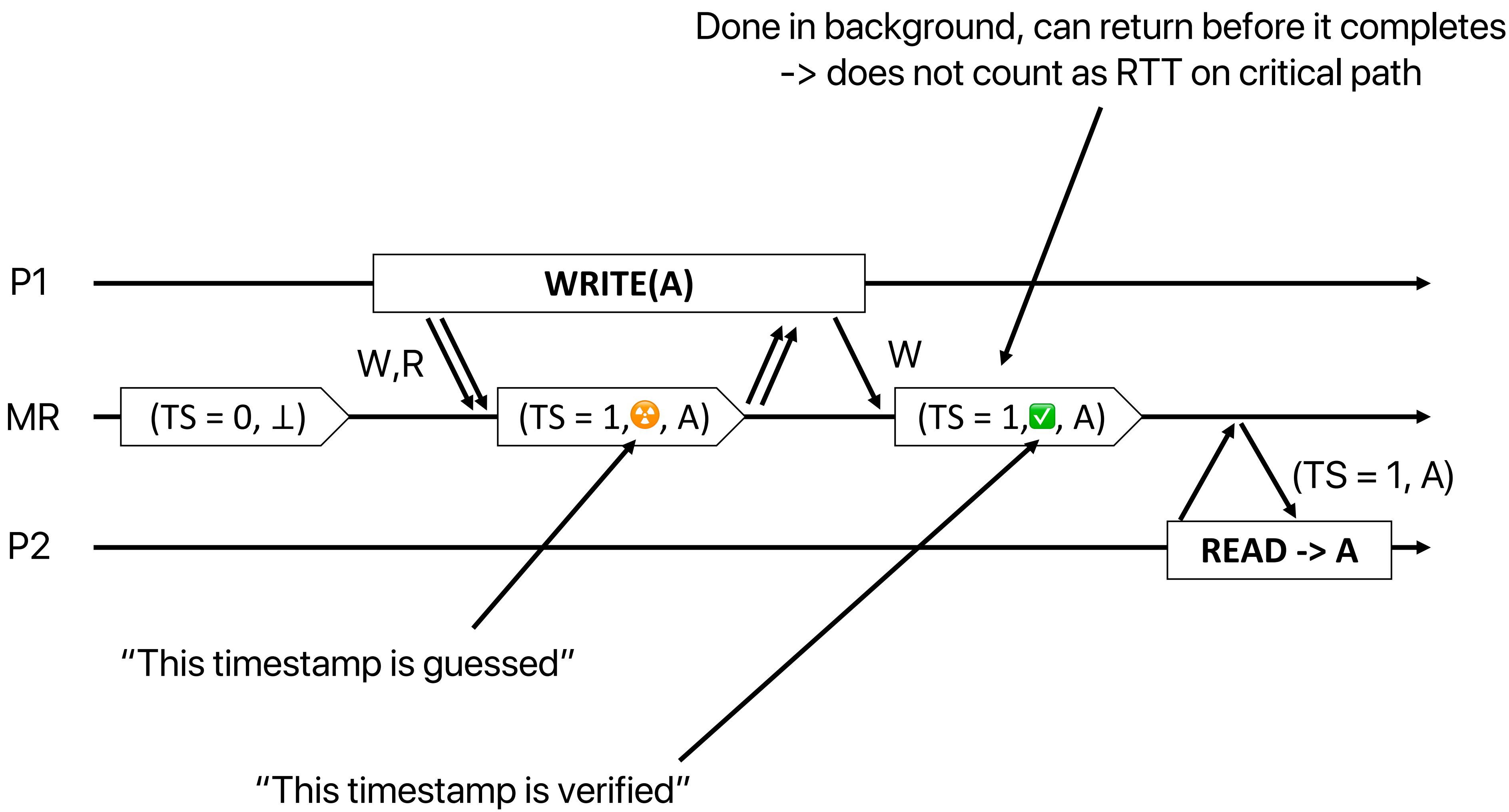
Example

- Each write needs to use a fresh timestamp, i.e., higher than all preceding (why?)
- Finding a fresh timestamp takes 1 RTT.



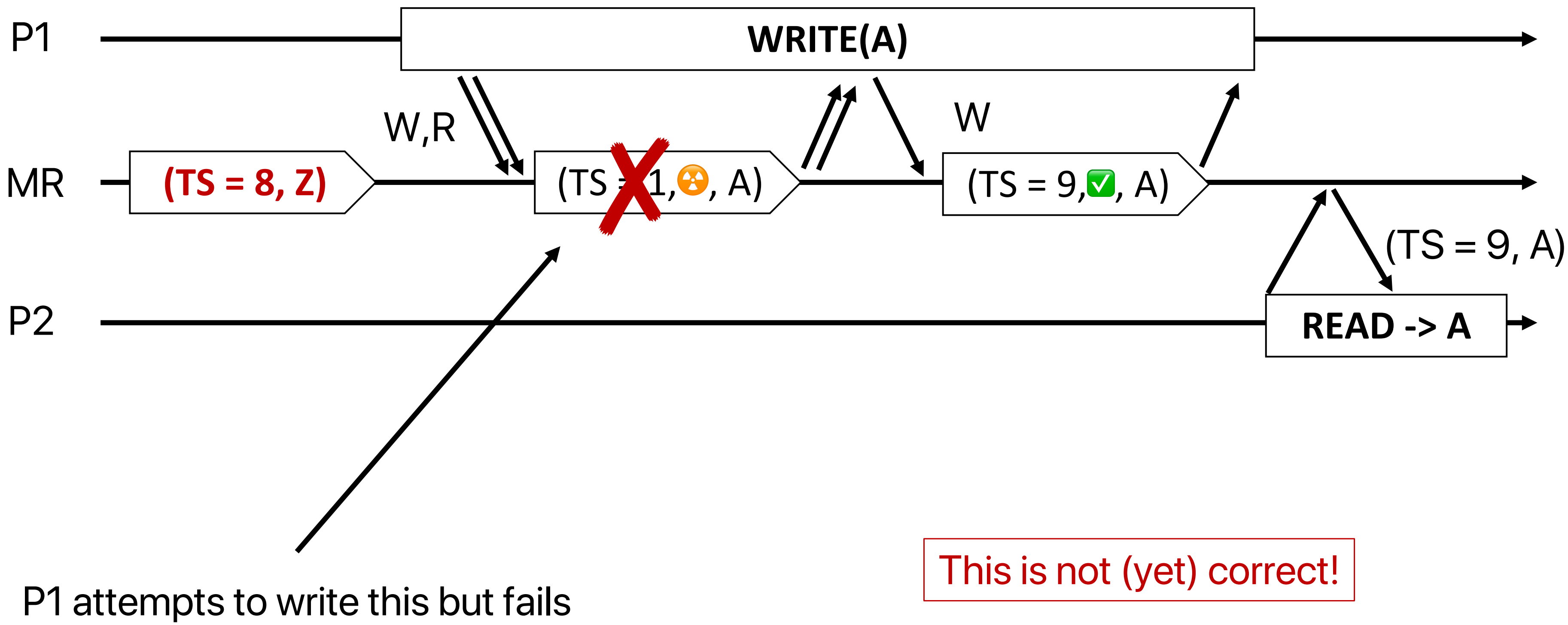
2nd WRITE RTT is unavoidable.
1st WRITE RTT: Could we guess the timestamp?

Guessing Timestamps



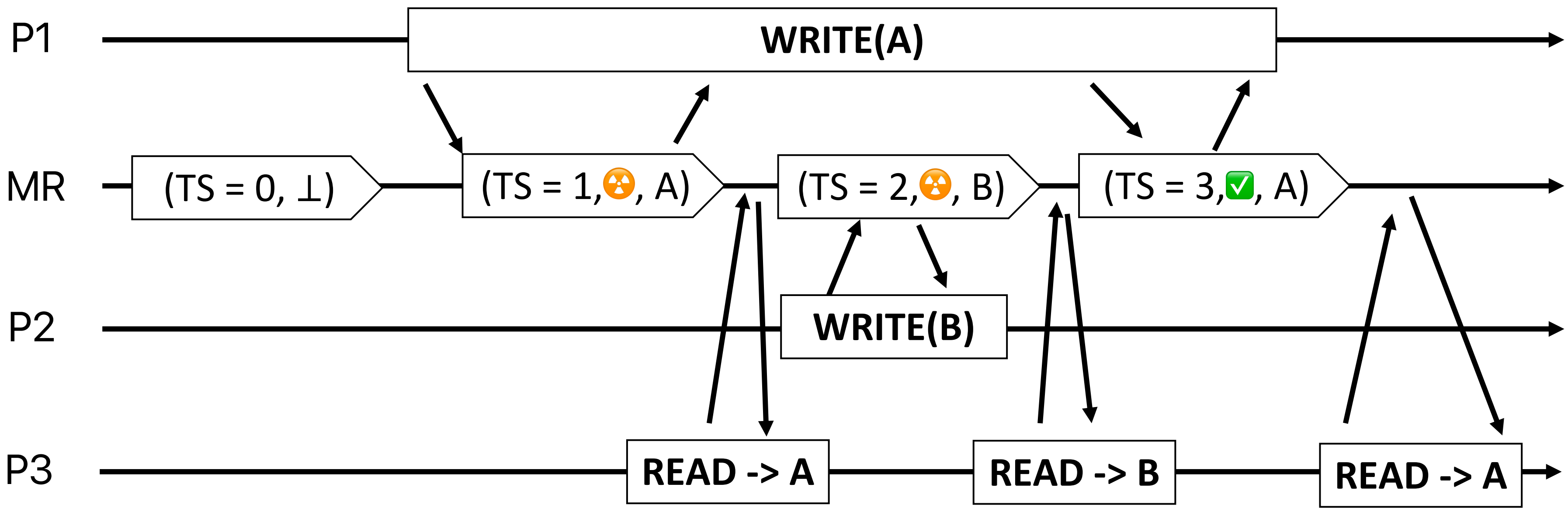
Guessing Timestamps

What if guessed timestamp is wrong?



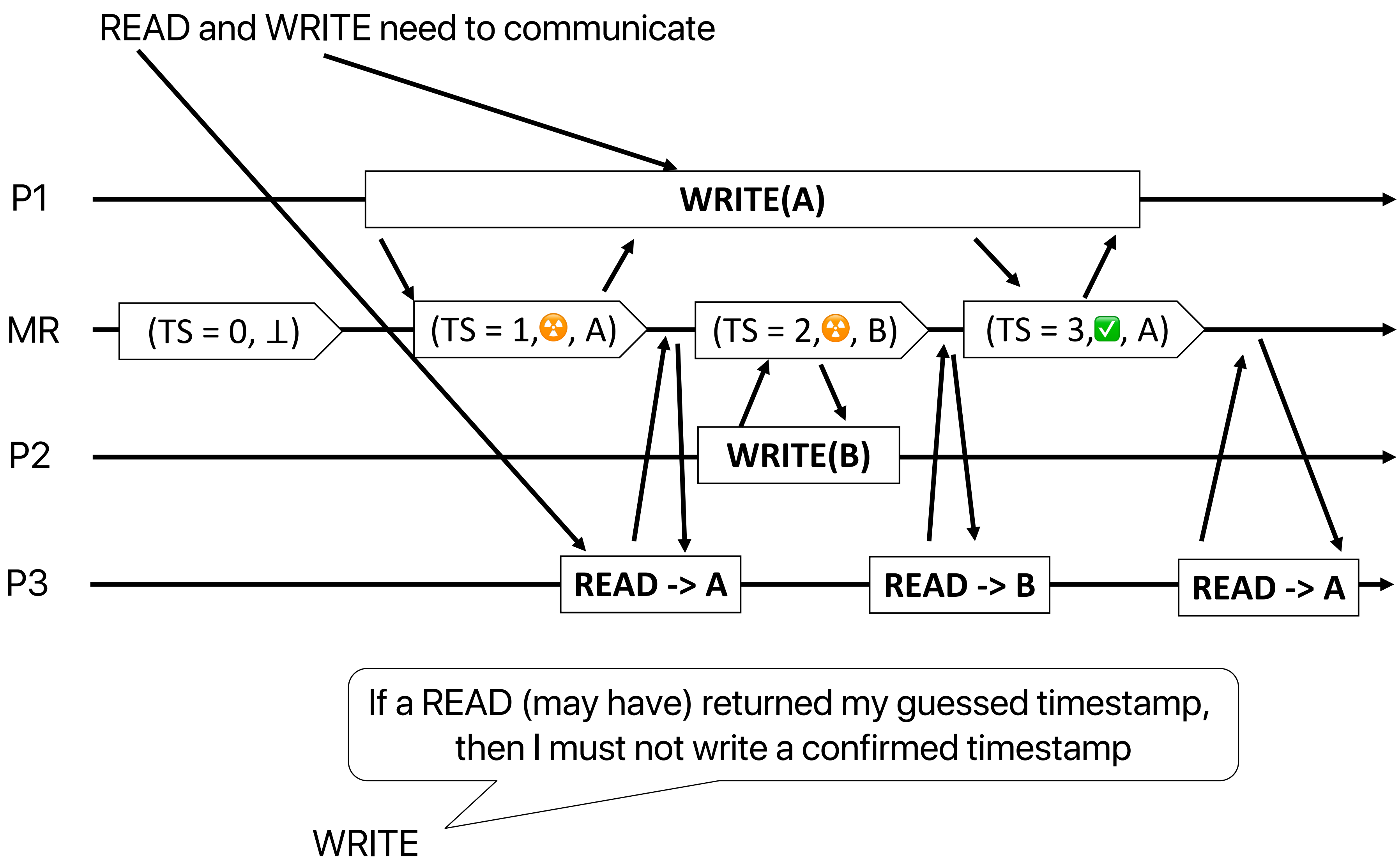
This is not (yet) correct!

Guessing Timestamps



Not atomic/linearizable ☹

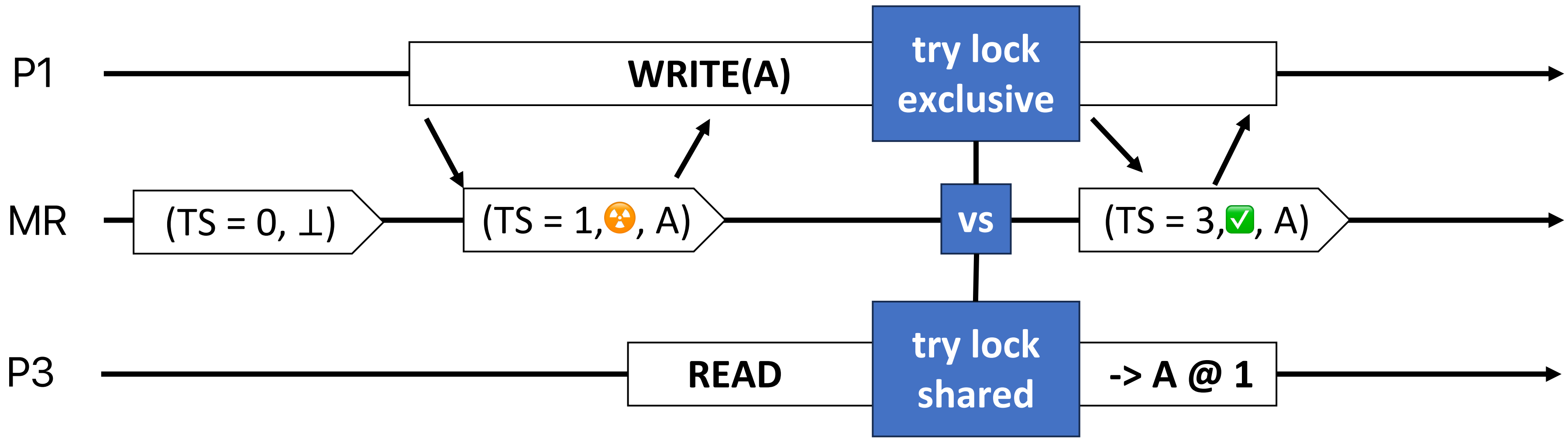
Solution:


READ and WRITE need to communicate
P1
WRITE(A)
MR
(TS = 0, ⊥)
(TS = 1,☢️, A)
(TS = 2,☢️, B)
(TS = 3,✅, A)
P2
WRITE(B)
P3
READ -> A
READ -> B
READ -> A
If a READ (may have) returned my guessed timestamp,
then I must not write a confirmed timestamp
WRITE

Solution

If a READ (may have) returned my guessed timestamp, then I must not write a confirmed timestamp

WRITE



Putting It All Together

Write algorithm

```
M = ((0, ⊥), VERIFIED, ⊥) // Max Register
TSL[tid] = {} // Timestamp Lock

def WRITE(v):
  w = (guessTs(), GUESSED, v)
  in parallel {m = M.READ(), M.WRITE(w)}
  if m <= w: // Fast path (fresh timestamp)
    in bg: M.WRITE(w with VERIFIED) // Spdup reads
  else: // Slow path (potentially stale timestamp)
    if TSL[tid].TRYLOCK(w.ts, WRITE):
      M.WRITE(((m.ts.i+1, tid), VERIFIED, v))
```

guess a timestamp
write guessed ts + read current ts
if guessed ts is fresh:
write verified ts in bg
if guessed ts is stale:
try to take exclusive lock
if successful, write fresh ts

Common case: 1 RTT!

Putting It All Together

Read algorithm

```
def READ():
  seen: dict<ThreadId, MValue> = {}
  while True:
    m = M.READ()
    if m is VERIFIED: return m.v // Fast path
    if m in seen.values: // Fresh timestamp
      if TSL[m.ts.tid].TRYLOCK(m.ts, READ):
        in bg: M.WRITE(m with VERIFIED) // Spdup rds
        return m.v
    elif m.ts.tid in seen.keys: // Wait-free path
      return seen[m.ts.tid].v
    seen[m.ts.tid] = m
```

read from MR
if ts is verified, return it

try to take shared lock
if successful, help reads & return

Common case: 1 RTT!

Putting It All Together

Read algorithm

```
def READ():
 seen: dict<ThreadId, MValue> = {}
 while True:
  m = M.READ()
  if m is VERIFIED: return m.v // Fast path
  if m in seen.values: // Fresh timestamp
   if TSL[m.ts.tid].TRYLOCK(m.ts, READ):
    in bg: M.WRITE(m with VERIFIED) // Spdup rds
    return m.v
  elif m.ts.tid in seen.keys: // Wait-free path
   return seen[m.ts.tid].v
  seen[m.ts.tid] = m
```

What about all this other stuff?

It's for wait-freedom. Check out the paper for the full explanation:

"SWARM: Replicating Shared Disaggregated-Memory Data in No Time"

Further Reading

1. ABGMZ. *The Impact of RDMA on Agreement*. PODC 2019.
2. ABGMXZ. *Microsecond Consensus for Microsecond Applications*. OSDI 2020.
3. ABGPXZ. *Frugal Byzantine Computing*. DISC 2021.
4. ABGMXZ. *uBFT: Microsecond-Scale BFT using Disaggregated Memory*. ASPLOS 2023.
5. MBXZAG. *SWARM: Replicating Shared Disaggregated-Memory Data in No Time*. SOSP 2024