# Distributed consensus revised

Heidi Howard @ Cambridge University

heidi.howard@cl.cam.ac.uk
@heidiann360
heidihoward.co.uk

# The story so far…

Flexible Paxos: Quorum Intersection Revisited

Heidi Howard[1], Dahlia Malkhi[2], and Alexander Spiegelman[3]

1 VMware Research, Palo Alto, CA, USA; and University of Cambridge Computing Laboratory, Cambridge, UK heidi.howard@cl.cam.ac.uk
2 VMware Research, Palo Alto, CA, USA dahliamalkhi@gmail.com
3 VMware Research, Palo Alto, CA, USA; and Viterbi Dept. of Electrical Engineering, Technion Haifa, Haifa, Israel sashas@tx.technion.ac.il

Abstract

Distributed consensus is integral to modern distributed systems. The widely adopted Paxos algorithm uses two phases, each requiring majority agreement, to reliably reach consensus. In this paper, we demonstrate that Paxos, which lies at the foundation of many production systems, is conservative. Specifically, we observe that each of the phases of Paxos may use non-intersecting quorums. Majority quorums are not necessary as intersection is required only across phases.

Using this weakening of the requirements made in the original formulation, we propose Flexible Paxos, which generalizes over the Paxos algorithm to provide flexible quorums. We show that Flexible Paxos is safe, efficient and easy to utilize in existing distributed systems. We discuss far reaching implications of this result. For example, improved availability results from reducing the size of second phase quorums by one when the system size is even, while keeping majority quorums in the first phase. Another example is improved throughput of replication by using much smaller phase 2 quorums, while increasing the leader election (phase 1) quorums. Finally, non intersecting quorums in either first or second phases may enhance the efficiency of both.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Paxos, Distributed Consensus, Quorums

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.25

1 Introduction

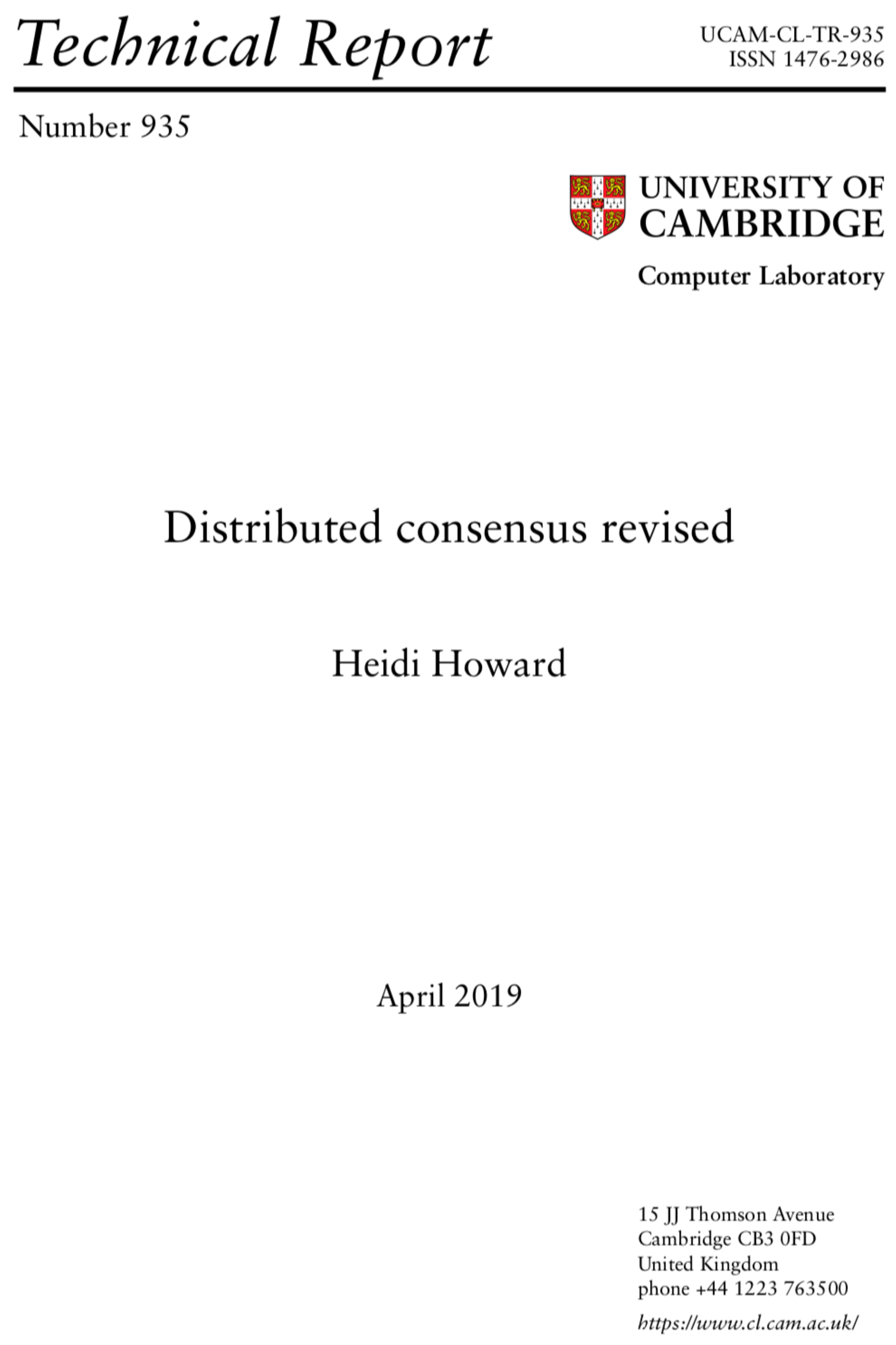
Distributed consensus is the problem of reaching agreement in the face of failures. It is a common problem in modern distributed systems and its applications range from distributed locking and atomic broadcast to strongly consistent key value stores and state machine replication [36]. Lamport's Paxos algorithm [19, 20] is one such solution to this problem and since its publication it has been widely built upon in teaching, research and practice.

At its core, Paxos uses two phases, each requires agreement from a subset of participants (known as a quorum) to proceed. The safety and liveness of Paxos is based on the guarantee that any two quorums will intersect. To satisfy this requirement, quorums are typically composed of any majority from a fixed set of participants, although other quorum schemes have been proposed.

In practice, we usually wish to reach agreement over a sequence of values, known as Multi-Paxos [20]. We use the first phase of Paxos to establish one participant as a *leader* and the second phase of Paxos to propose a series of values. To commit a value, the leader must

© Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman; licensed under Creative Commons License CC-BY
20th International Conference on Principles of Distributed Systems (OPODIS 2016).
Editors: Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone; Article No. 25; pp. 25:1–25:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Flexible Paxos: Quorum intersection revisited, 2016



Technical Report UCAM-CL-TR-935 ISSN 1476-2986

Number 935

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Distributed consensus revised

Heidi Howard

April 2019

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500
*https://www.cl.cam.ac.uk/*

Distributed consensus revised, 2018

A Generalised Solution to Distributed Consensus

Heidi Howard, Richard Mortier
University of Cambridge
first.last@cl.cam.ac.uk

Abstract

Distributed consensus, the ability to reach agreement in the face of failures and asynchrony, is a fundamental primitive for constructing reliable distributed systems from unreliable components. The Paxos algorithm is synonymous with distributed consensus, yet it performs poorly in practice and is famously difficult to understand. In this paper, we re-examine the foundations of distributed consensus. We derive an abstract solution to consensus, which utilises immutable state for intuitive reasoning about safety. We prove that our abstract solution generalises over Paxos as well as the Fast Paxos and Flexible Paxos algorithms. The surprising result of this analysis is a substantial weakening to the quorum requirements of these widely studied algorithms.

1 Introduction

We depend upon distributed systems, yet the computers and networks that make up these systems are asynchronous and unreliable. The longstanding problem of distributed consensus formalises how to reliably reach agreement in such systems. When solved, we become able to construct strongly consistent distributed systems from unreliable components [13, 21, 4, 17]. Lamport's Paxos algorithm [14] is widely deployed in production to solve distributed consensus [5, 6], and experience with it has led to extensive research to improve its performance and our understanding but, despite its popularity, both remain problematic.

Paxos performs poorly in practice because its use of majorities means that each decision requires a round trip to many participants, thus placing substantial load on each participant and the network connecting them. As a result, systems are typically limited in practice to just three or five participants. Furthermore, Paxos is usually implemented in the form of *Multi-Paxos*, which establishes one participant as the *master*, introducing a performance bottleneck and increasing latency as all decisions are forwarded via the master. Given these limitations, many production systems often opt to sacrifice strong consistency guarantees in favour of performance and high availability [7, 3, 18]. Whilst compromise is inevitable in practical distributed systems [10], Paxos offers just one point in the space of possible trade-offs. In response, this paper aims to improve performance by offering a generalised solution allowing engineers the flexibility to choose their own trade-offs according to the needs of their particular application and deployment environment.

Paxos is also notoriously difficult to understand, leading to much follow up work, explaining the algorithm in simpler terms [20, 15, 19, 23] and filling the gaps in the original description, necessary for constructing practical systems [6, 2]. In recent years, immutability has been increasingly widely utilised in distributed systems to tame complexity [11]. Examples such as append-only log stores [1, 8] and CRDTs [22] have inspired us to apply immutability to the problem of consensus.

1

A generalised solution to distributed consensus, 2019

# Distributed Dream

**Performance** – scalability, low latency, high throughput, low cost, energy efficiency, versatility, adaptability

**Reliability** – fault-tolerance, dependability, high availability, self-healing, geo-replicated

**Correctness** – consistency, bug-free, easy to understand

# A Hundred Impossibility Proofs for Distributed Computing

Nancy A. Lynch *
Lab for Computer Science
MIT, Cambridge, MA 02139
lynch@tds.lcs.mit.edu

## 1 Introduction

This talk is about impossibility results in the area of distributed computing. In this category, I include not just results that say that a particular task cannot be accomplished, but also lower bound results, which say that a task cannot be accomplished within a certain bound on cost.

I started out with a simple plan for preparing this talk: I would spend a couple of weeks reading all the impossibility proofs in our field, and would categorize them according to the ideas used. Then I would make wise and general observations, and try to predict where the future of this area is headed. That turned out to be a bit too ambitious; there are many more such results than I thought. Although it is often hard to say what constitutes a "different result", I managed to count over 100 such impossibility proofs! And my search wasn't even very systematic or exhaustive.

It's not quite as hopeless to understand this area as it might seem from the number of papers. Although there are 100 different results, there aren't 100 different ideas. I thought I could contribute something by identifying some of the commonality among the different results.

So what I will do in this talk will be an incomplete version of what I originally intended. I will give you

*This work was supported in part by the National Science Foundation (NSF) under Grant CCR-86-11442, by the Office of Naval Research (ONR) under Contract N00014-85-K-0168 and by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125.

**Keywords:** impossibility, distributed computing

a tour of the impossibility results that I was able to collect. I apologize for not being comprehensive, and in particular for placing perhaps undue emphasis on results I have been involved in (but those are the ones I know best!). I will describe the techniques used, as well as giving some historical perspective. I'll intersperse this with my opinions and observations, and I'll try to collect what I consider to be the most important of these at the end. Then I'll make some suggestions for future work.

## 2 The Results

I classified the impossibility results I found into the following categories: shared memory resource allocation, distributed consensus, shared registers, computing in rings and other networks, communication protocols, and miscellaneous.

### 2.1 Shared Memory Resource Allocation

This was the area that introduced me not only to the possibility of doing impossibility proofs for distributed computing, but to the entire distributed computing research area.

In 1976, when I was at the University of Southern California, Armin Cremers and Tom Hibbard were playing with the problem of *mutual exclusion* (or allocation of one resource) in a shared-memory environment. In the environment they were considering, a group of asynchronous processes communicate via shared memory, using operations such as read and write or test-and-set.

The previous work in this area had consisted of a series of papers by Dijkstra [38] and others, each presenting a new algorithm guaranteeing mutual exclusion, along with some other properties such as progress and fairness. The properties were specified somewhat loosely; there was no formal model used for



**[PODC'89]**

# Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER
*Yale University, New Haven, Connecticut*

NANCY A. LYNCH
*Massachusetts Institute of Technology, Cambridge, Massachusetts*

AND

MICHAEL S. PATERSON
*University of Warwick, Coventry, England*

Abstract. The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the "Byzantine Generals" problem.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols–*protocol architecture*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems–*distributed applications; distributed databases; network operating systems*; C.4 [**Performance of Systems**]: Reliability, Availability, and Serviceability; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation–*parallelism*; H.2.4 [**Database Management**]: Systems–*distributed systems; transaction processing*

General Terms: Algorithms, Reliability, Theory

Additional Key Words and Phrases: Agreement problem, asynchronous system, Byzantine Generals problem, commit problem, consensus problem, distributed computing, fault tolerance, impossibility proof, reliability

## 1. *Introduction*

The problem of reaching agreement among remote processes is one of the most fundamental problems in distributed computing and is at the core of many

Editing of this paper was performed by guest editor S. L. Graham. The Editor-in-Chief of JACM did not participate in the processing of the paper.

This work was supported in part by the Office of Naval Research under Contract N00014-82-K-0154, by the Office of Army Research under Contract DAAG29-79-C-0155, and by the National Science Foundation under Grants MCS-7924370 and MCS-8116678.

This work was originally presented at the 2nd ACM Symposium on Principles of Database Systems, March 1983.

Authors' present addresses: M. J. Fischer, Department of Computer Science, Yale University, P.O. Box 2158, Yale Station, New Haven, CT 06520; N. A. Lynch, Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139; M. S. Paterson, Department of Computer Science, University of Warwick, Coventry CV4 7AL, England



Journal of the Association for Computing Machinery, Vol. 32, No. 2, April 1985, pp. 374–382.

**[JACM'85]**


Linearizability
Fork
Timed serial & Δ,Γ-atomicity
Regular
Eventual linearizability
Sequential
Prefix linearizable
Weak fork-lin.
Safe
Per-object models
Staleness-based models
Strong eventual
Bounded fork-join causal
Causal+
Real-time causal
Prefix sequential
Processor
Synchronized models
k-atomicity
Eventual serializability
Timed causal
Weak ordering
Bounded staleness & Delta
Per-key sequential
Per-record timeline & Coherence
Causal models
Release
Fork sequential
k-regular
Composite and tunable models
Causal
Lazy release
Fork*
Hybrid
Tunable
Rationing
RedBlue
Conit
Vector-field
PBS <k,t>-staleness
Scope
Per-object causal
Fork-join causal
PRAM (FIFO)
Entry
Location
PBS t-visibility
Slow memory
PBS k-staleness
Fork-based models
k-safe
Writes-follow-reads (WFR)
Read-your-writes (RYW)
Monotonic Writes (MW)
Monotonic Reads (MR)
Eventual
Session models
Quiescent
Weak

**[CSUR'16]**

# Deciding a single value

In this talk, we will reach agreement over a single value

The system is comprised of:

- **servers** which store the value
- **clients** which propose values and learn the decided value

# This is not a blockchain talk

# Requirements of consensus

**Safety** – All clients must learn the same decided value

**Progress** – Eventually, all clients must learn the decided value

# Requirements of consensus

**Safety** – All clients must learn the same decided value

**Progress** – Eventually, all clients must learn the decided value

Safety must hold even in unreliable and asynchronous systems

# The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament's protocol provides a new way of implementing the state machine approach to the design of distributed systems.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*network operating systems*; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*; J.1 [**Computer Applications**]: Administrative Data Processing—*government*

General Terms: Design, Reliability

Additional Key Words and Phrases: State machines, three-phase commit, voting

## 1. THE PROBLEM

### 1.1 The Island of Paxos

Early in this millennium, the Aegean island of Paxos was a thriving mercantile center.[1] Wealth led to political sophistication, and the Paxons replaced their ancient theocracy with a parliamentary form of government. But trade came before civic duty, and no one in Paxos was willing to devote his life to Parliament. The Paxon Parliament had to function even though legislators continually wandered in and out of the parliamentary Chamber.

The problem of governing with a part-time parliament bears a remarkable correspondence to the problem faced by today's fault-tolerant distributed systems, where legislators correspond to processes, and leaving the Chamber corresponds to failing. The Paxons' solution may therefore be of some interest to computer scientists. I present here a short history of the Paxos Parliament's protocol, followed by an even shorter discussion of its relevance for distributed systems.

[TOCS'98]

# Theory perspective

“The Paxos algorithm, when presented in plain English, is very simple.”

“The Paxos algorithm … is among the simplest and most obvious of distributed algorithms”

“… this consensus algorithm follows almost unavoidably from the properties we want it to satisfy.”

Leslie Lamport, Paxos Made Simple

# Engineering perspective

"Paxos is exceptionally difficult to understand… few people succeed in understanding it, and only with great effort. …"

"… we found few people who were comfortable with Paxos, even among seasoned researchers."

"We concluded that Paxos does not provide a good foundation either for system building or for education."

Diego Ongaro and John Ousterhout, In Search of an Understandable Consensus Algorithm

# Limitations of Paxos

# Limitations of Paxos

Paxos is subtle

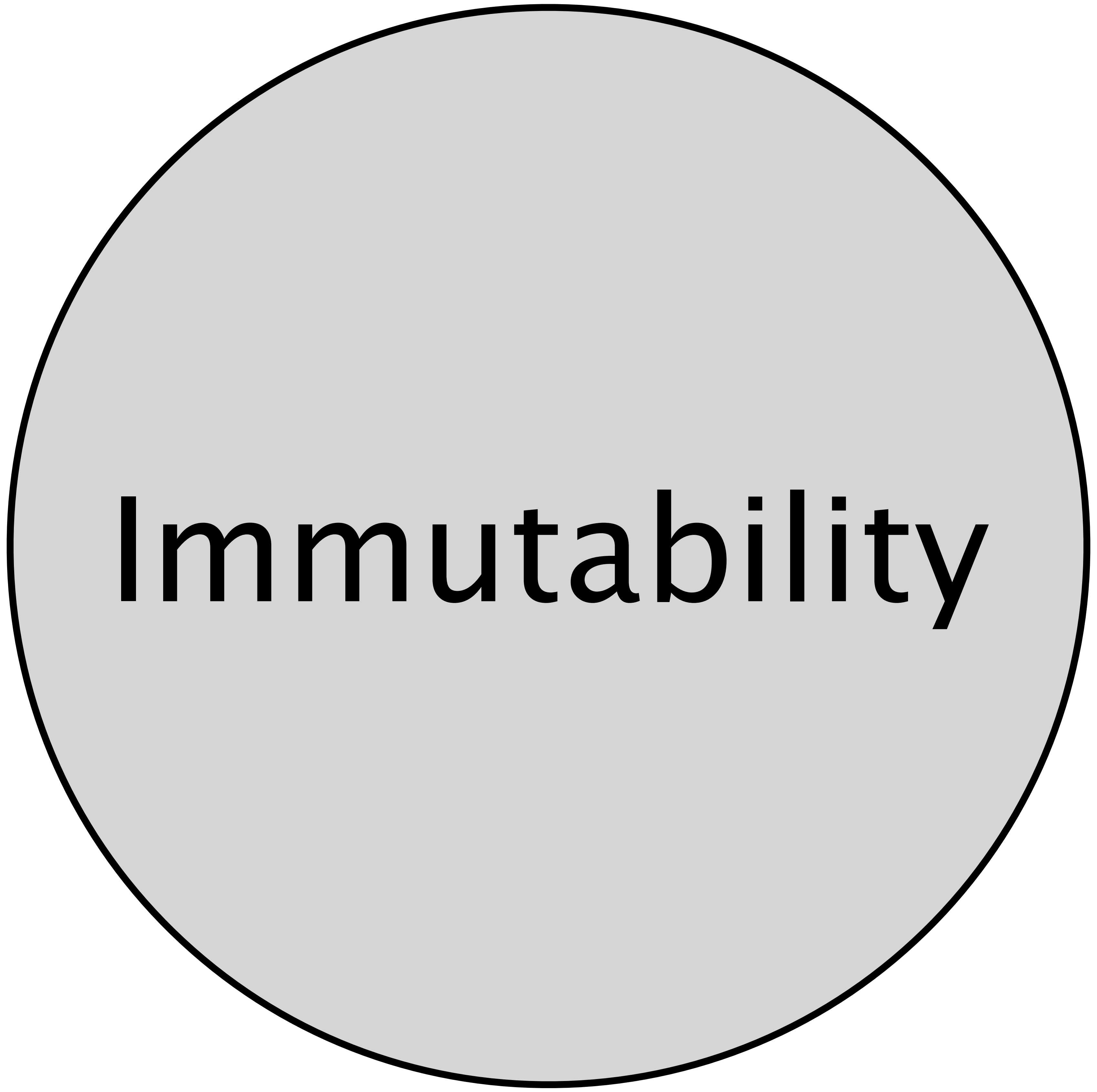
# Limitations of Paxos

Paxos is subtle

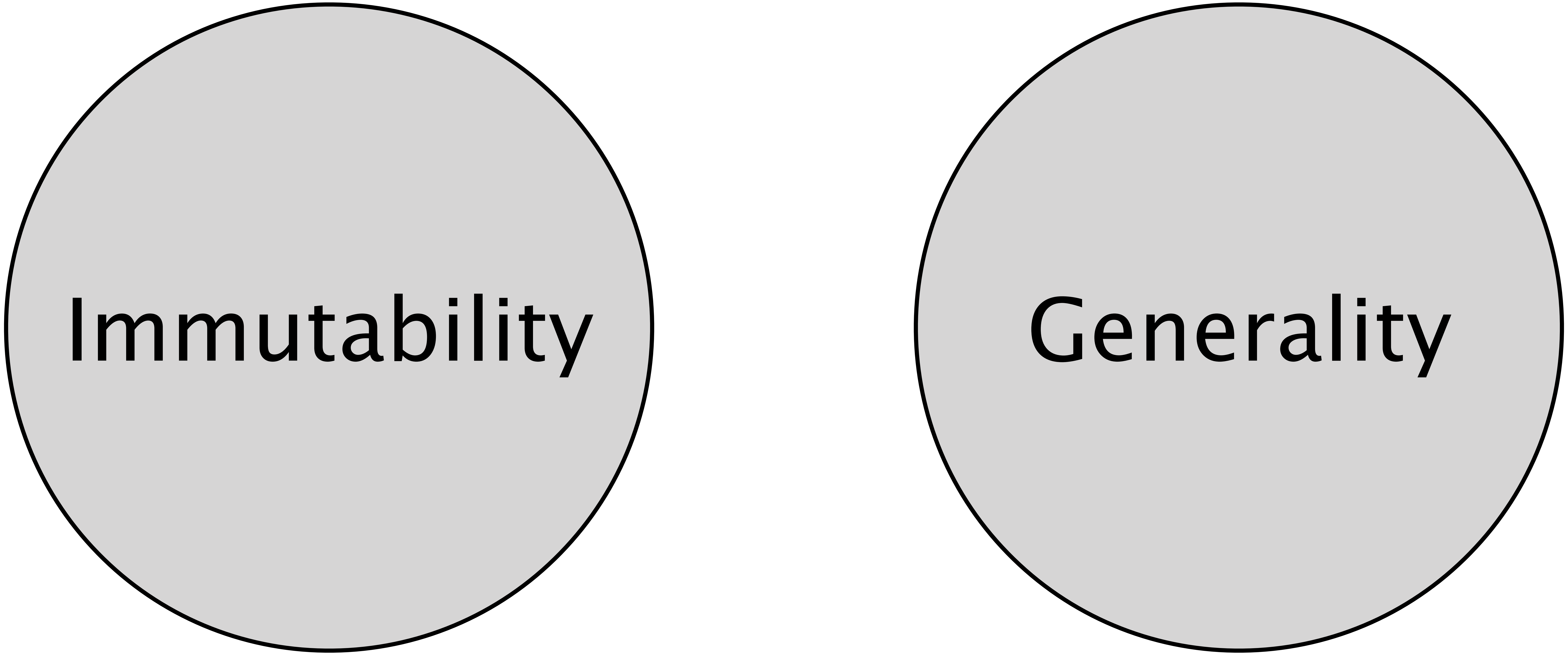
Paxos is slow

# Back to basics

# Back to basics



# Back to basics



# Today's Talk

# Today’s Talk

**Part 1**

We reframe the problem of distributed consensus.

# Today's Talk

**Part 1**

We reframe the problem of distributed consensus.

→

**Part 2**

We generalise the Paxos algorithm.

# Today’s Talk

**Part 1**

We reframe the problem of distributed consensus.

**Part 2**

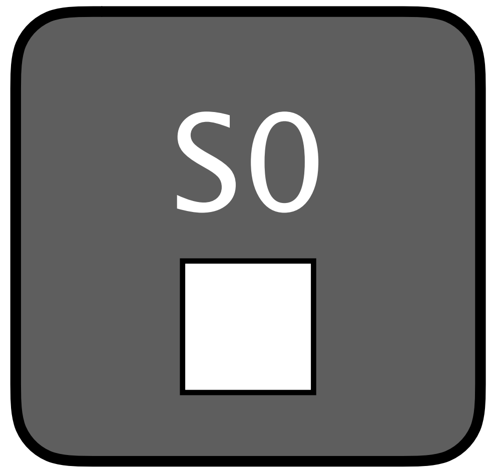
We generalise the Paxos algorithm.

**Part 3**

We introduce the All aboard algorithm.

# **Part 1**
# Distributed consensus using write-once registers

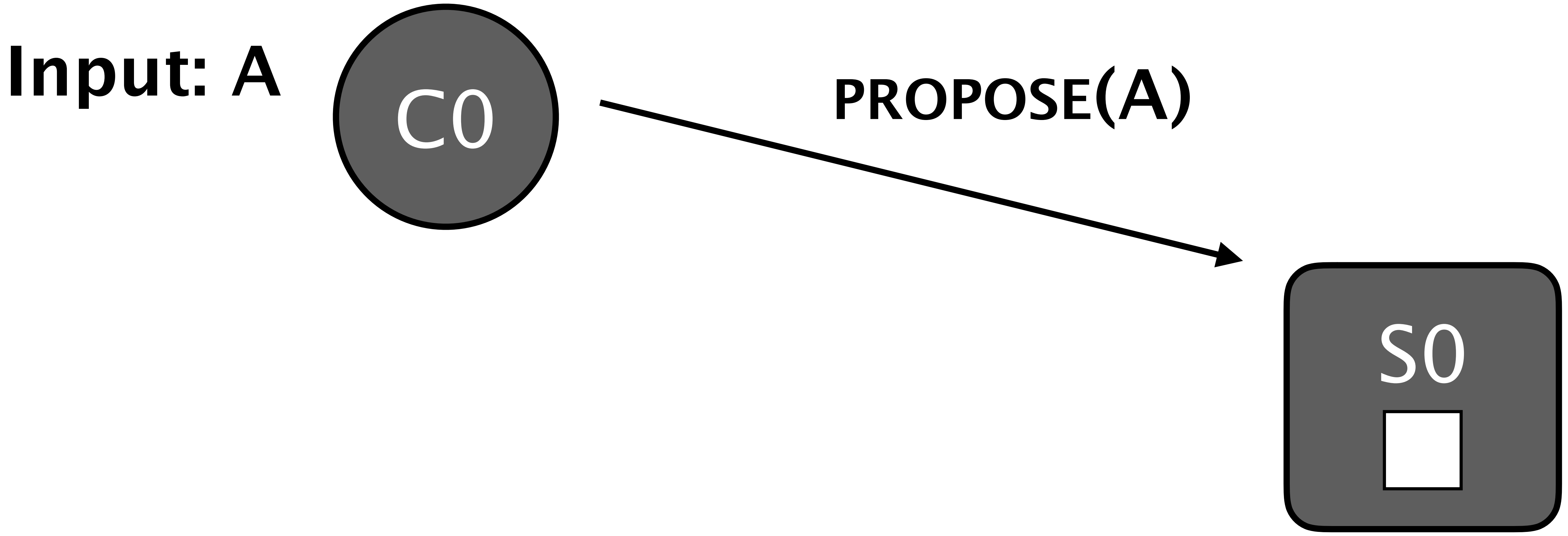
# Single server



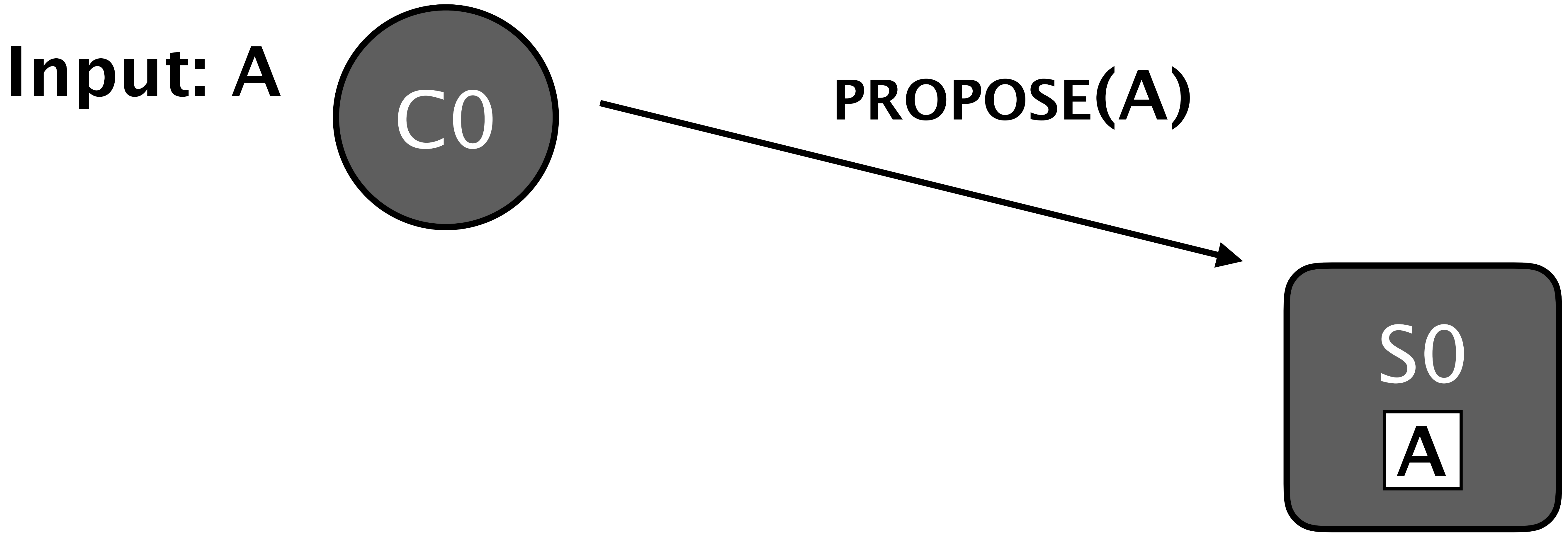
# Single server

# Single server

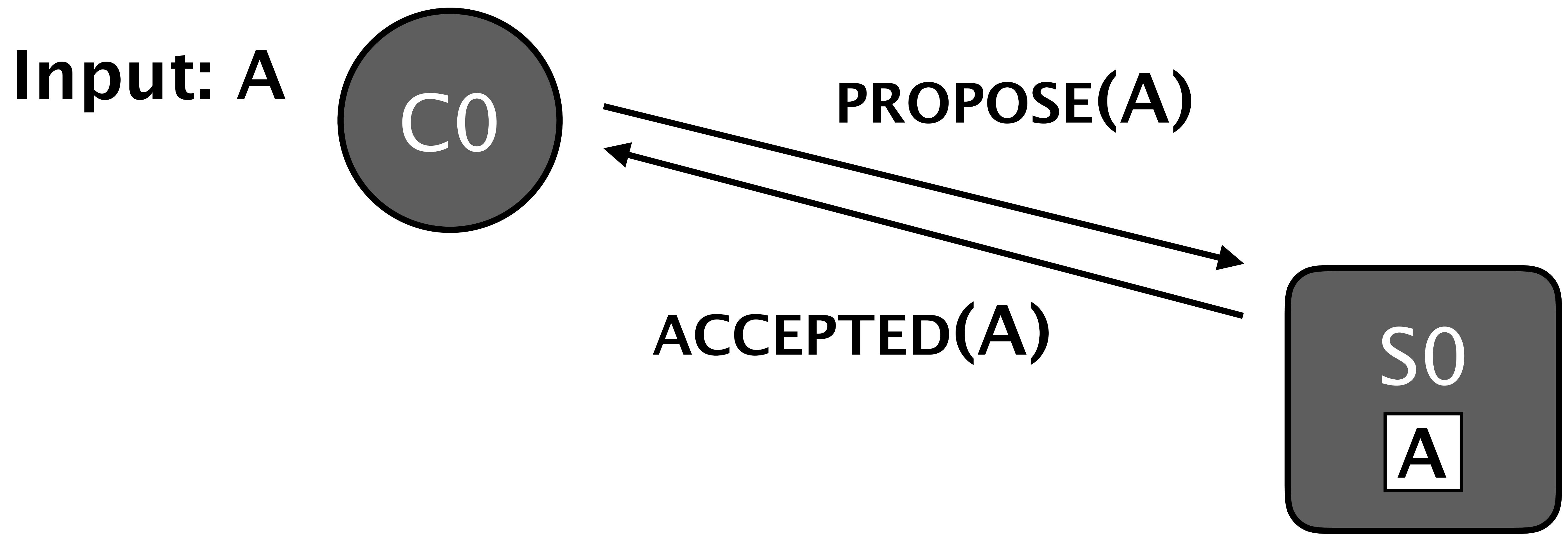
# Single server



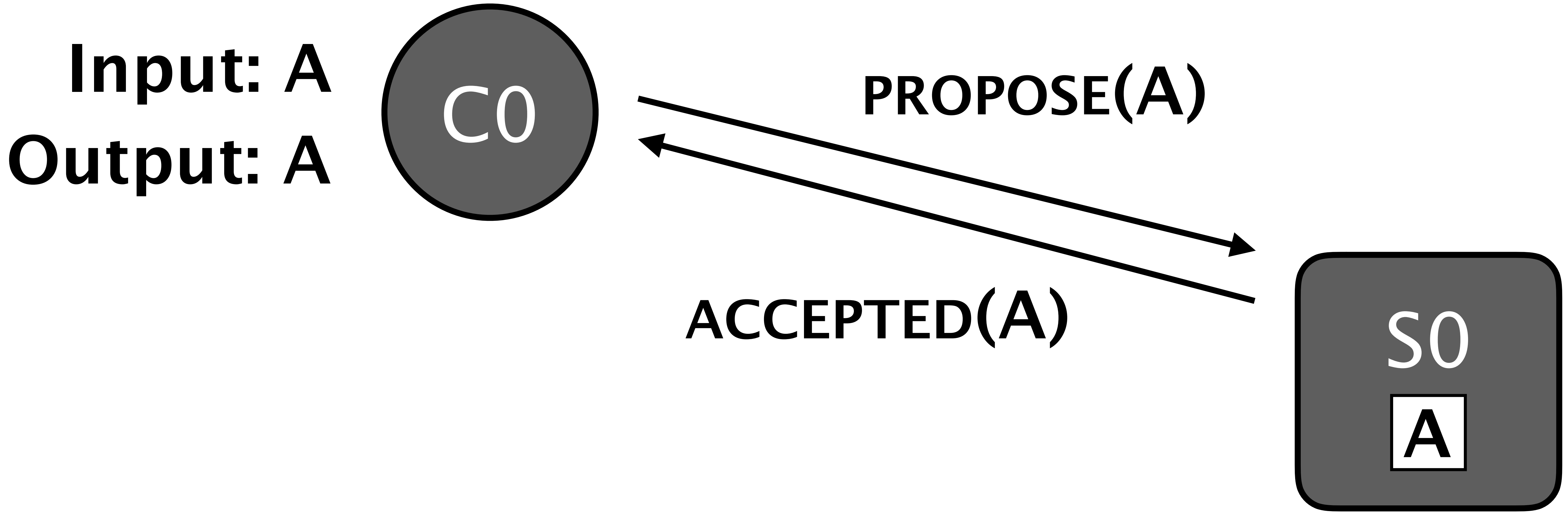
# Single server



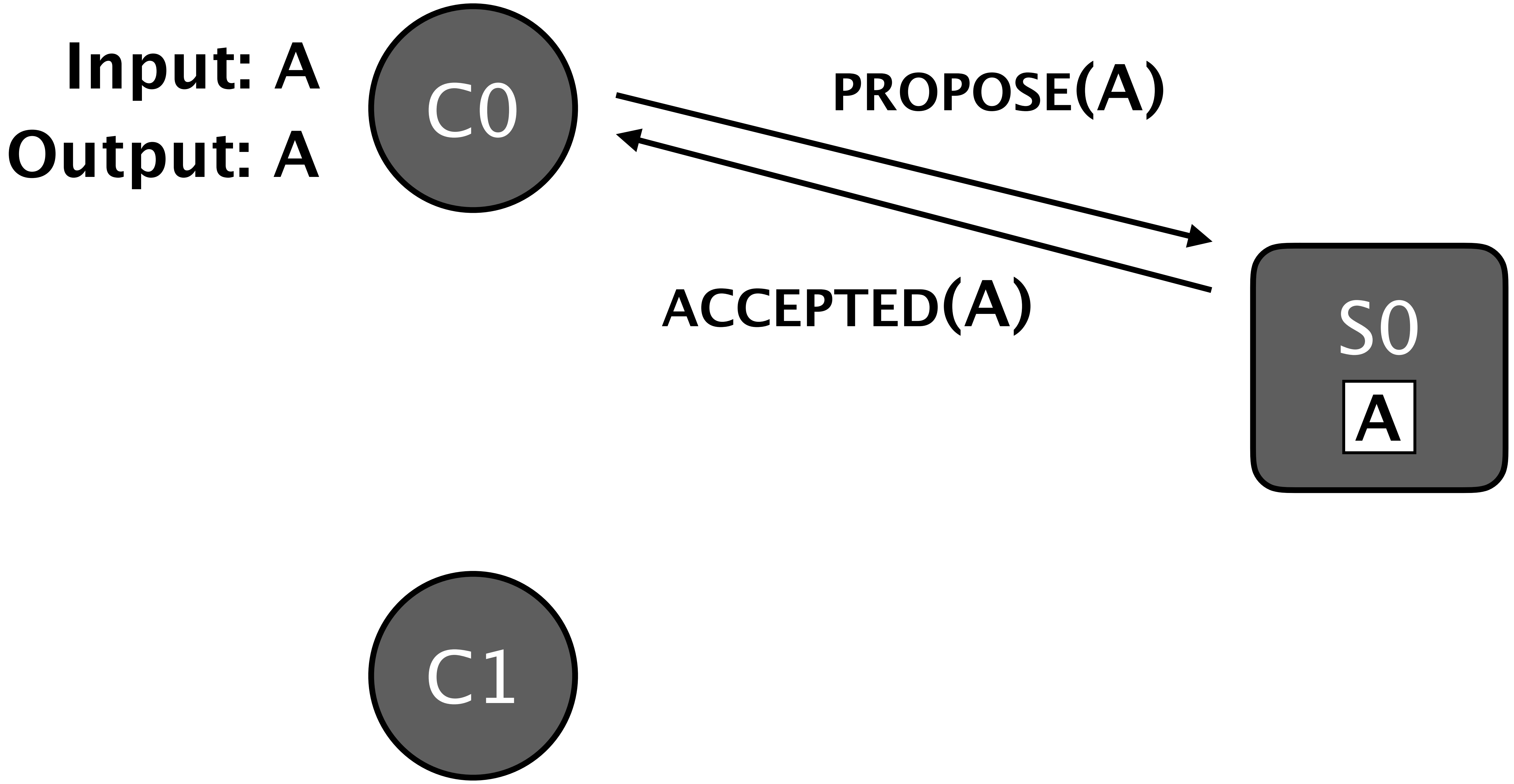
# Single server



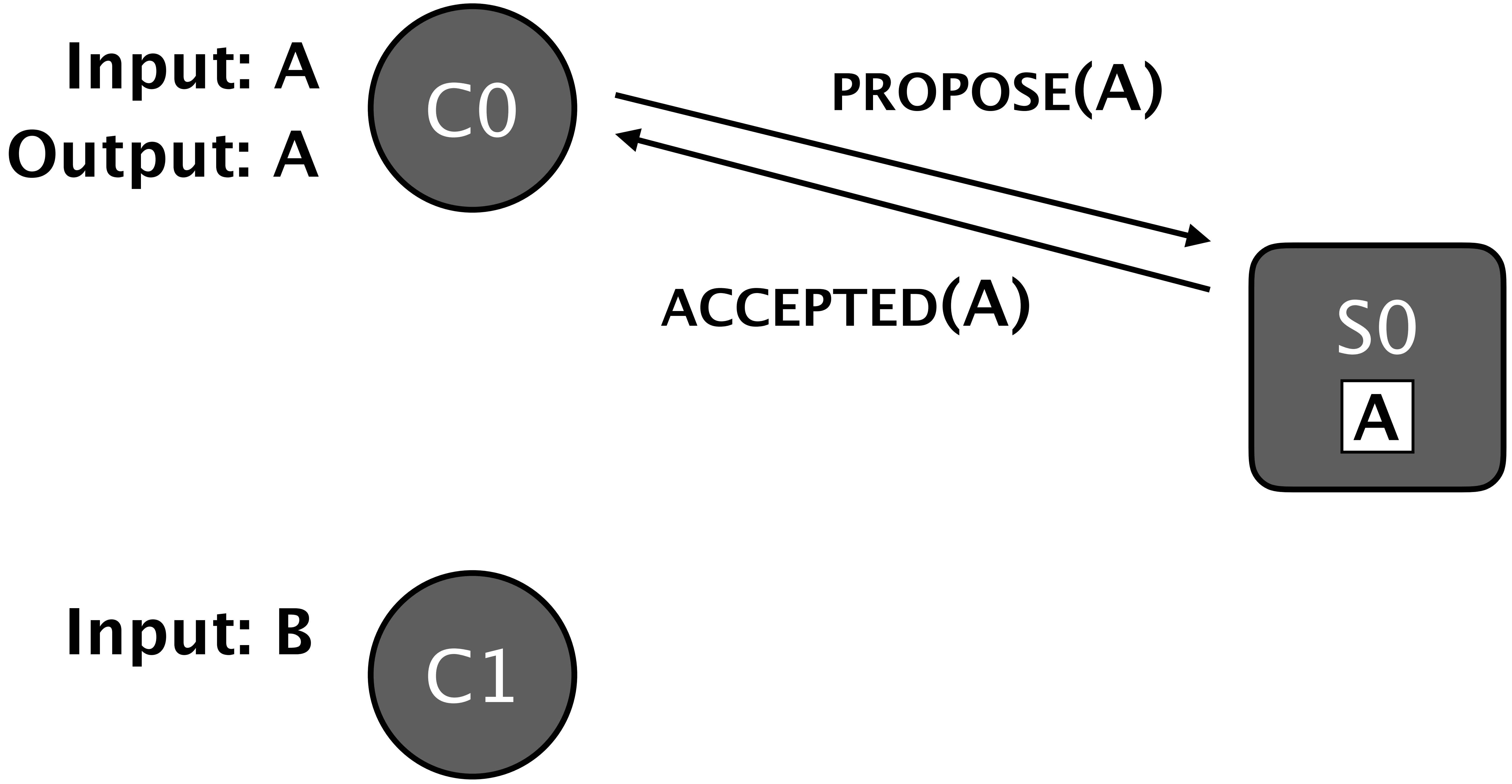
# Single server



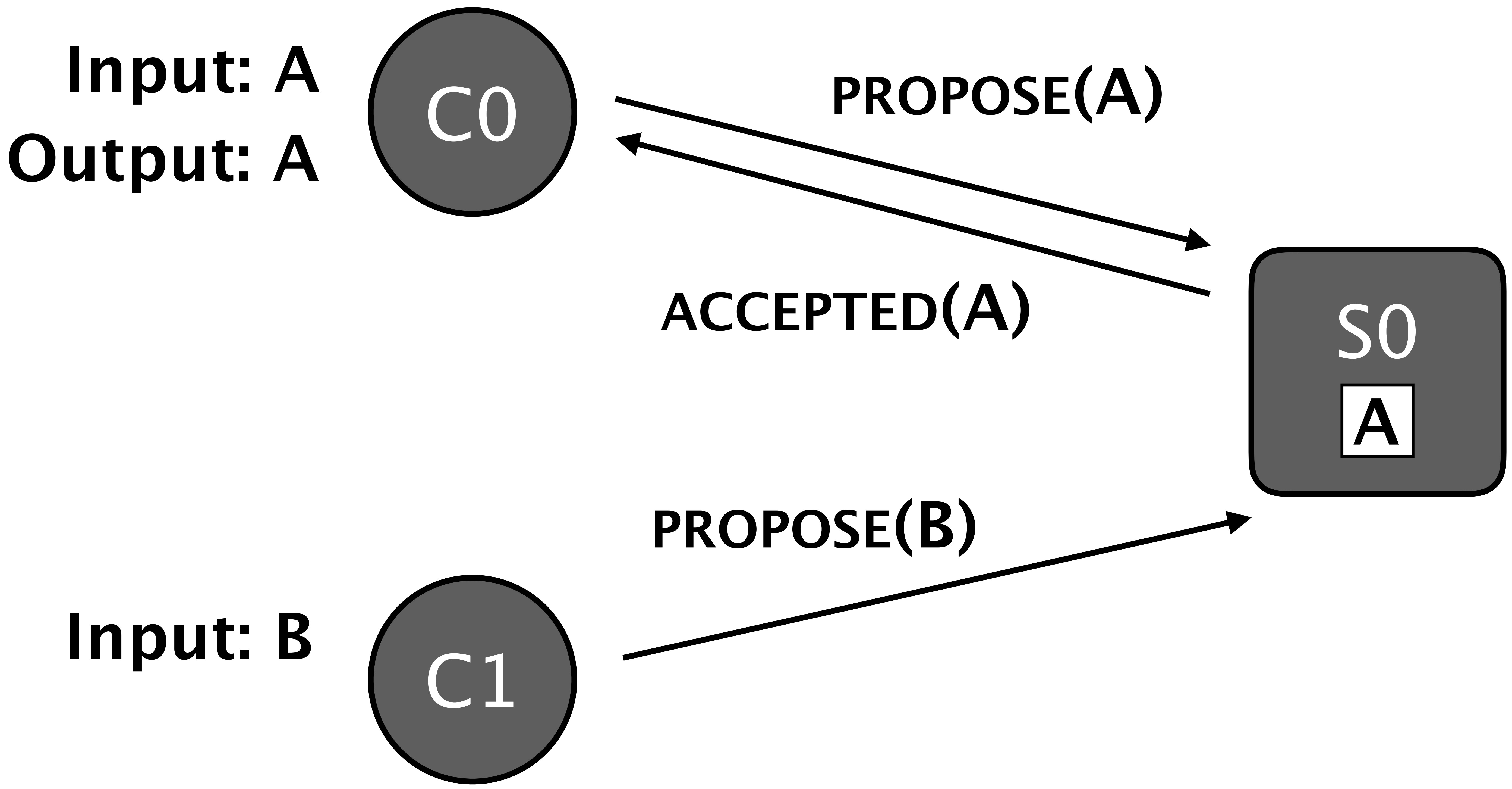
# Single server



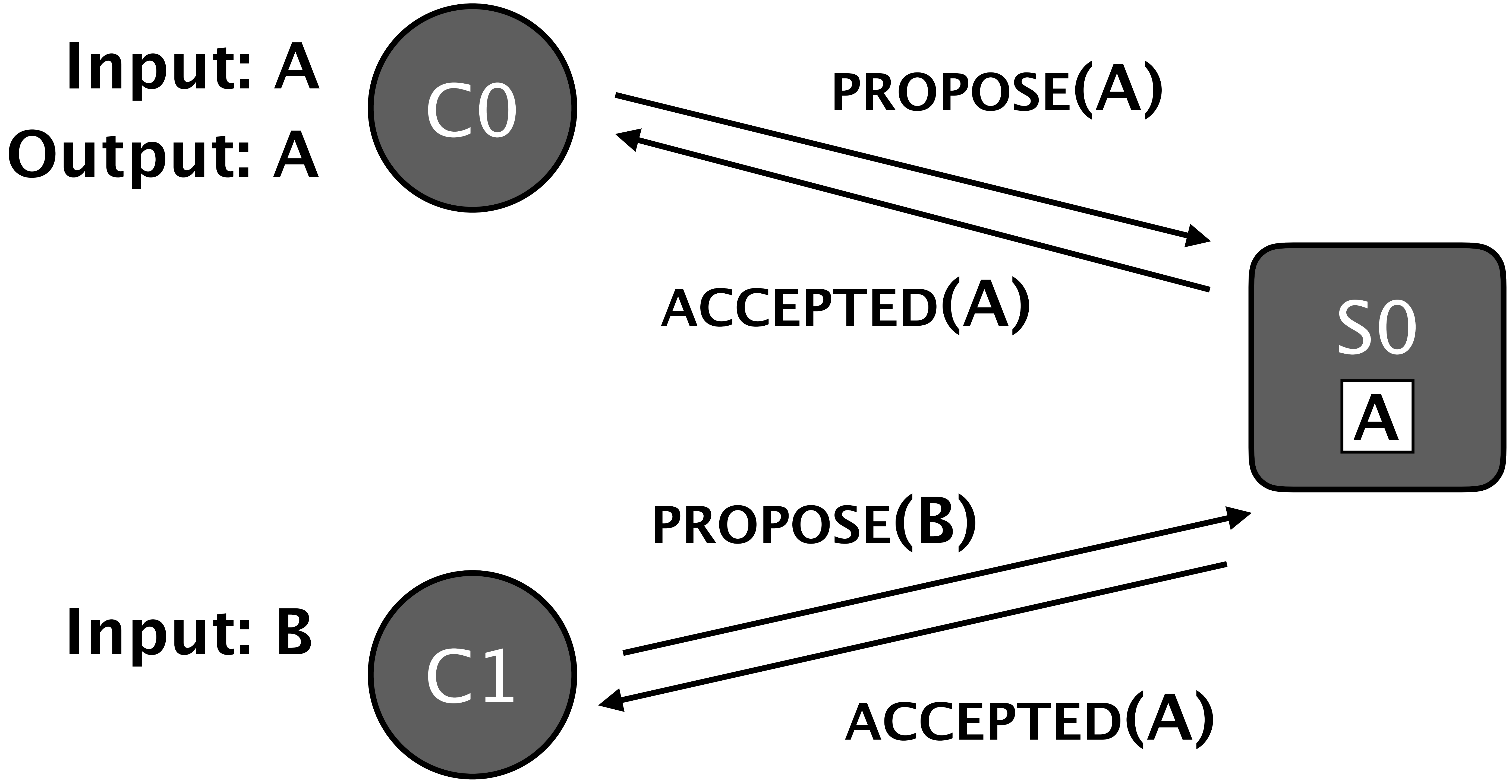
# Single server



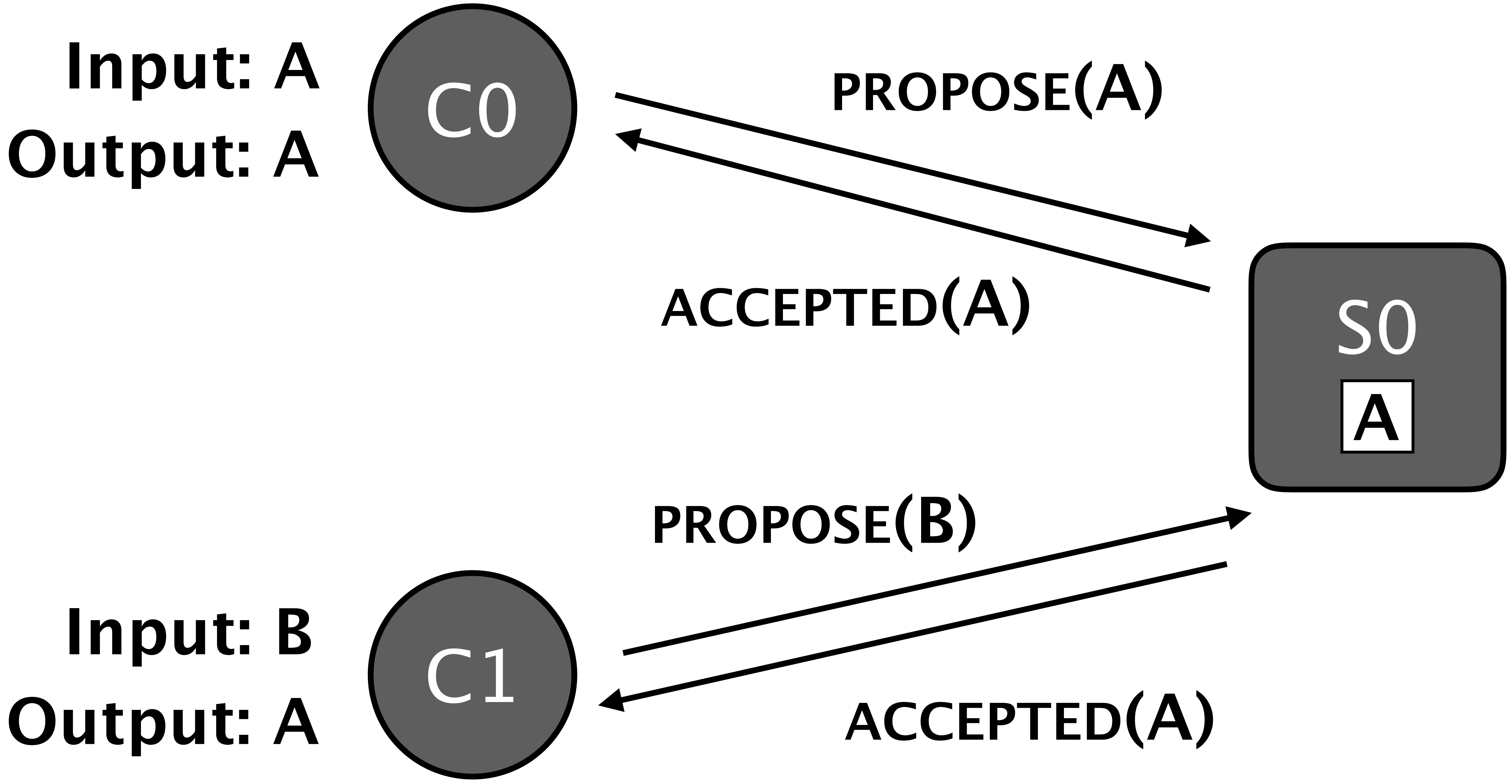
# Single server



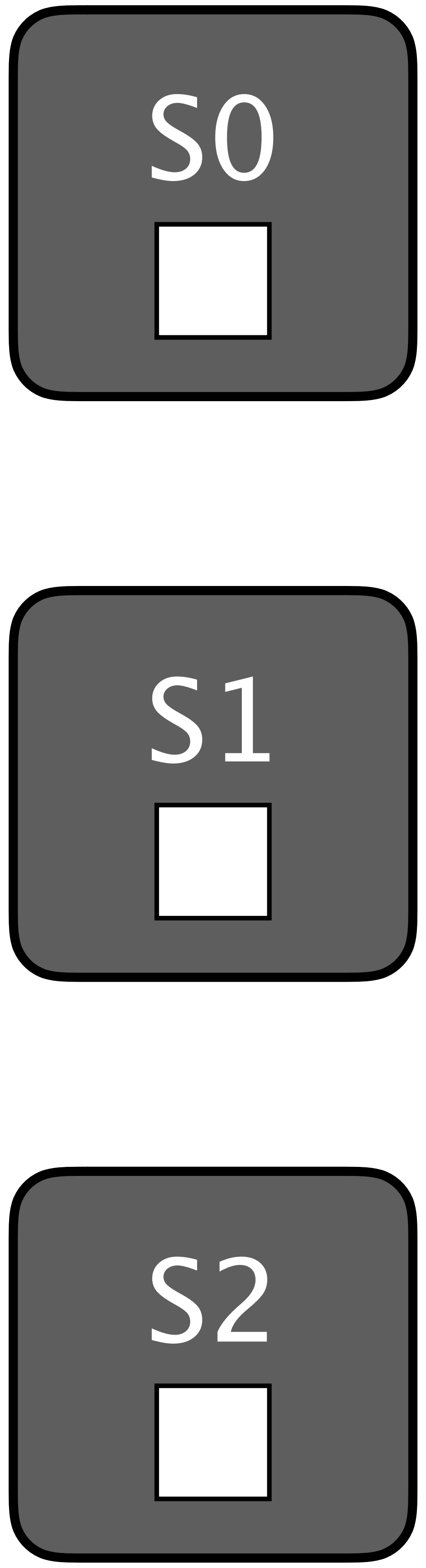
# Single server



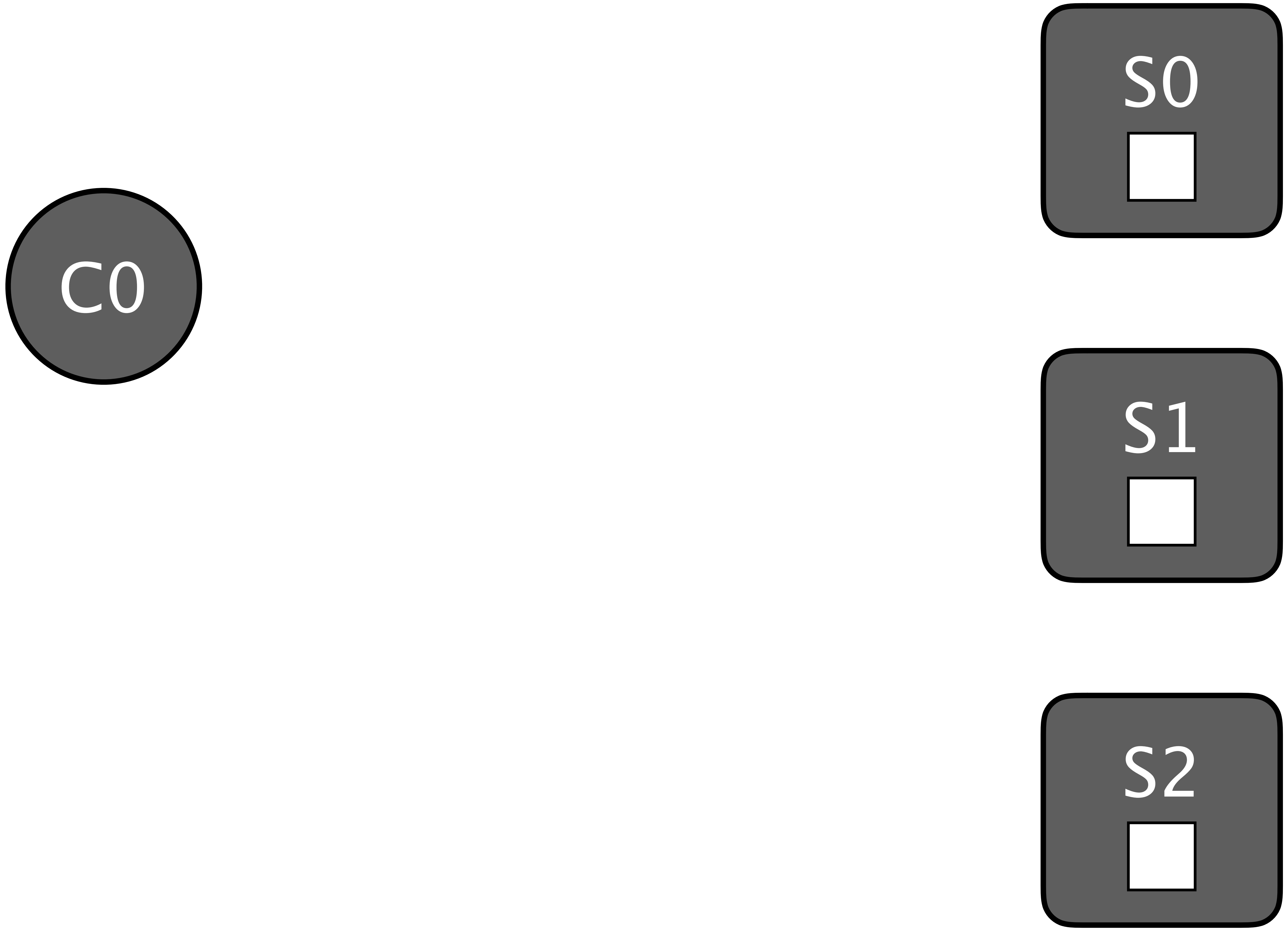
# Single server



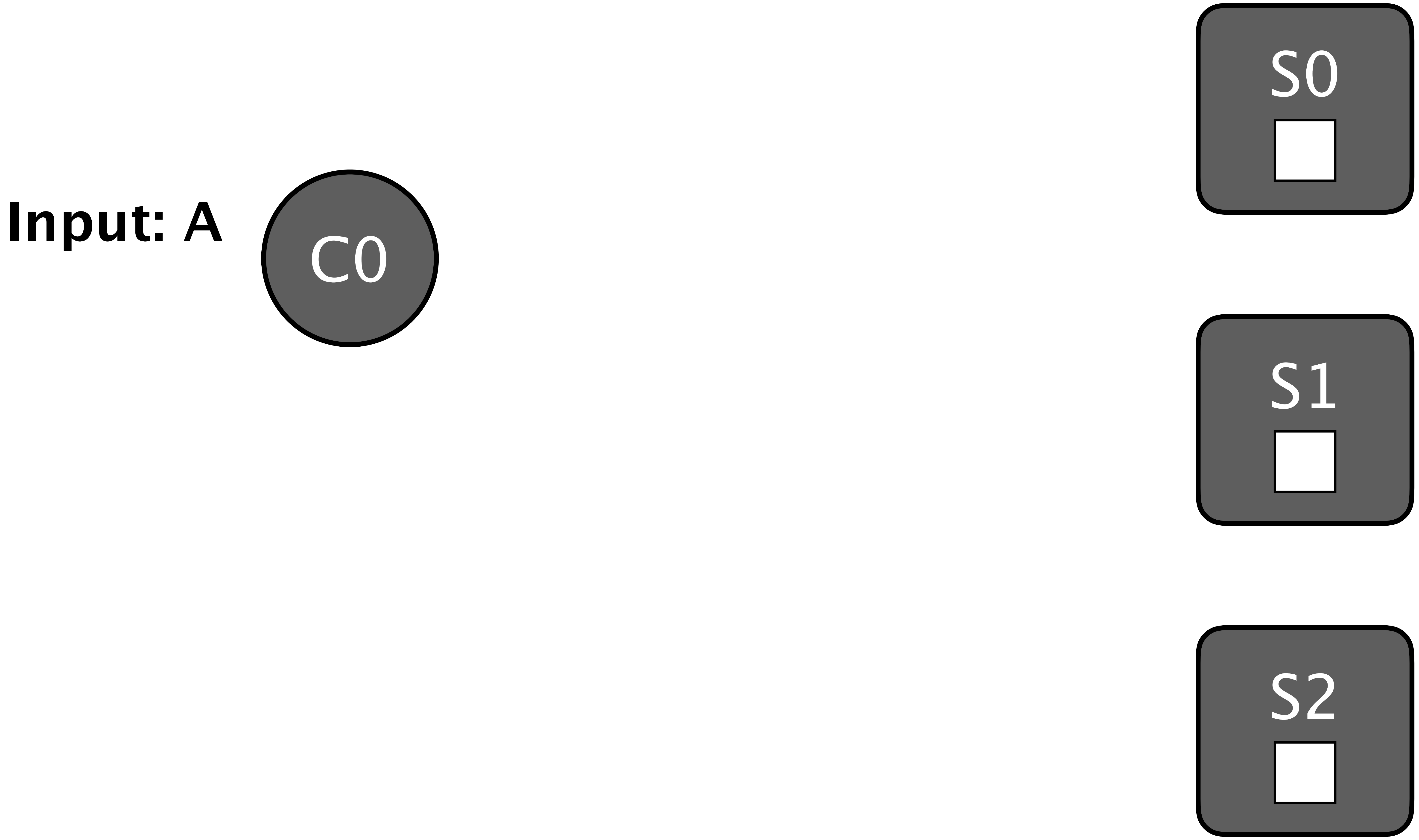
# Multiple servers



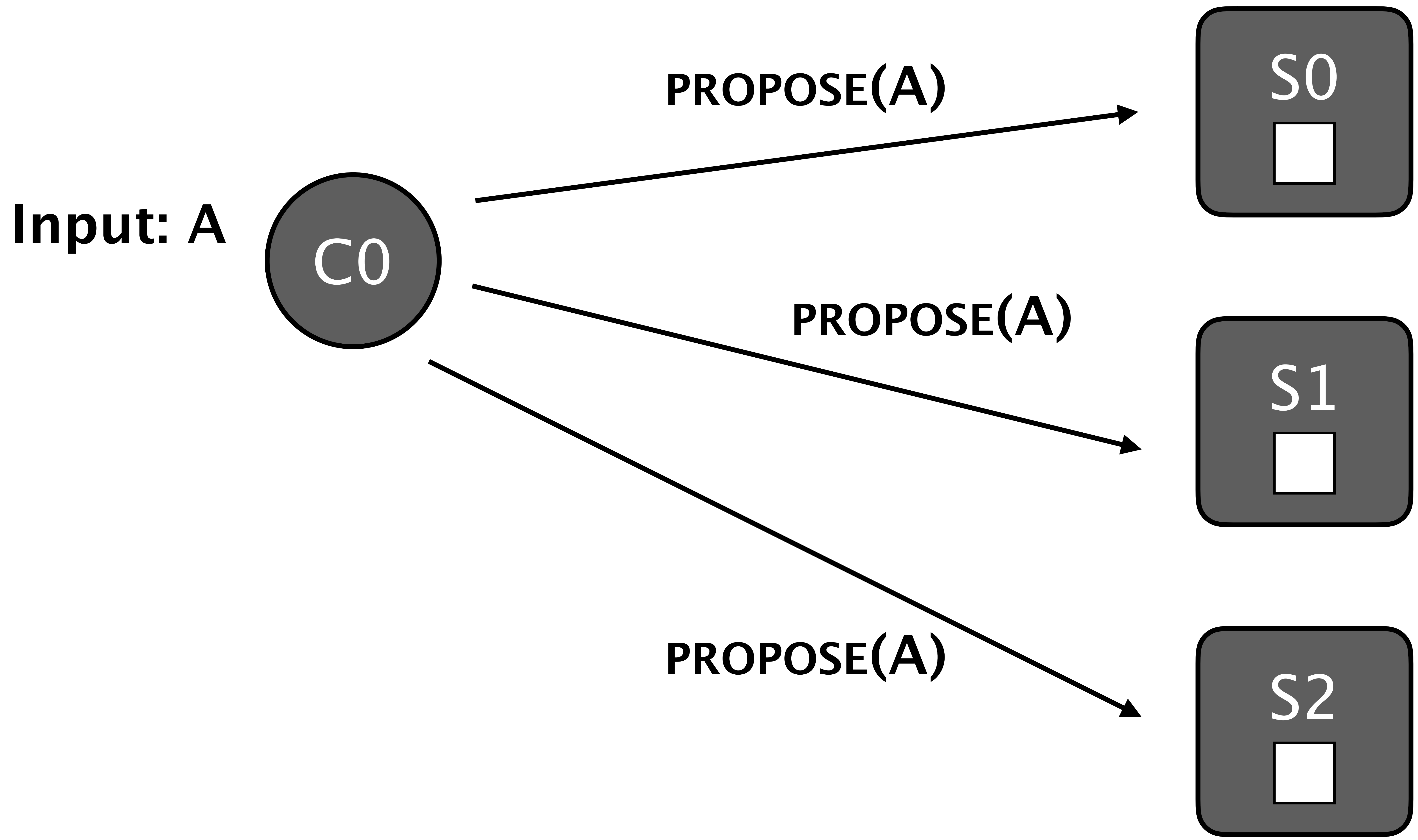
# Multiple servers



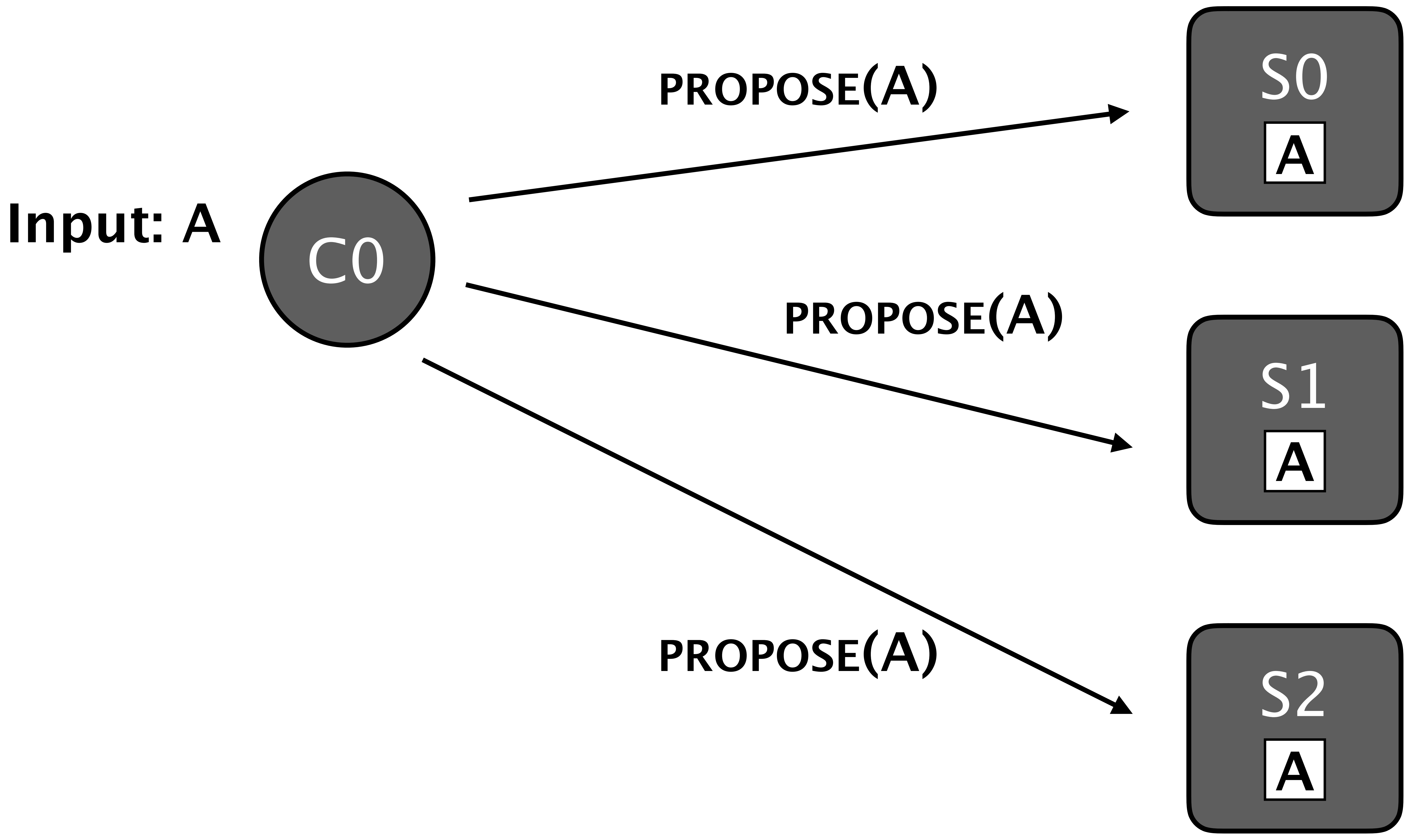
# Multiple servers



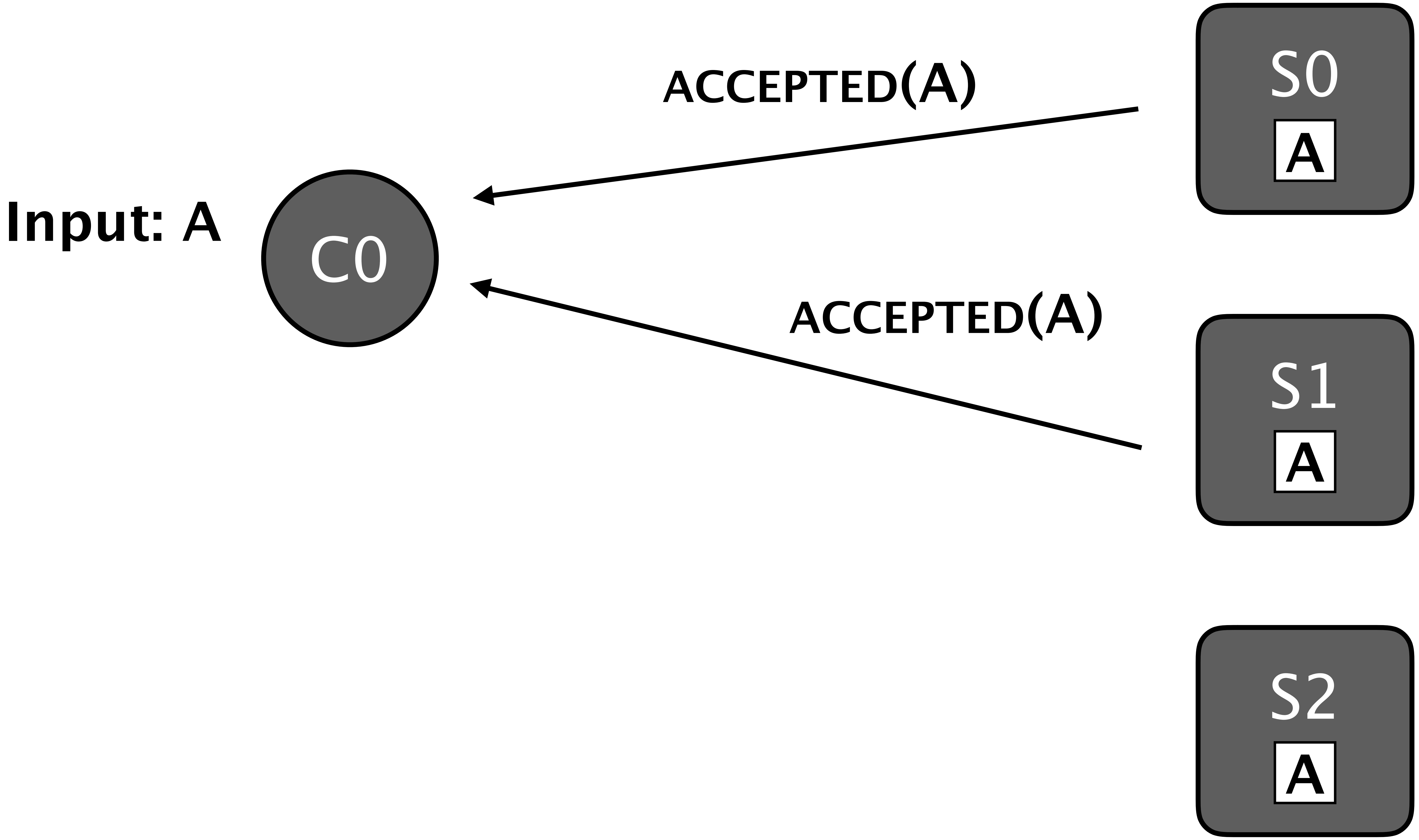
# Multiple servers



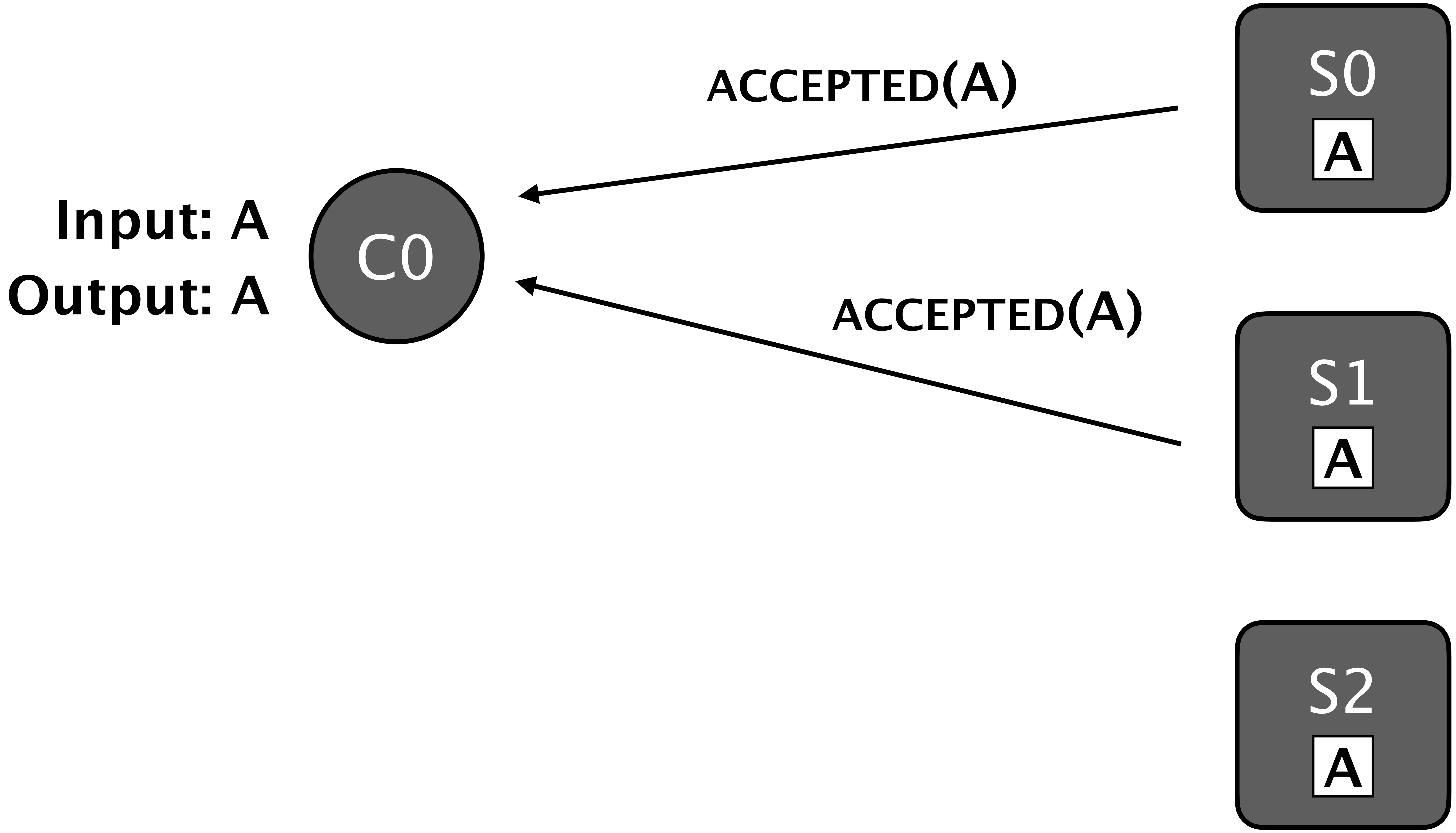
# Multiple servers



# Multiple servers



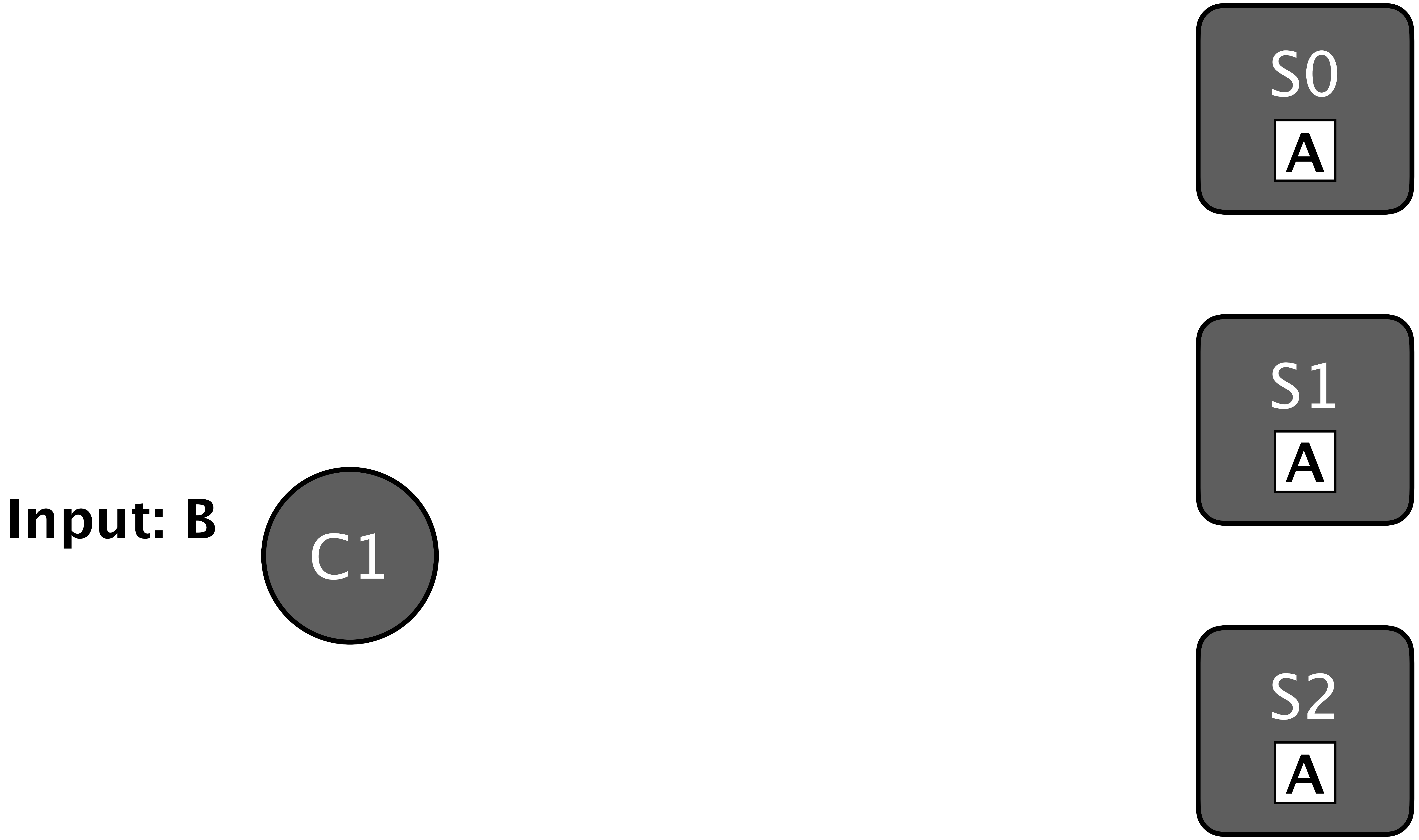
# Multiple servers



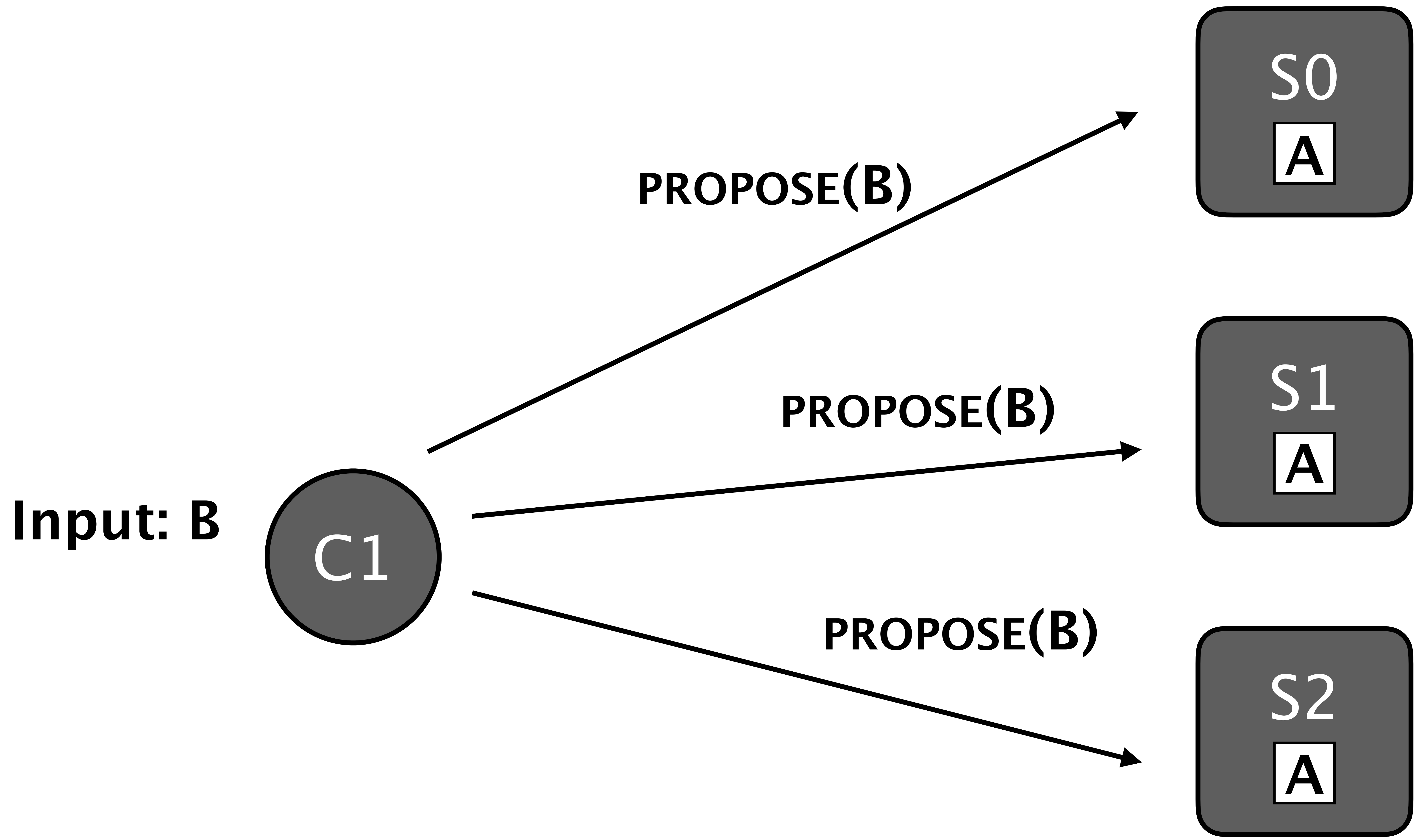
# Multiple servers

# Multiple servers

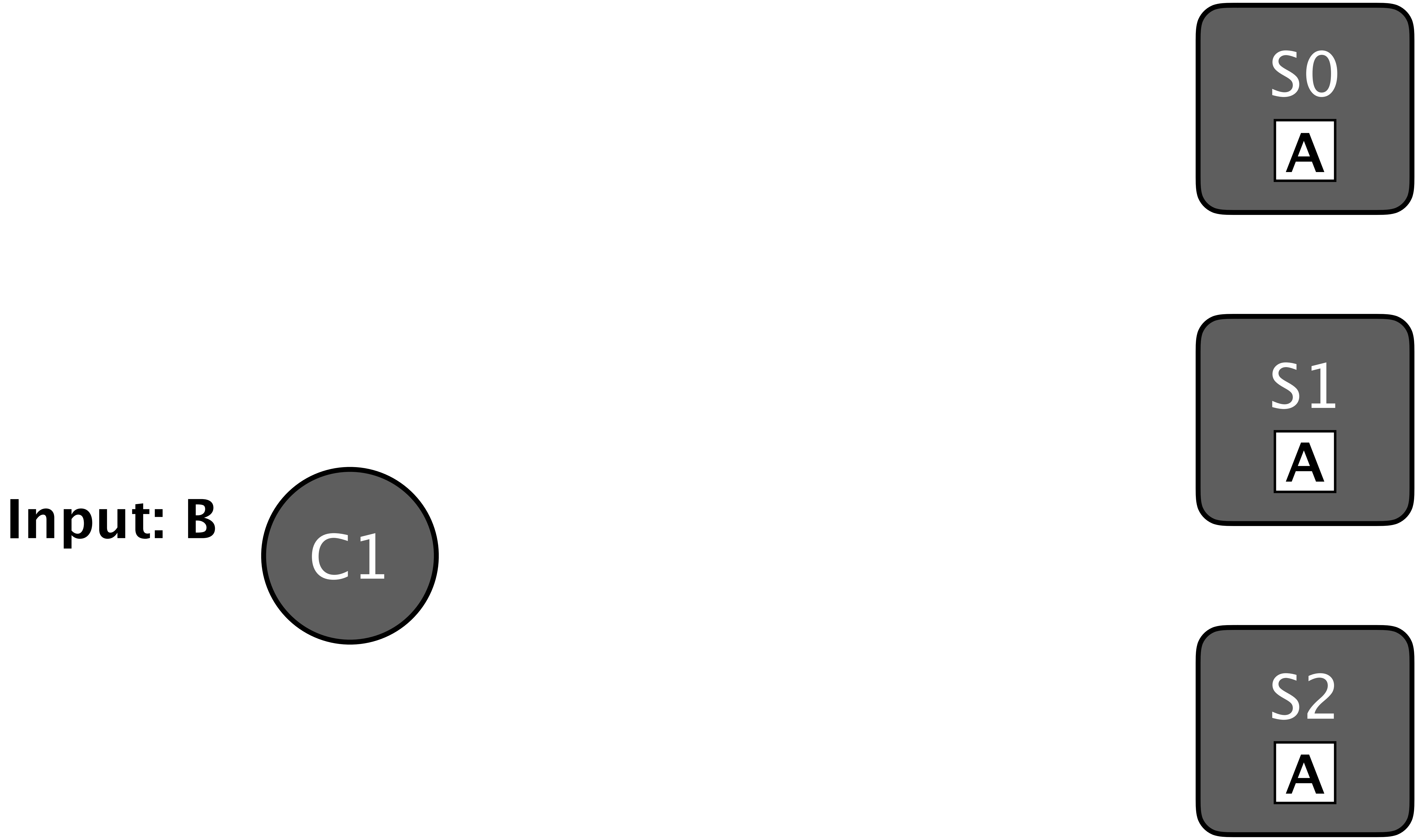
# Multiple servers



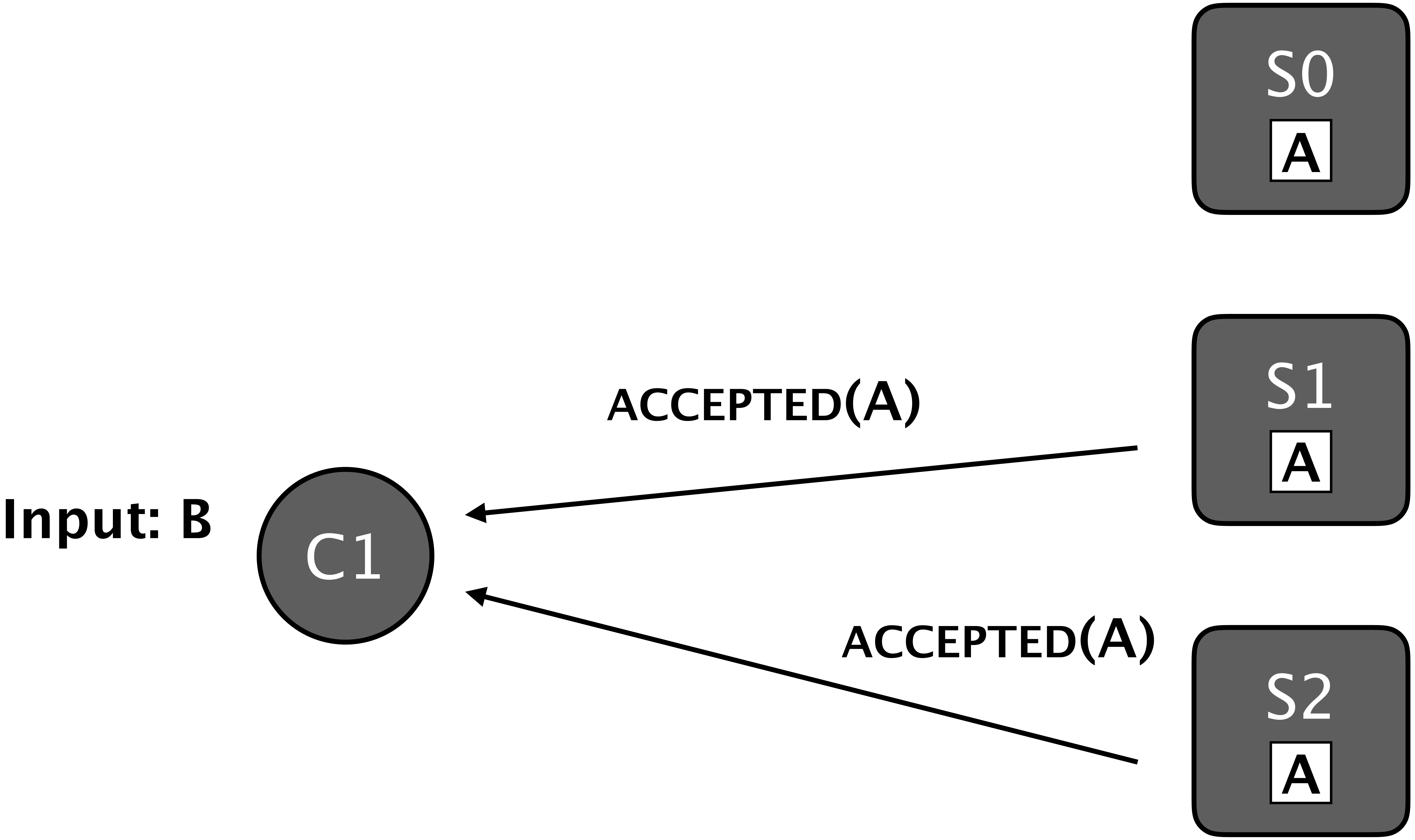
# Multiple servers



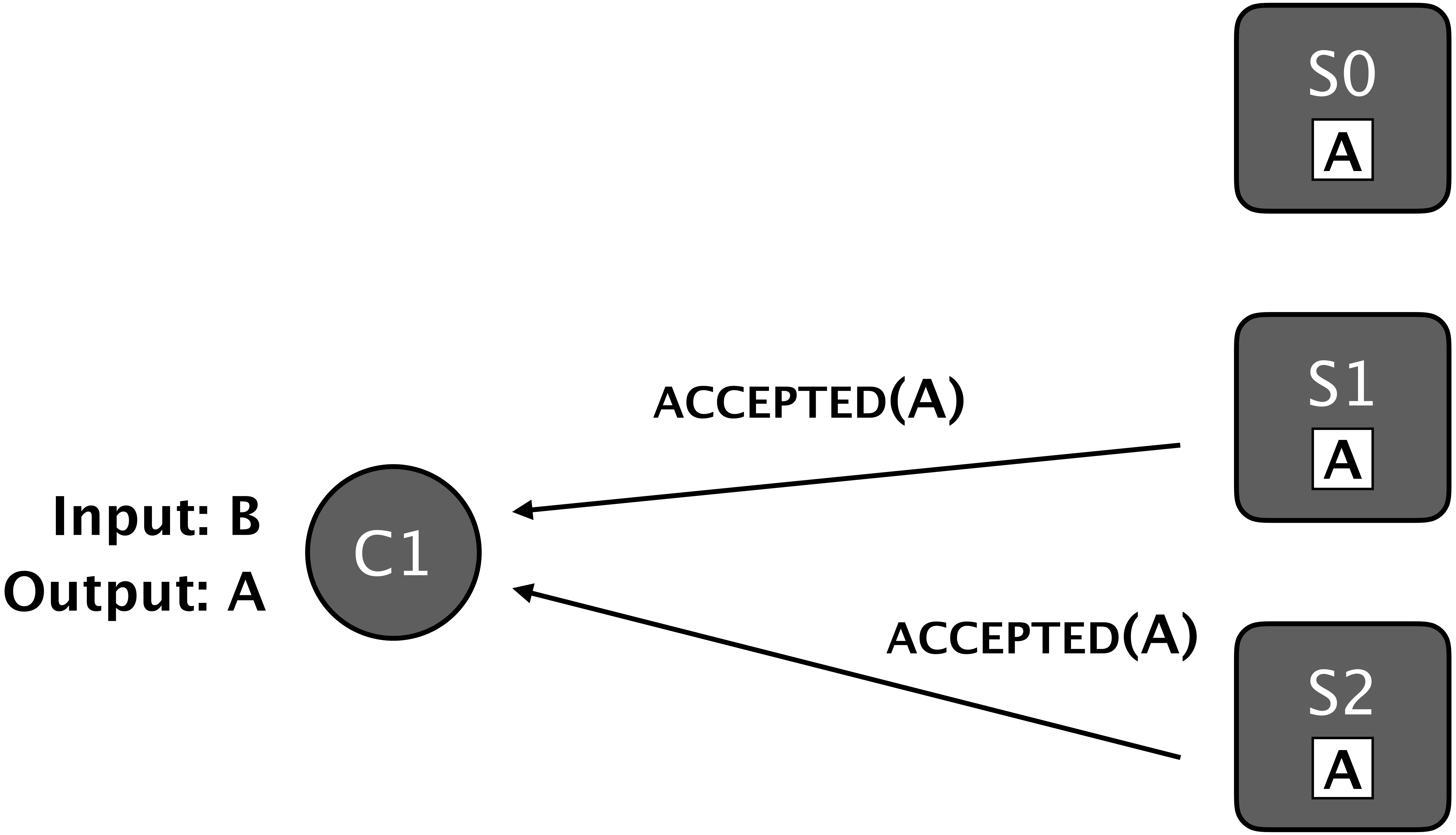
# Multiple servers



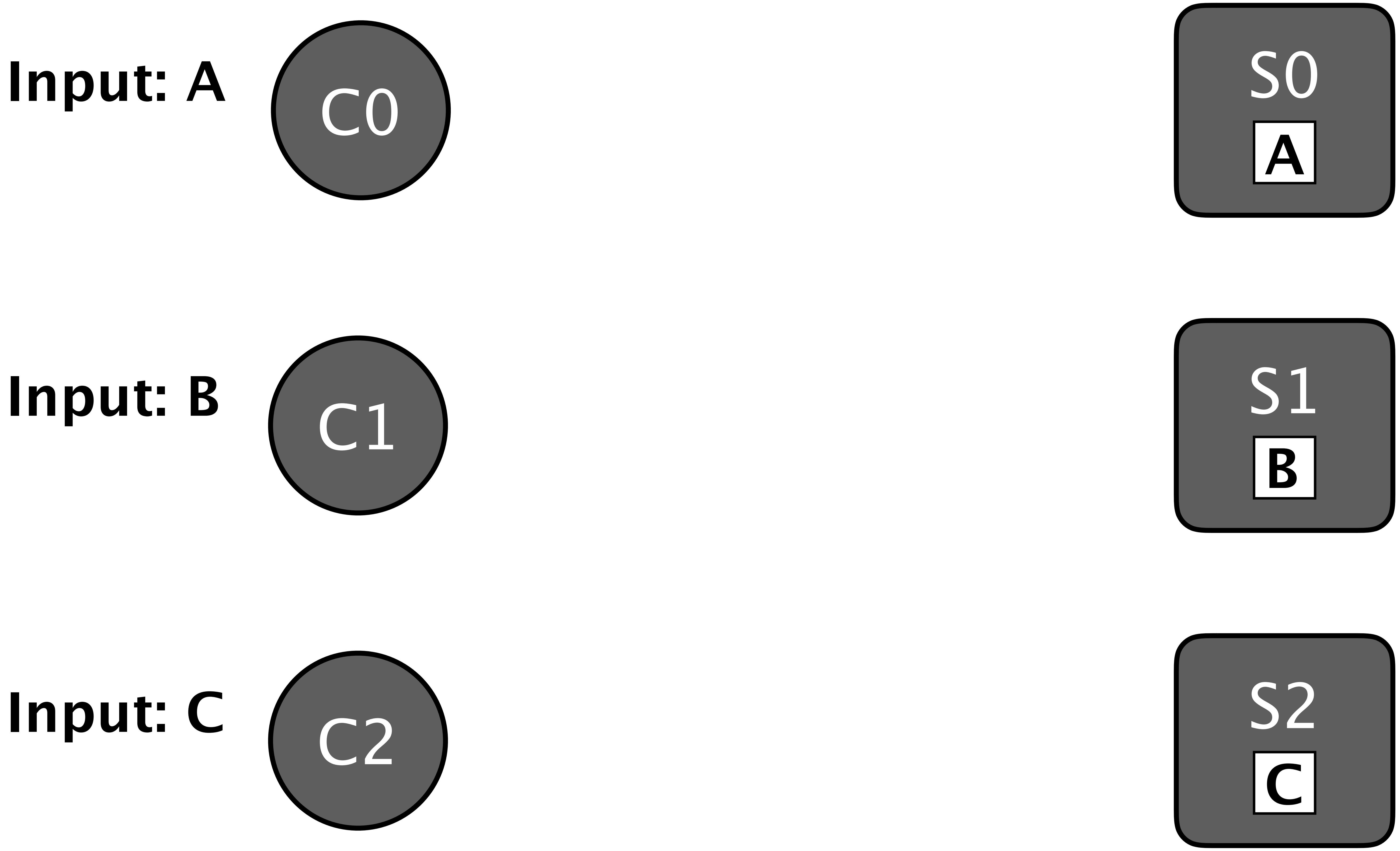
# Multiple servers



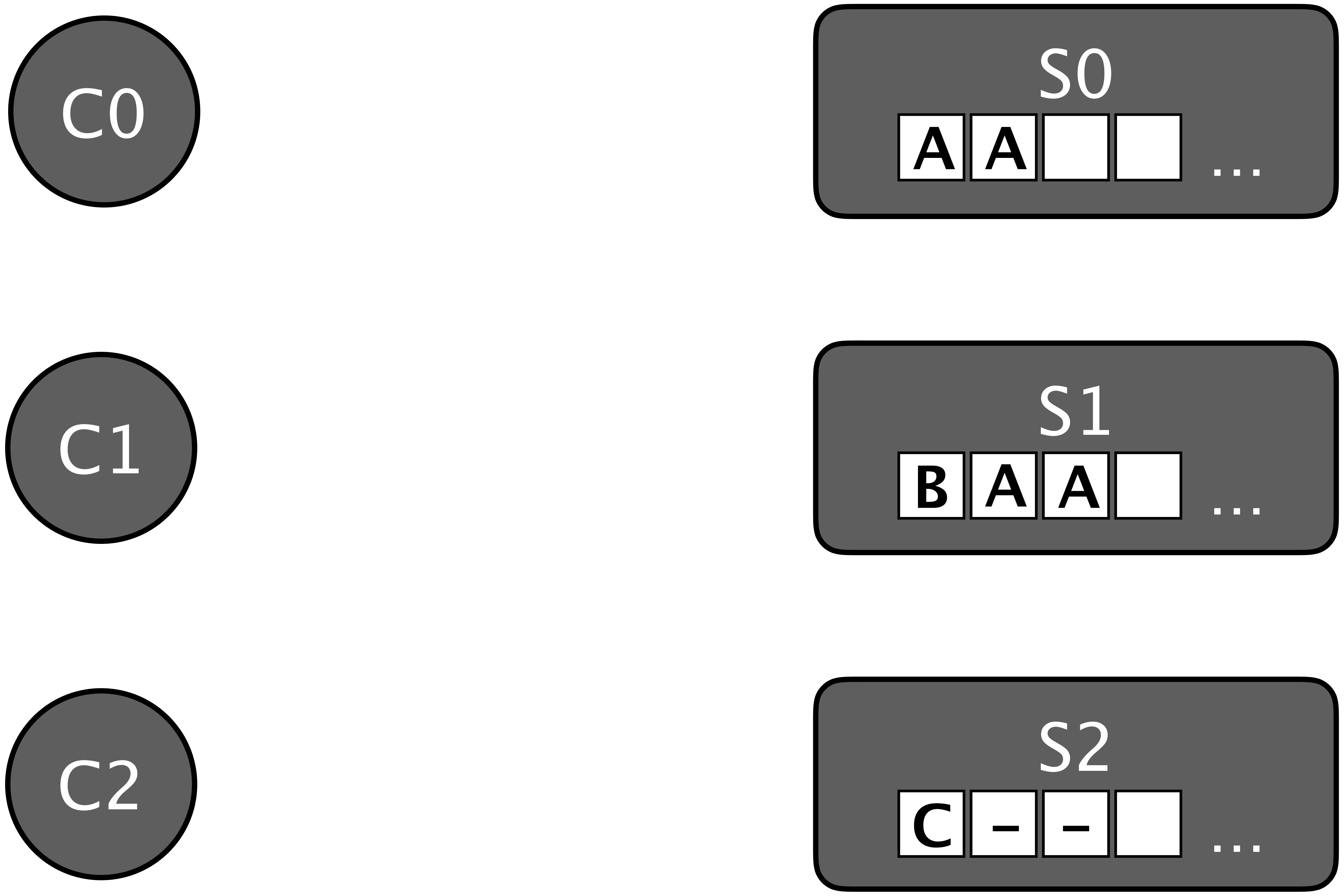
# Multiple servers



# Split Votes



# Multiple write-once registers



# Example state table

# Example state table

| | S0 | S1 | S2 |
|---|---|---|---|
| **R0** | A | B | C |
| **R1** | A | A | – |
| **R2** | | A | – |

# Example state table

Servers

| | S0 | S1 | S2 |
|---|---|---|---|
| **R0** | A | B | C |
| **R1** | A | A | – |
| **R2** | | A | – |

Register sets

Nil value

# Making decisions

# Making decisions

A value is **decided** when it has been written to the same register on a subset of servers, known as a **quorum**.

# Example quorum table

| | Quorums |
|---|---|
| **R0** | {S0,S1} |
| **R1** | {S2,S3} |
| **R2+** | {S0,S1} {S2,S3} |

# Putting it all together

# Putting it all together

| | Quorums |
|---|---|
| **R0+** | {S0,S1} {S1,S2} {S0,S2} |

# Putting it all together

| | Quorums |
|---|---|
| **R0+** | {S0,S1} {S1,S2} {S0,S2} |

| | S0 | S1 | S2 |
|---|---|---|---|
| **R0** | – | A | A |
| **R1** | – | A | |

# Putting it all together

| | Quorums |
|---|---|
| **R0+** | {S0,S1} {S1,S2} {S0,S2} |

| | Quorums |
|---|---|
| **R0** | {S0,S1,S2,S3} |
| **R1+** | {S0,S1} {S2,S3} |

| | S0 | S1 | S2 |
|---|---|---|---|
| **R0** | – | A | A |
| **R1** | – | A | |

# Putting it all together

| | Quorums |
|---|---|
| R0+ | {S0,S1} {S1,S2} {S0,S2} |

| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | A | A |
| R1 | – | A | |

| | Quorums |
|---|---|
| R0 | {S0,S1,S2,S3} |
| R1+ | {S0,S1} {S2,S3} |

| | S0 | S1 | S2 | S3 |
|---|---|---|---|---|
| R0 | B | B | | A |
| R1 | – | – | A | A |
| R2 | A | A | | |

# We can decide multiple values

# We can decide multiple values

| | Quorums |
|---|---|
| **R0** | {S0,S1,S2,S3} |
| **R1+** | {S0,S1} {S2,S3} |

# We can decide multiple values

| | Quorums |
|---|---|
| **R0** | {S0,S1,S2,S3} |
| **R1+** | {S0,S1} {S2,S3} |

| | S0 | S1 | S2 | S3 |
|---|---|---|---|---|
| **R0** | – | A | A | |
| **R1** | C | C | A | A |

# We can decide multiple values

| | Quorums |
|---|---|
| **R0** | {S0,S1,S2,S3} |
| **R1+** | {S0,S1} {S2,S3} |

| | Quorums |
|---|---|
| **R0+** | {S0,S1} {S1,S2} {S0,S2} |

| | S0 | S1 | S2 | S3 |
|---|---|---|---|---|
| **R0** | – | A | A | |
| **R1** | C | C | A | A |

# We can decide multiple values

| | Quorums |
|---|---|
| R0 | {S0,S1,S2,S3} |
| R1+ | {S0,S1} {S2,S3} |

| | Quorums |
|---|---|
| R0+ | {S0,S1} {S1,S2} {S0,S2} |

| | S0 | S1 | S2 | S3 |
|---|---|---|---|---|
| R0 | – | A | A | |
| R1 | C | C | A | A |

| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | C | A | A |
| R1 | B | B | A |

# Safety

Before a client writes a value to register i it must ensure that no other values could be decided in register sets 0 to i.

# **Part 2**
# Generalising Paxos

# Safety

Before a client writes a value to register i it must ensure that:

1. No other values could be decided in register set i
2. No other values could be decided in register sets 0 to i-1

# Register allocation rule

Paxos allocates registers to clients round robin and requires clients to write at most one value to each of their allocated registers.

| Client | Registers |
|---|---|
| C0 | R0, R3, … |
| C1 | R1, R4, … |
| C2 | R2, R5, … |

# Safety

Before a client writes a value to register i it must ensure that:

1. No other values could be decided in register set i
2. No other values could be decided in register sets 0 to i−1

**Register allocation rule**

# Value selection rule

Paxos requires clients to read one register from each quorum of register sets 0 to i−1 and ensure that:

1. All of the registers are written, and
2. If any registers contain non-nil values, the client must write the value from the greatest register.

# Safety

Before a client writes a value to register i it must ensure that:

1. No other values could be decided in register set i — **Register allocation rule**
2. No other values could be decided in register sets 0 to i−1 — **Value selection rule**

# Classic Paxos

Paxos is a two phase consensus algorithm.

- **Phase one** ensures the safety of phase two.
- **Phase two** writes a value to the servers to achieve consensus.

# Classic Paxos

Paxos is a two phase consensus algorithm.

- **Phase one** ensures the safety of phase two.
- **Phase two** writes a value to the servers to achieve consensus.

| | Quorums |
|---|---|
| **R0+** | {S0,S1} {S1,S2} {S0,S2} |

# Classic Paxos – Phase one

# Classic Paxos – Phase one

- The client chooses an allocated register set i and sends **PREPARE(i)** to all servers.

# Classic Paxos – Phase one

- The client chooses an allocated register set i and sends **PREPARE(i)** to all servers.
- Each server writes nil in any unwritten registers from 0 to i–1 and replies with the register number j and value w of the greatest non-nil register using **PROMISED(i,j,w)** or **PROMISED(i)** if no such register exists.

# Classic Paxos - Phase one

- The client chooses an allocated register set i and sends **PREPARE(i)** to all servers.
- Each server writes nil in any unwritten registers from 0 to i-1 and replies with the register number j and value w of the greatest non-nil register using **PROMISED(i,j,w)** or **PROMISED(i)** if no such register exists.
- When **PROMISED(i,**...) has been received from a quorum of servers, the client chooses the value v from the greatest register or its own value if none exists.

# Classic Paxos – Phase two

# Classic Paxos – Phase two

- The client sends **PROPOSE(i,v)** to all servers.

# Classic Paxos – Phase two

- The client sends **PROPOSE(i,v)** to all servers.
- Each server checks if register i is unwritten. If so, it writes the value v to register i and replies with **ACCEPTED(i)**.

# Classic Paxos – Phase two

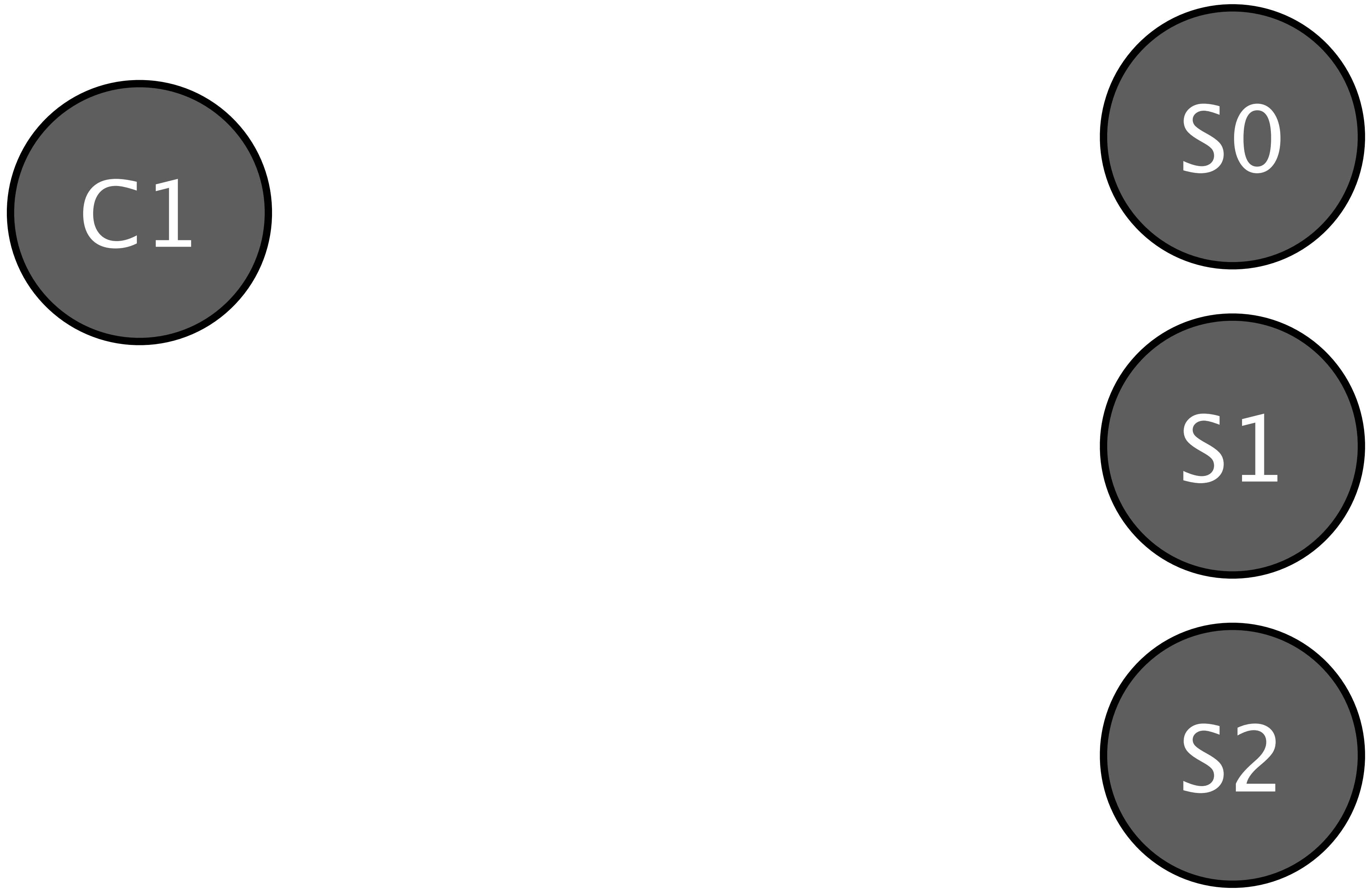
- The client sends **PROPOSE(i,v)** to all servers.
- Each server checks if register i is unwritten. If so, it writes the value v to register i and replies with **ACCEPTED(i)**.
- The client terminates when **ACCEPTED(i)** has been received from a quorum of servers.

# Example – Phase one


| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | | | |
| R1 | | | |
| R2 | | | |
| R3 | | | |

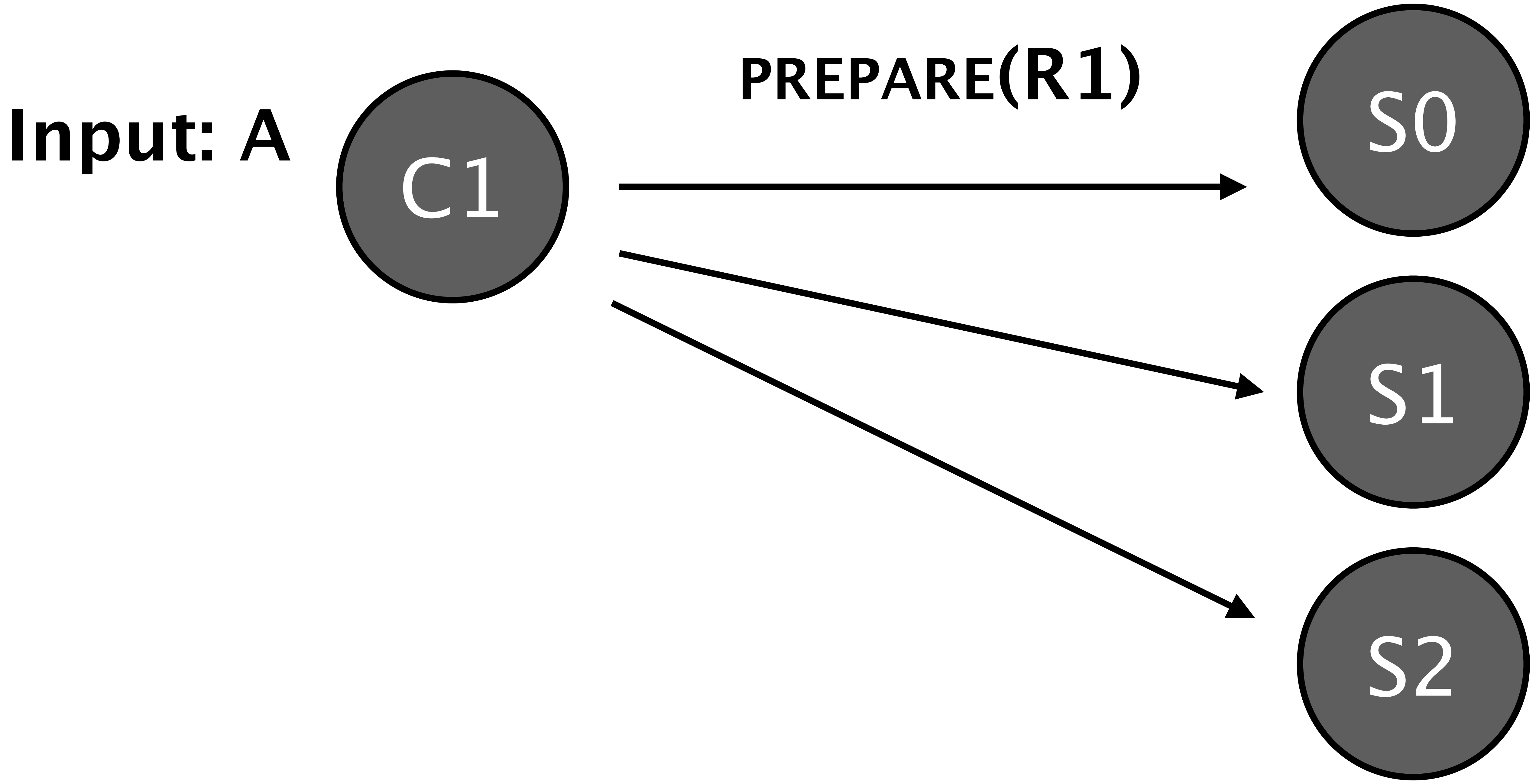
# Example – Phase one

**Input: A**



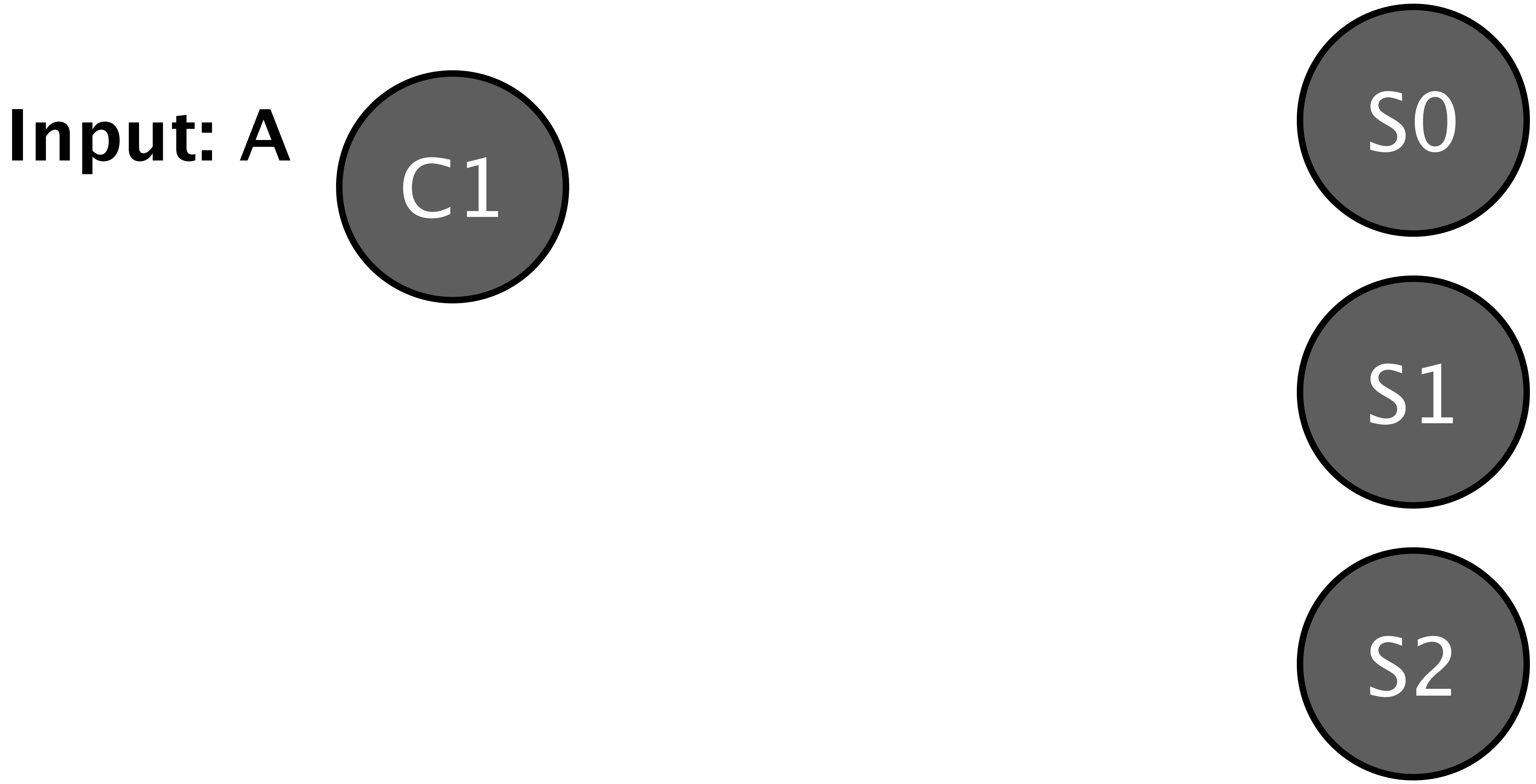

| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | | | |
| R1 | | | |
| R2 | | | |
| R3 | | | |

# Example – Phase one



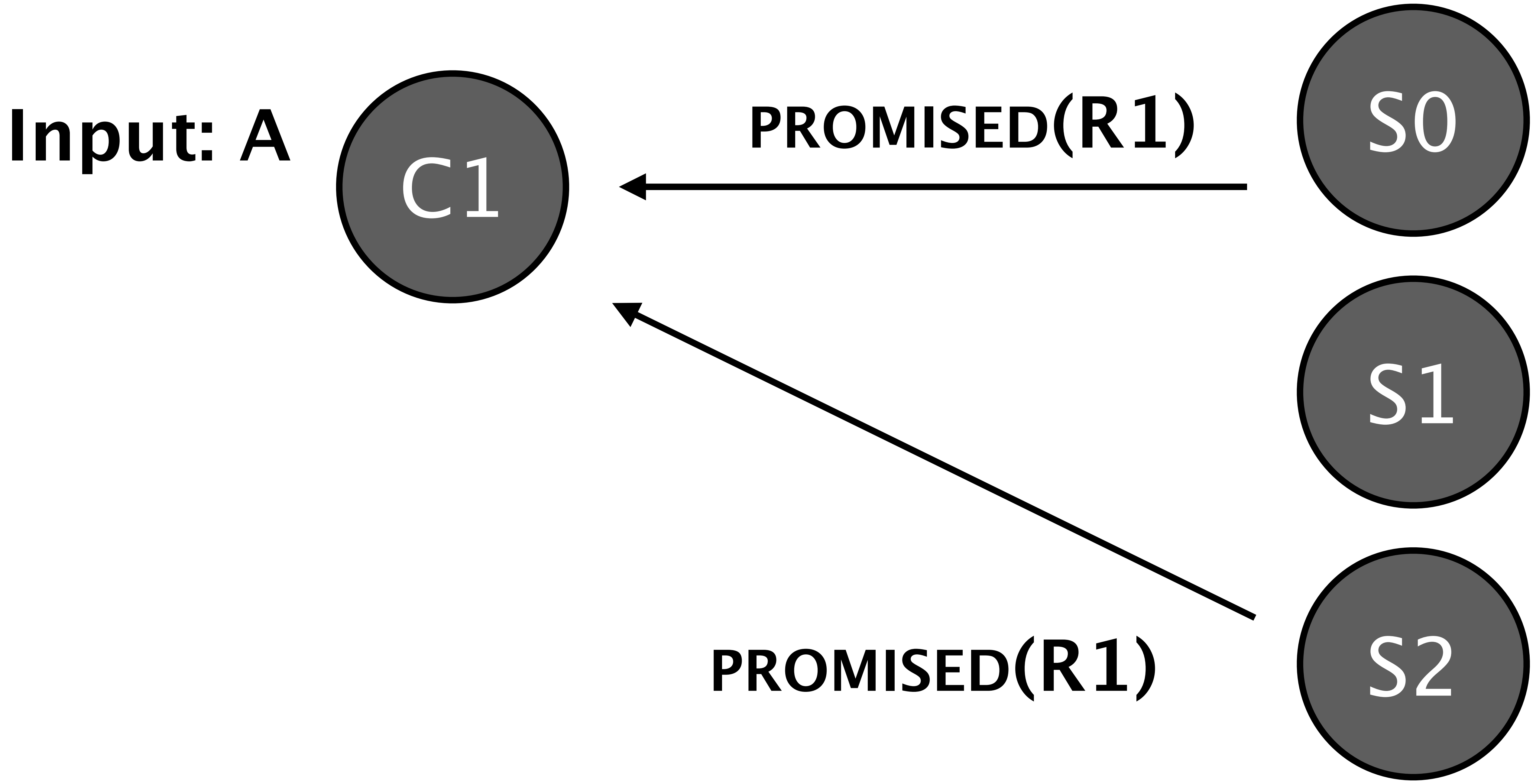

| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | | | |
| R1 | | | |
| R2 | | | |
| R3 | | | |

# Example – Phase one



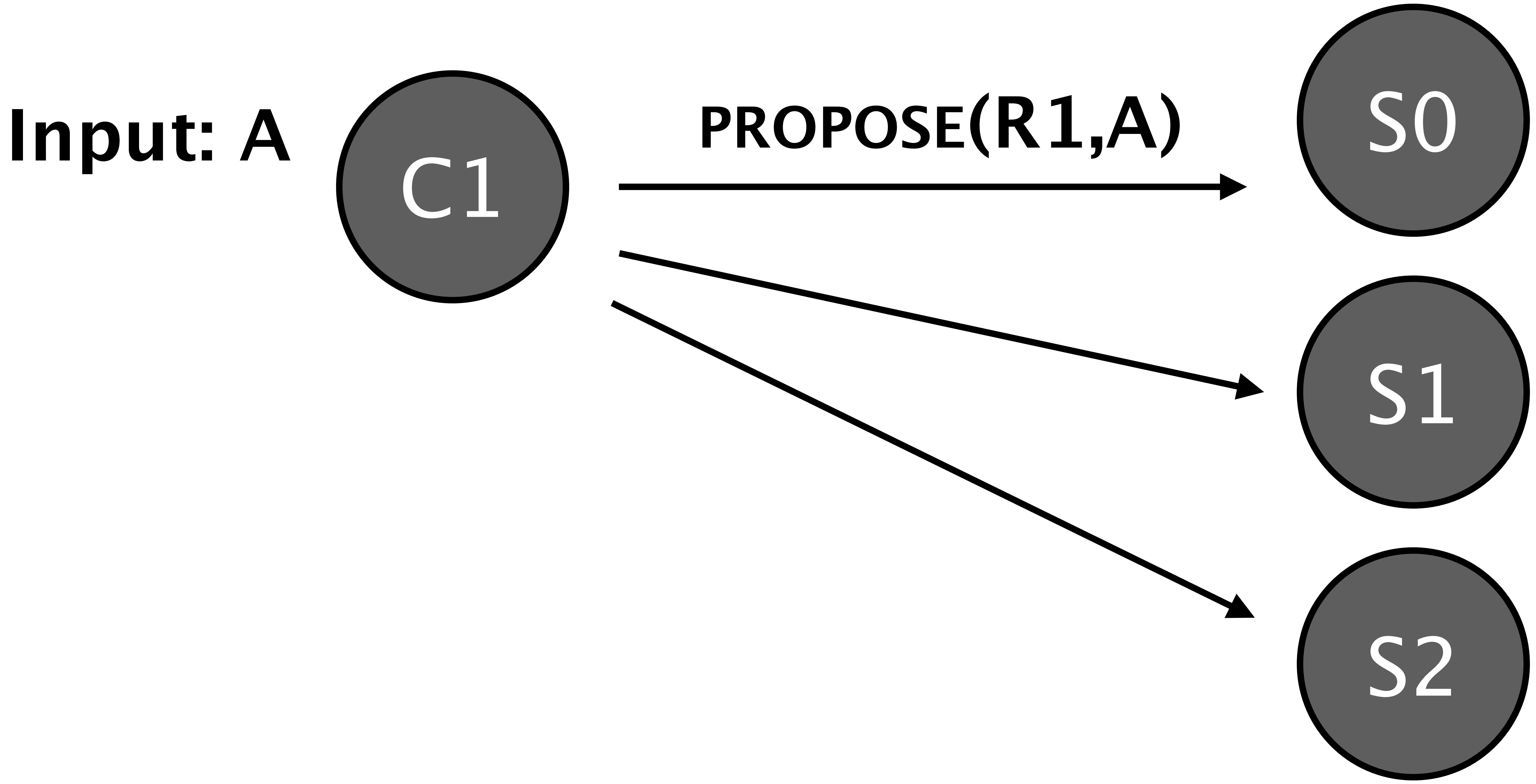

| | S0 | S1 | S2 |
|---|---|---|---|
| **R0** | – | – | – |
| **R1** | | | |
| **R2** | | | |
| **R3** | | | |

# Example – Phase one




| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | | | |
| R2 | | | |
| R3 | | | |

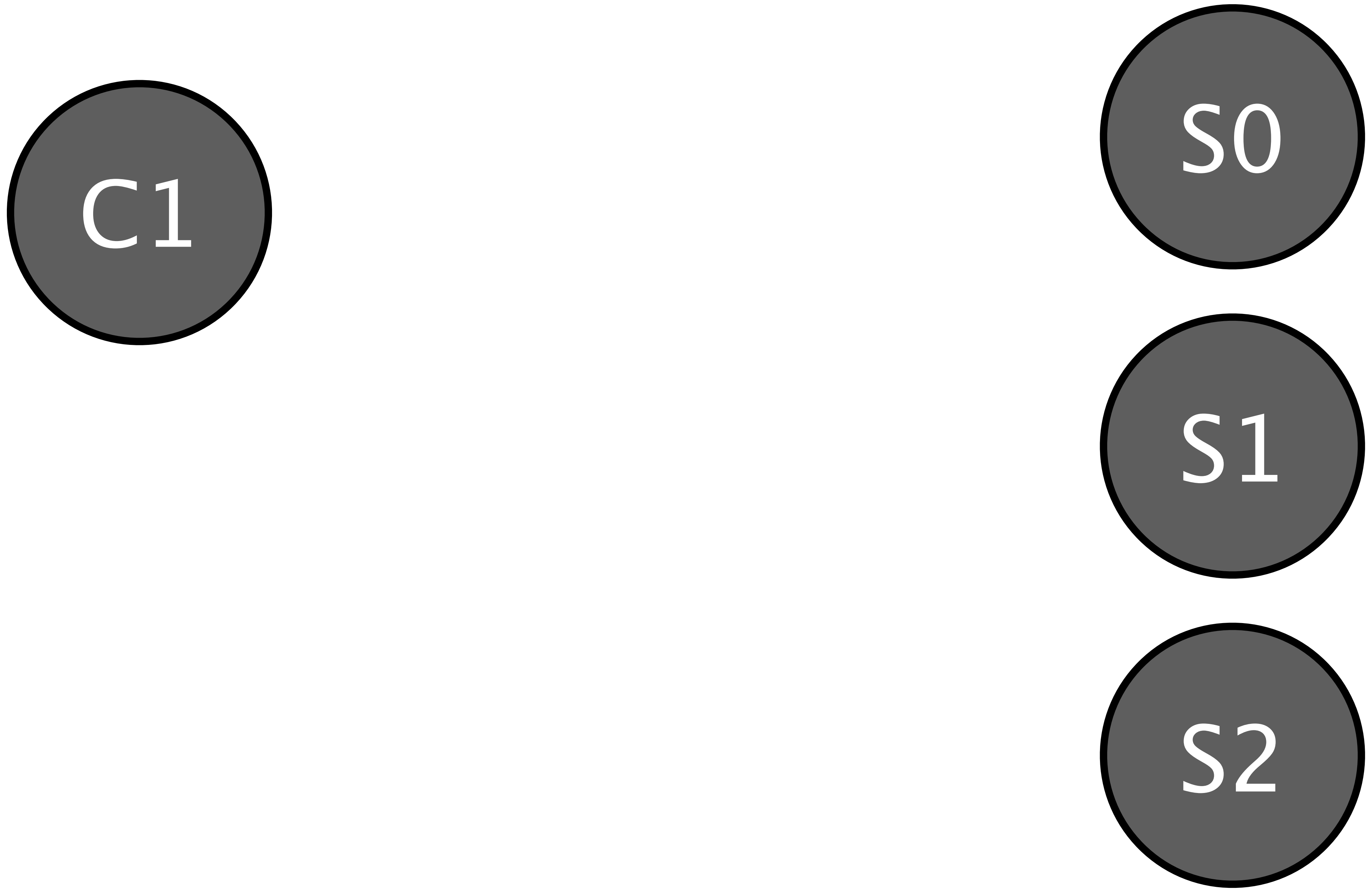
# Example - Phase two




| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | | | |
| R2 | | | |
| R3 | | | |

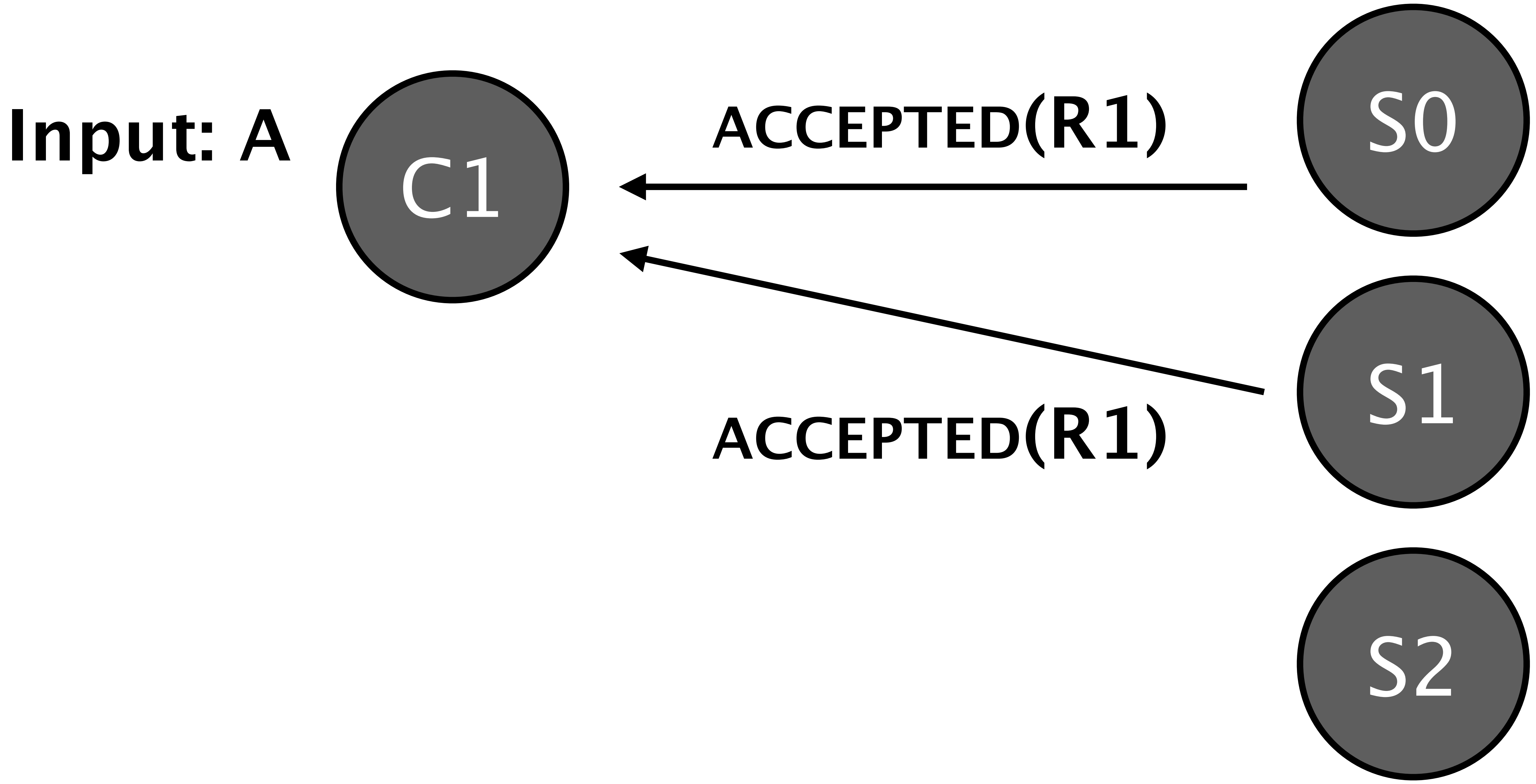
# Example – Phase two

**Input: A**



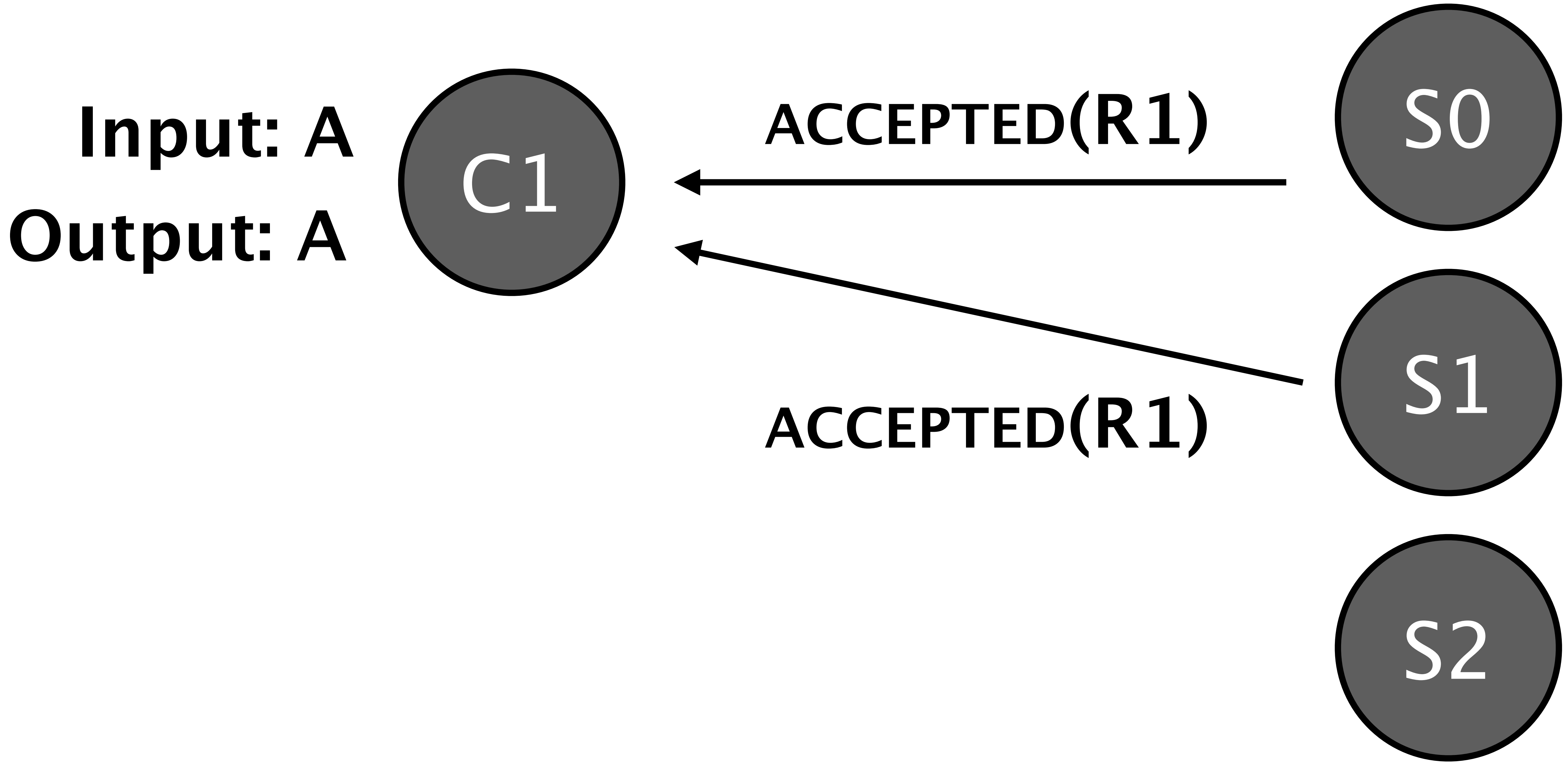

| | S0 | S1 | S2 |
|---|---|---|---|
| **R0** | – | – | – |
| **R1** | A | A | A |
| **R2** | | | |
| **R3** | | | |

# Example – Phase two




| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | A | A |
| R2 | | | |
| R3 | | | |

# Example – Phase two

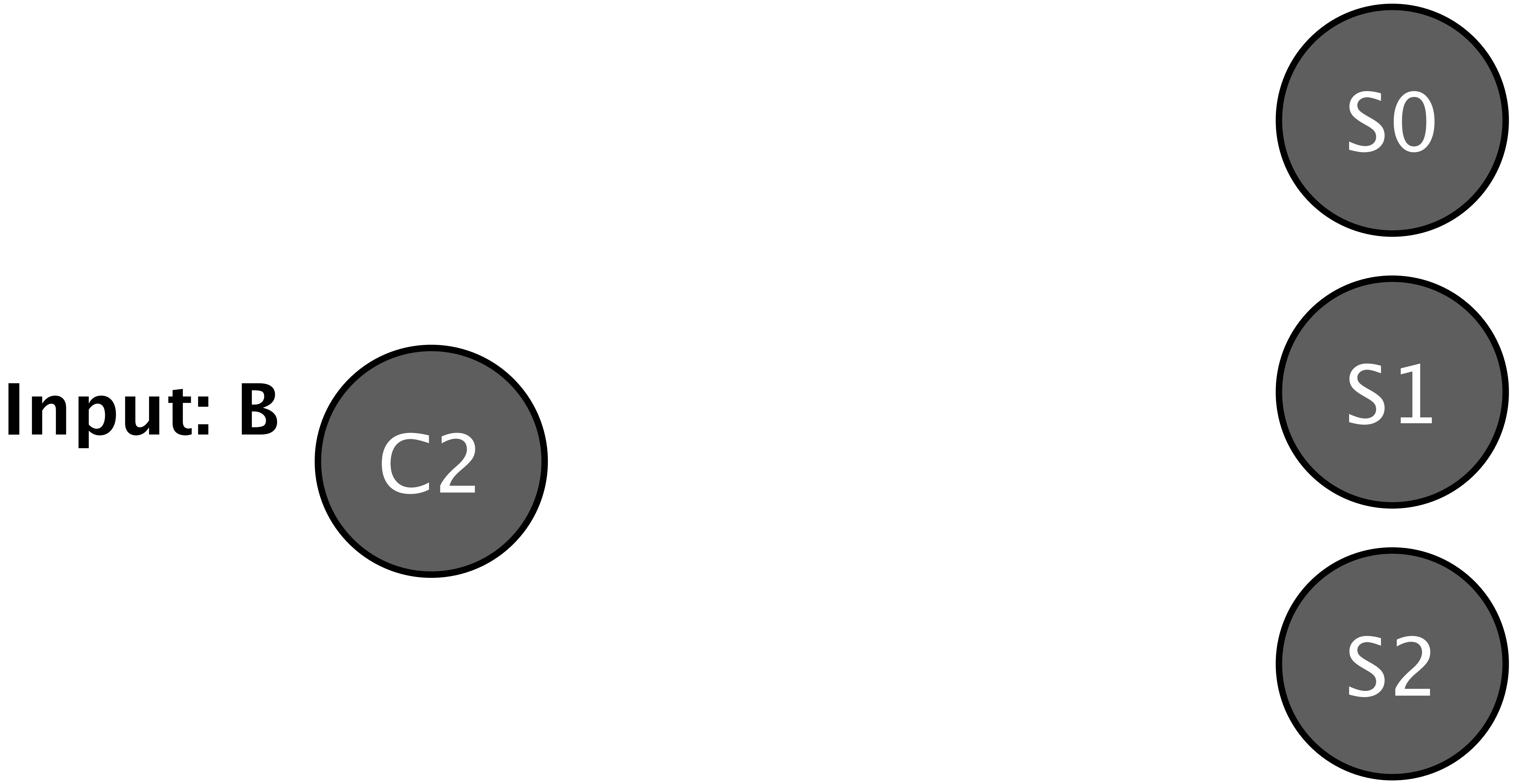



| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | A | A |
| R2 | | | |
| R3 | | | |

# Example – Phase one

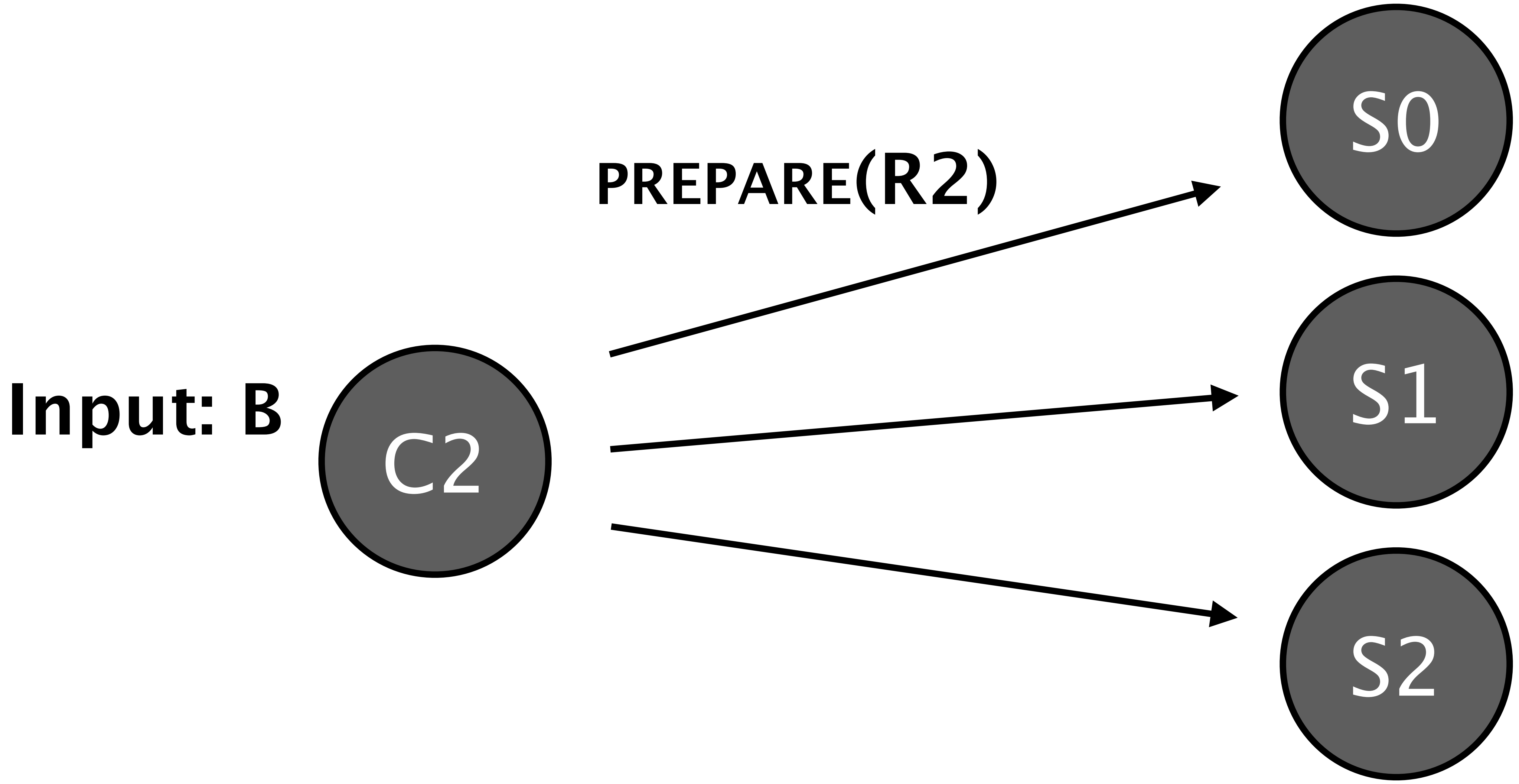

| | S0 | S1 | S2 |
|---|---|---|---|
| **R0** | – | – | – |
| **R1** | A | A | A |
| **R2** | | | |
| **R3** | | | |

# Example – Phase one



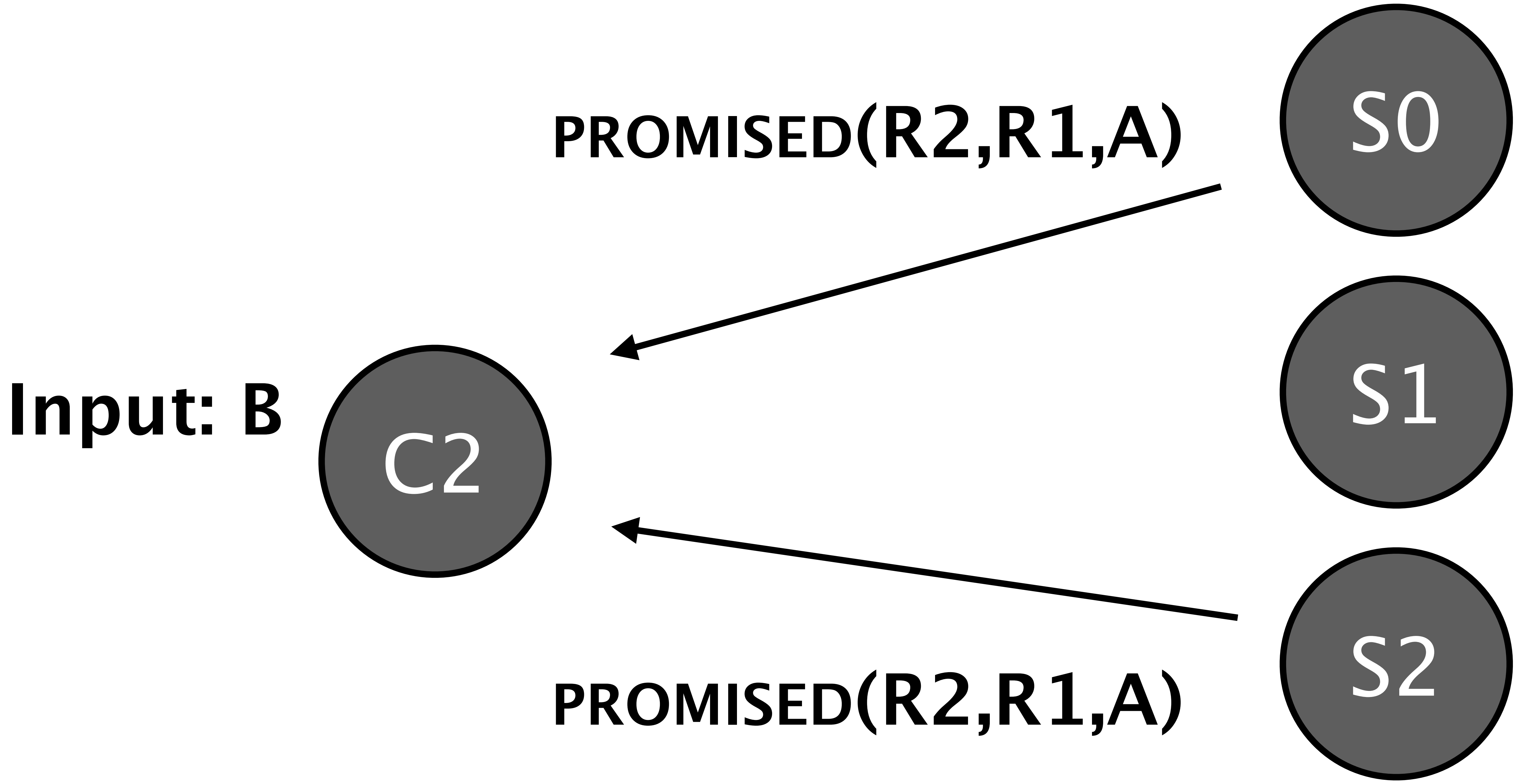

| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | A | A |
| R2 | | | |
| R3 | | | |

# Example – Phase one



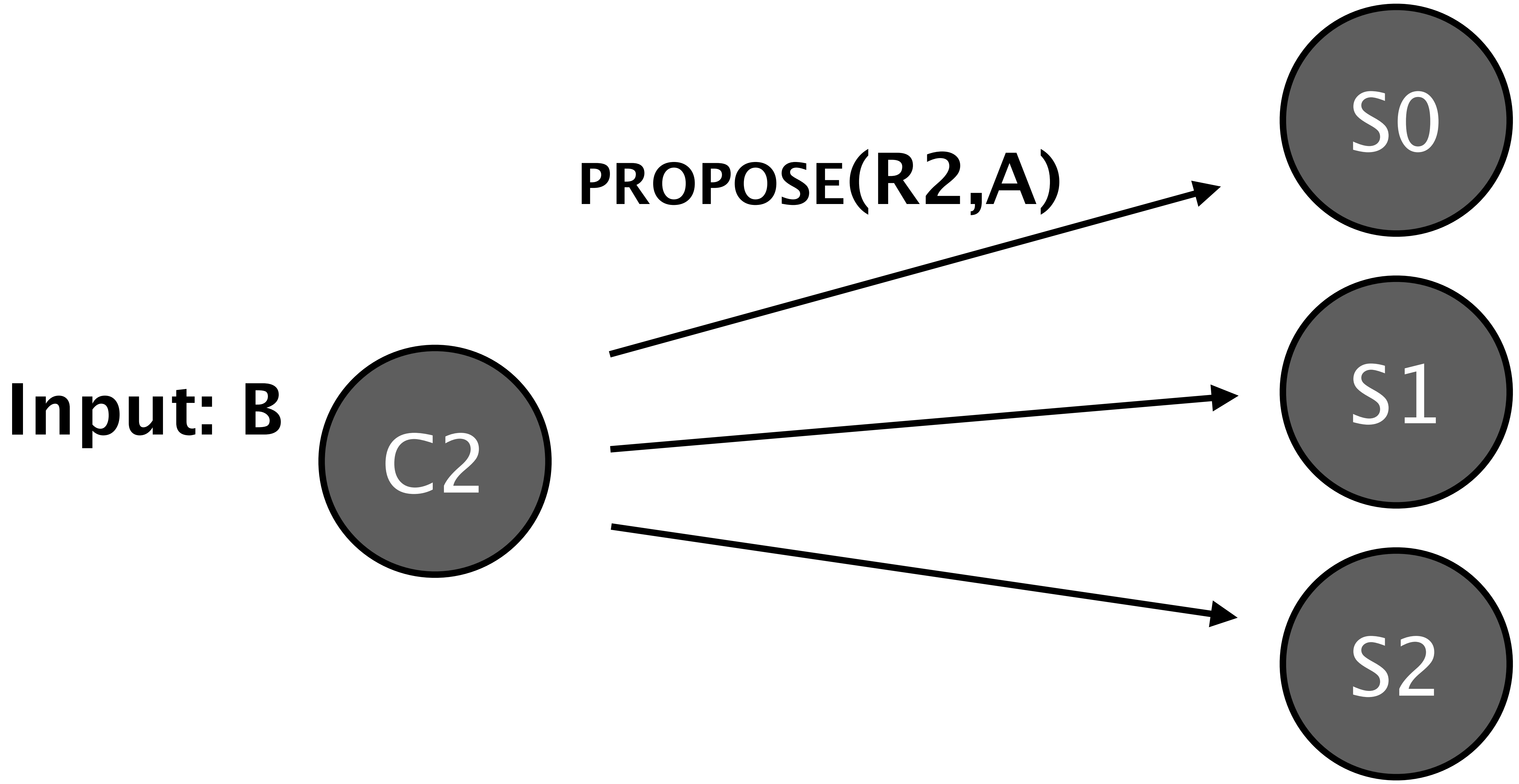

| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | A | A |
| R2 | | | |
| R3 | | | |

# Example – Phase one



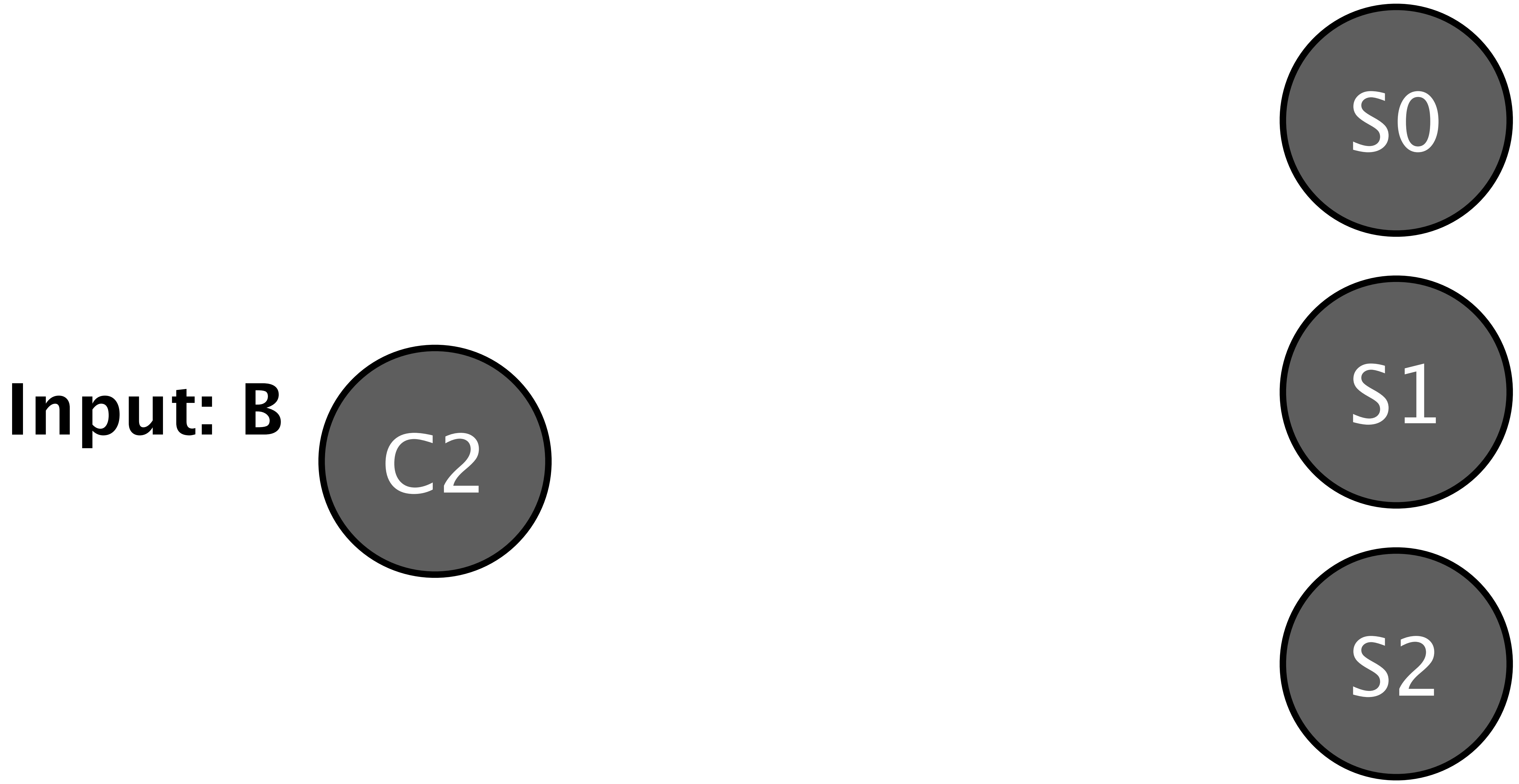

| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | A | A |
| R2 | | | |
| R3 | | | |

# Example – Phase two



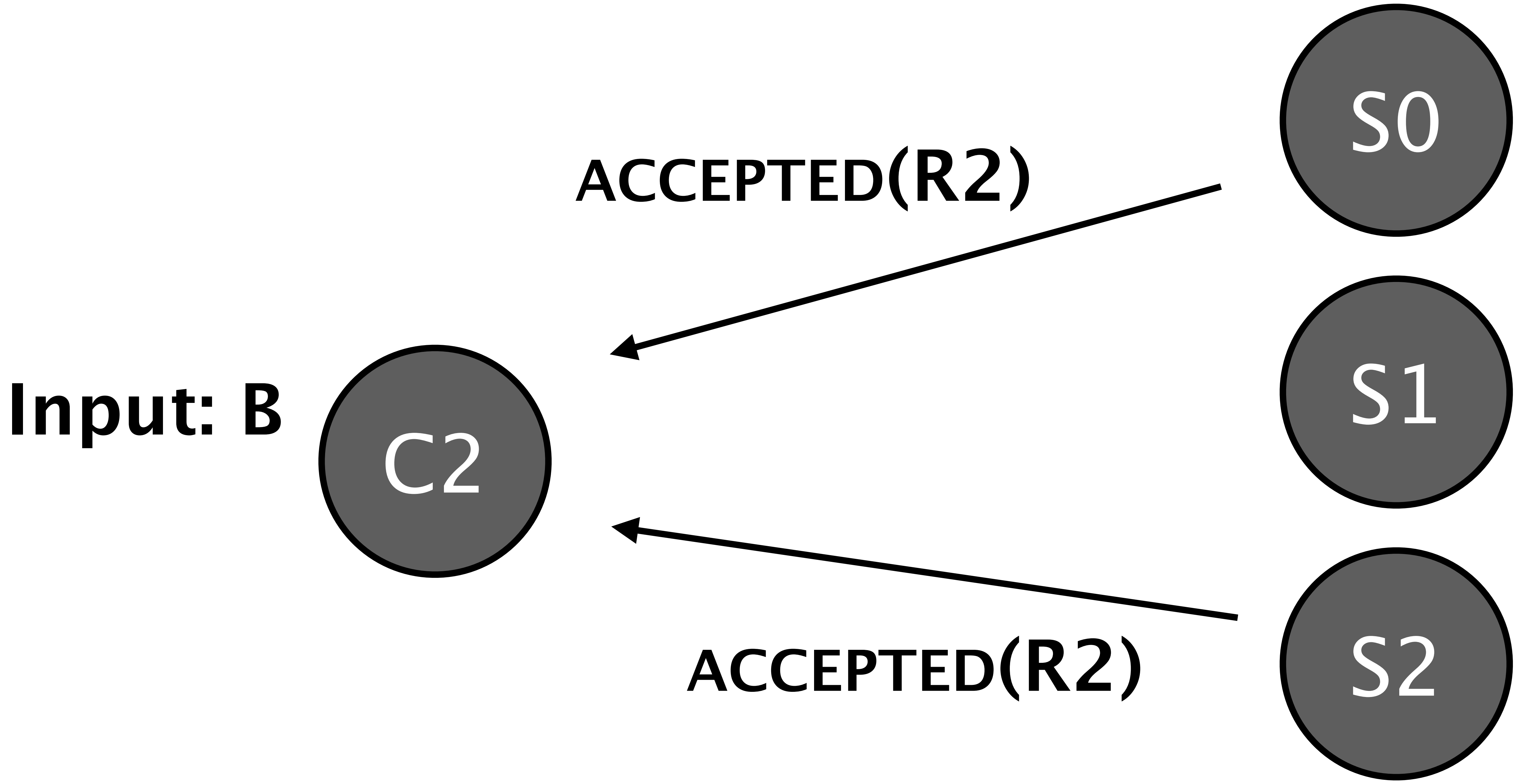

| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | A | A |
| R2 | | | |
| R3 | | | |

# Example – Phase two



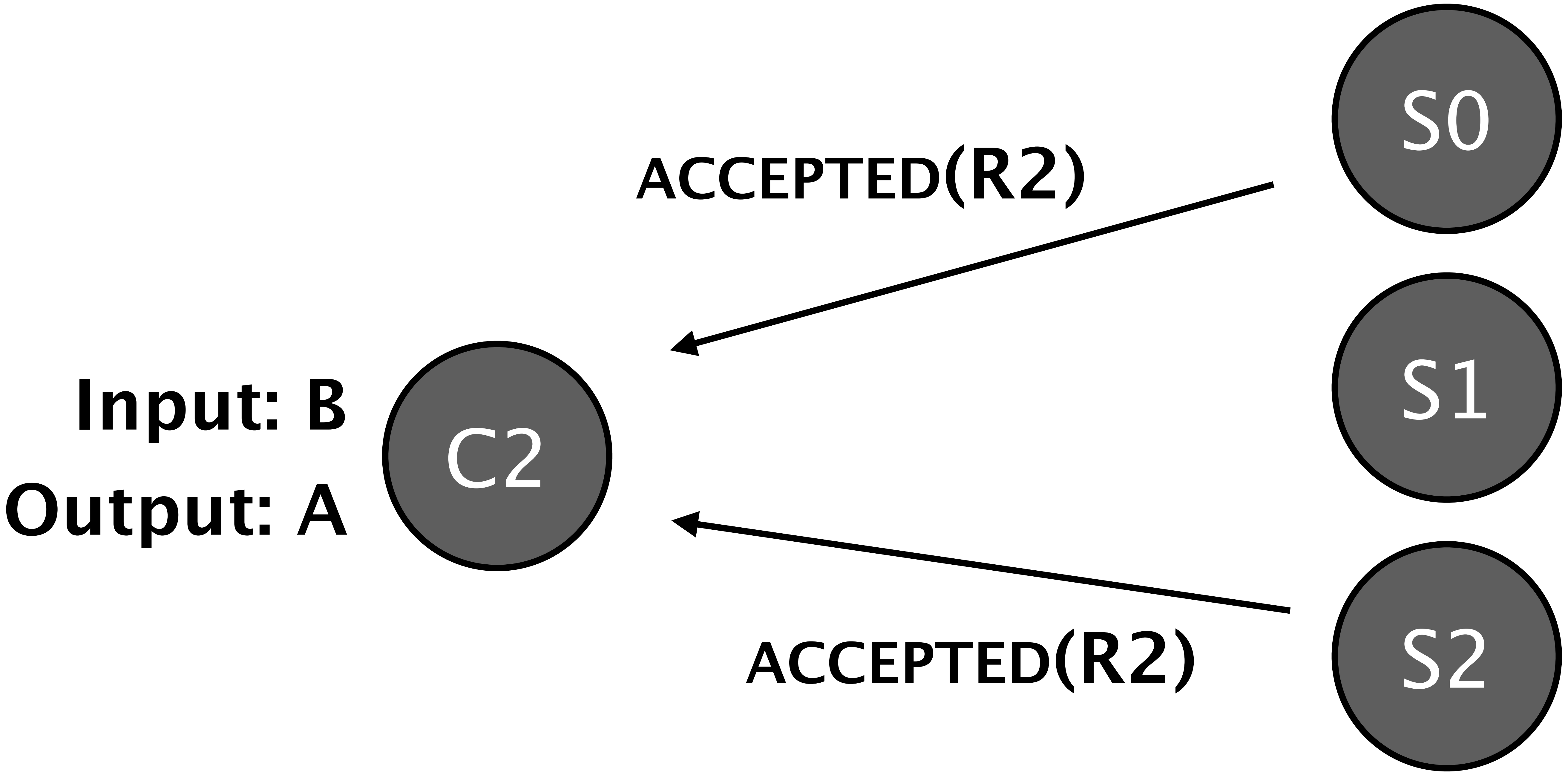

| | S0 | S1 | S2 |
|---|---|---|---|
| **R0** | – | – | – |
| **R1** | A | A | A |
| **R2** | A | A | A |
| **R3** | | | |

# Example – Phase two




| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | A | A |
| R2 | A | A | A |
| R3 | | | |

# Example – Phase two




| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | A | A |
| R2 | A | A | A |
| R3 | | | |

# Slow/faulty clients

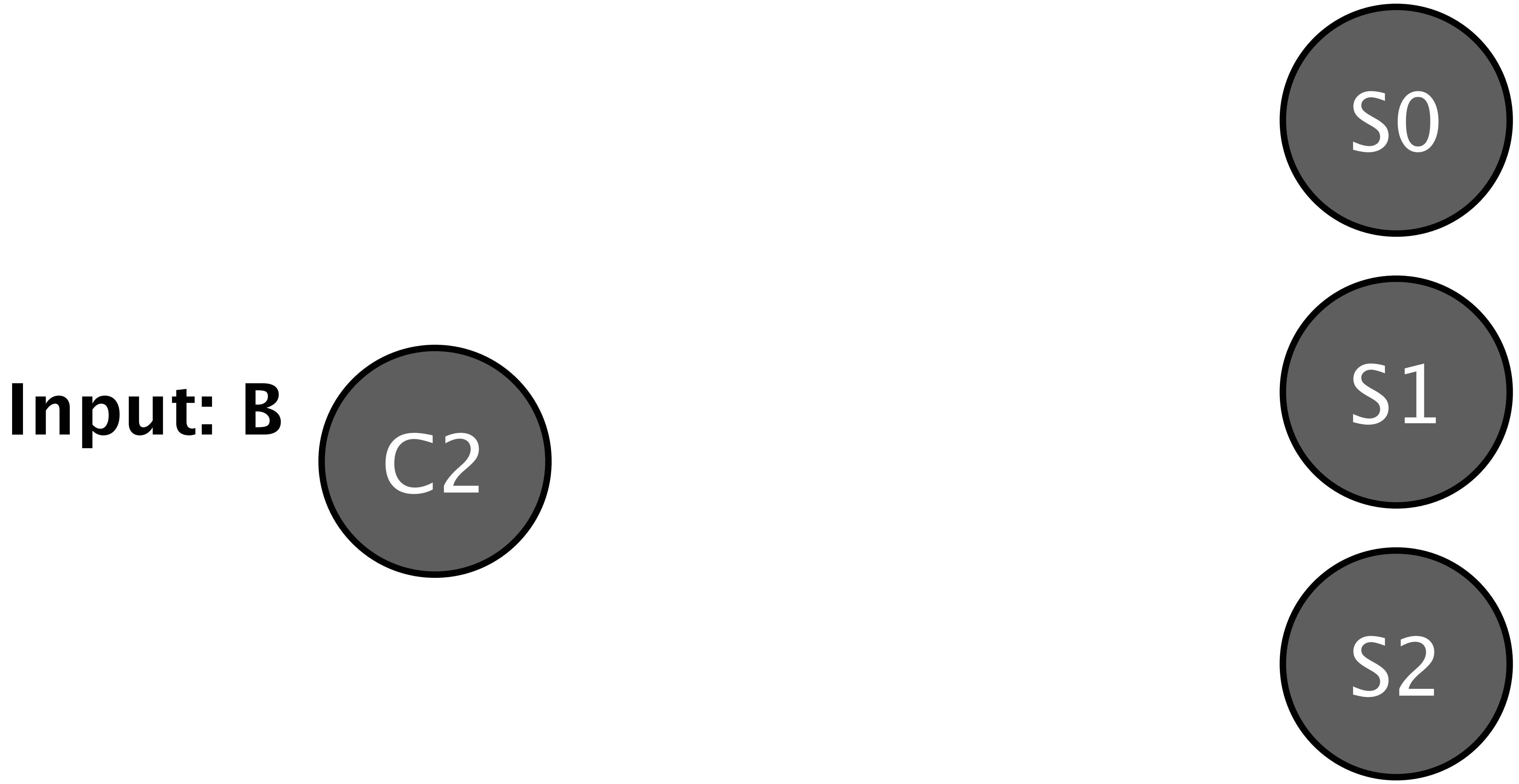
***Bonus Slide**

# Example – Phase one


| | S0 | S1 | S2 |
|---|---|---|---|
| **R0** | – | – | – |
| **R1** | A | | |
| **R2** | | | |
| **R3** | | | |

***Bonus Slide**

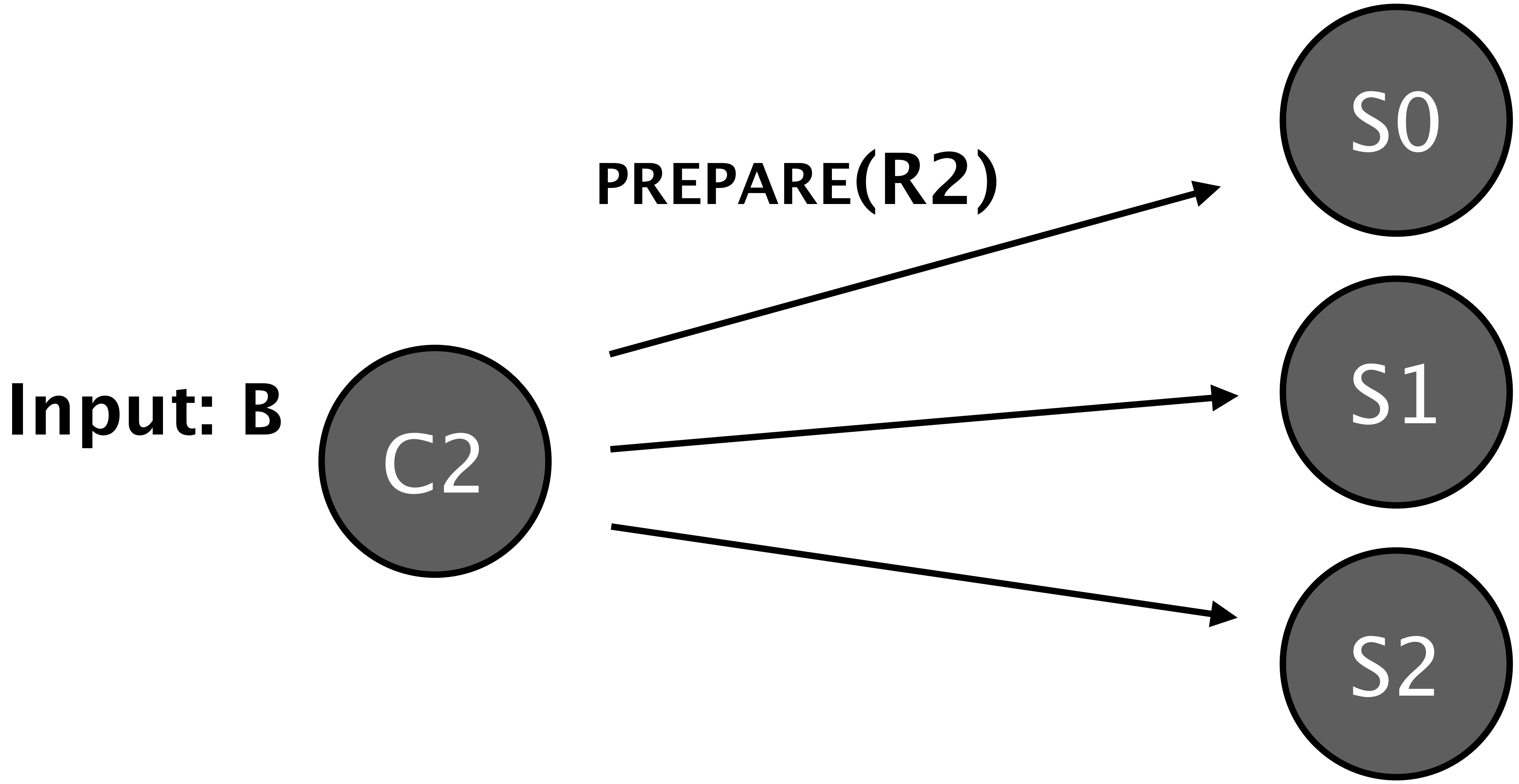
# Example - Phase one




| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | - | - | - |
| R1 | A | | |
| R2 | | | |
| R3 | | | |

*Bonus Slide

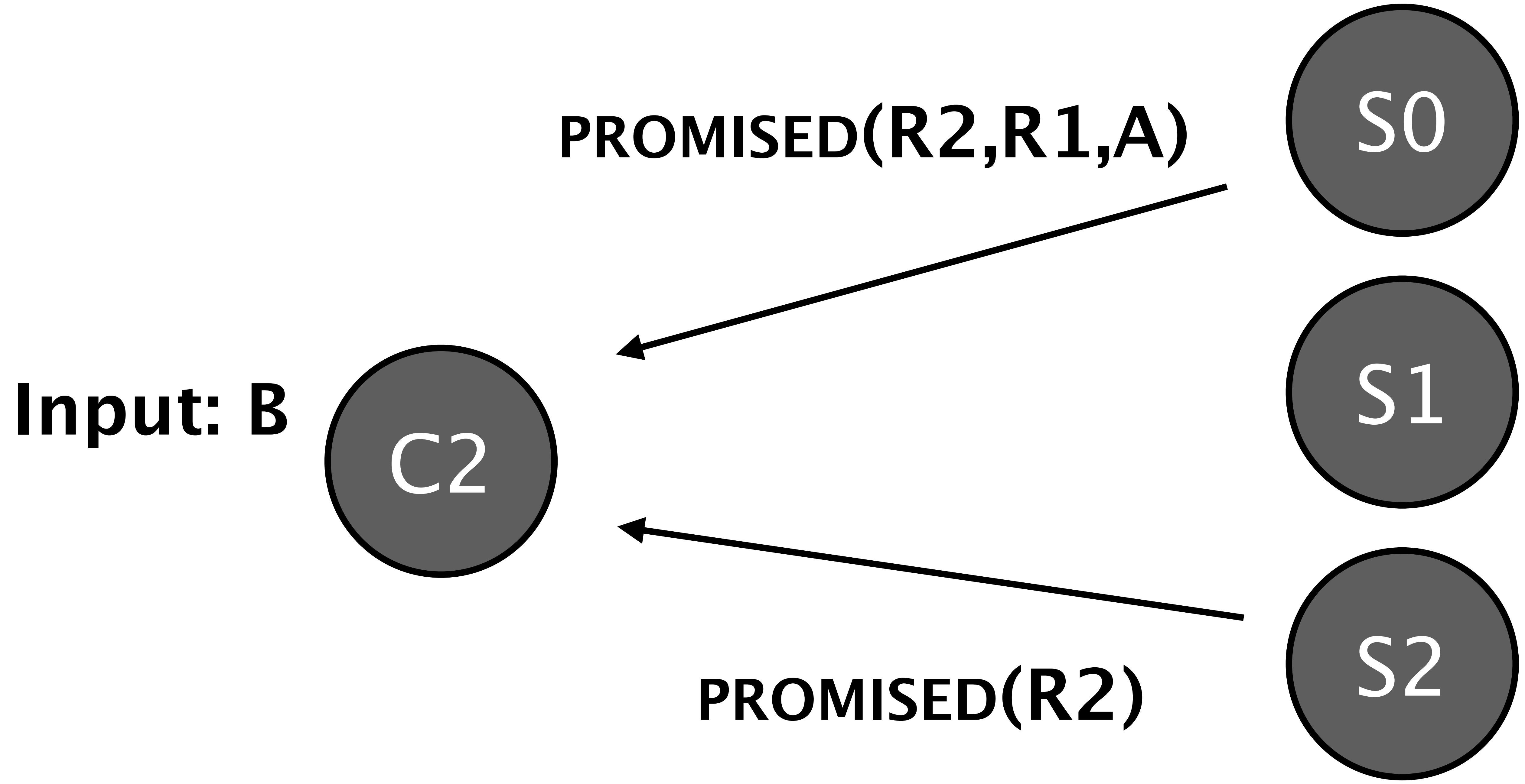
# Example – Phase one




| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | | |
| R2 | | | |
| R3 | | | |

***Bonus Slide**

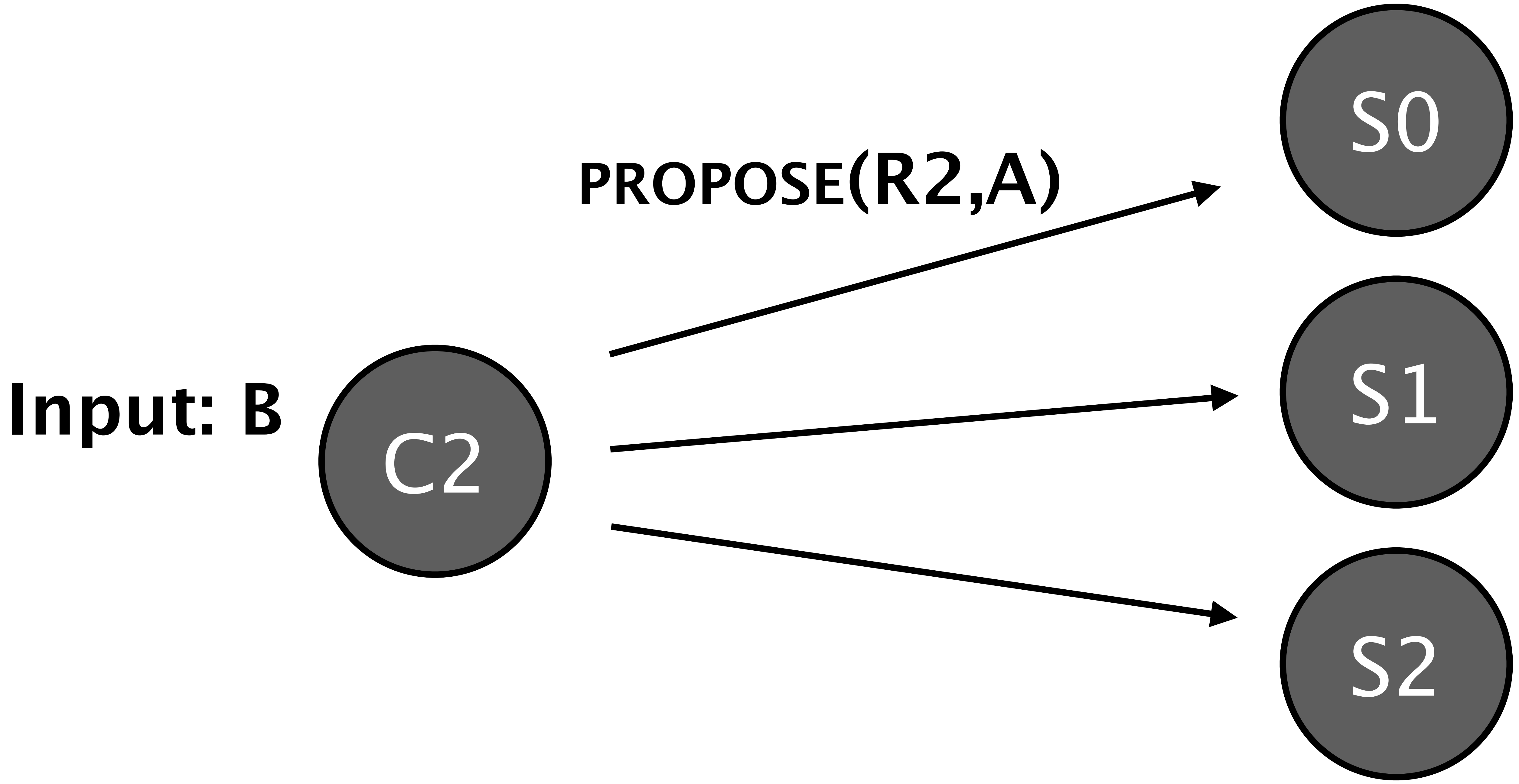
# Example – Phase one




| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | – | – |
| R2 | | | |
| R3 | | | |

*Bonus Slide

# Example – Phase two

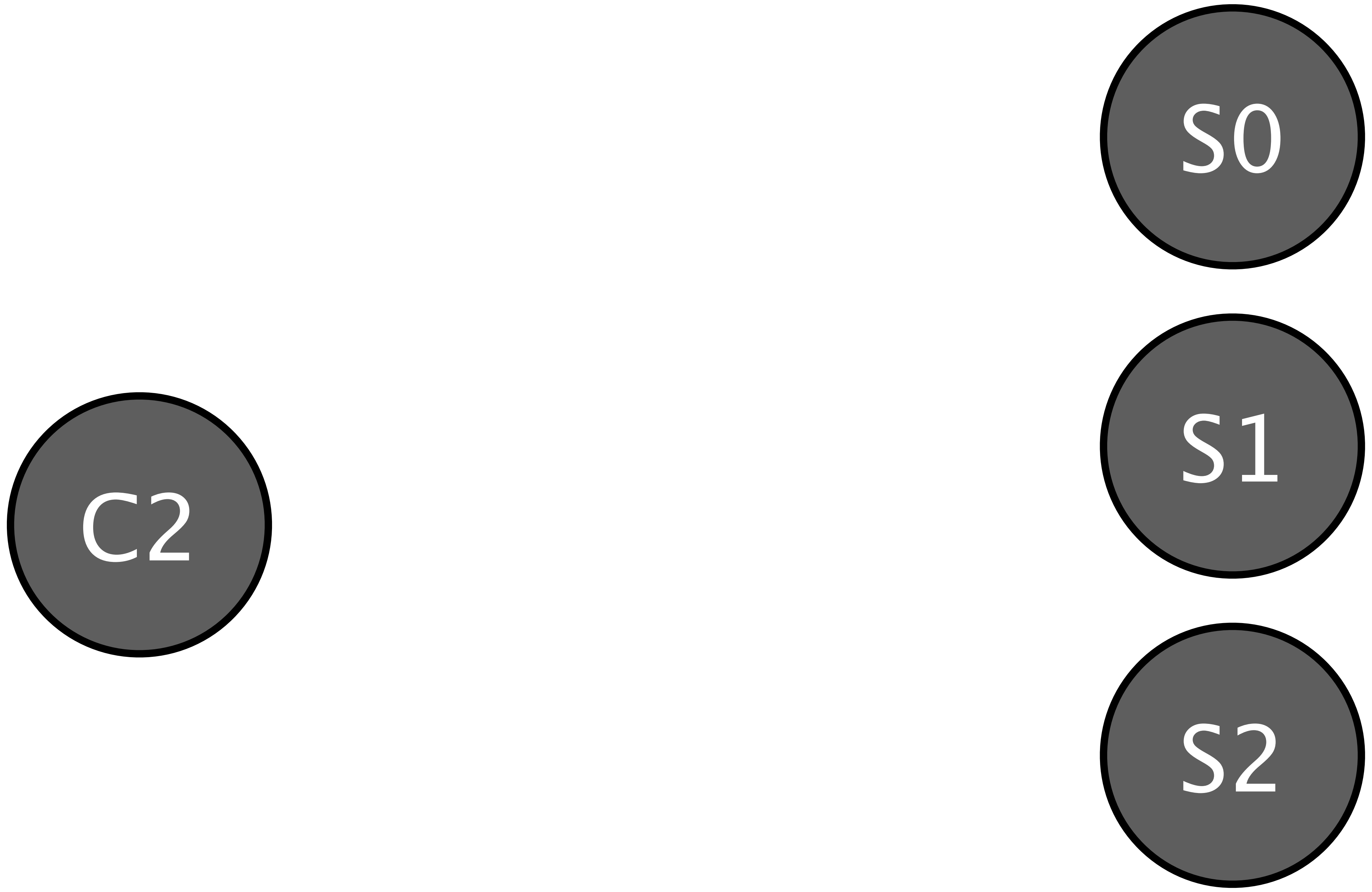



| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | – | – |
| R2 | | | |
| R3 | | | |

***Bonus Slide**

# Example – Phase two

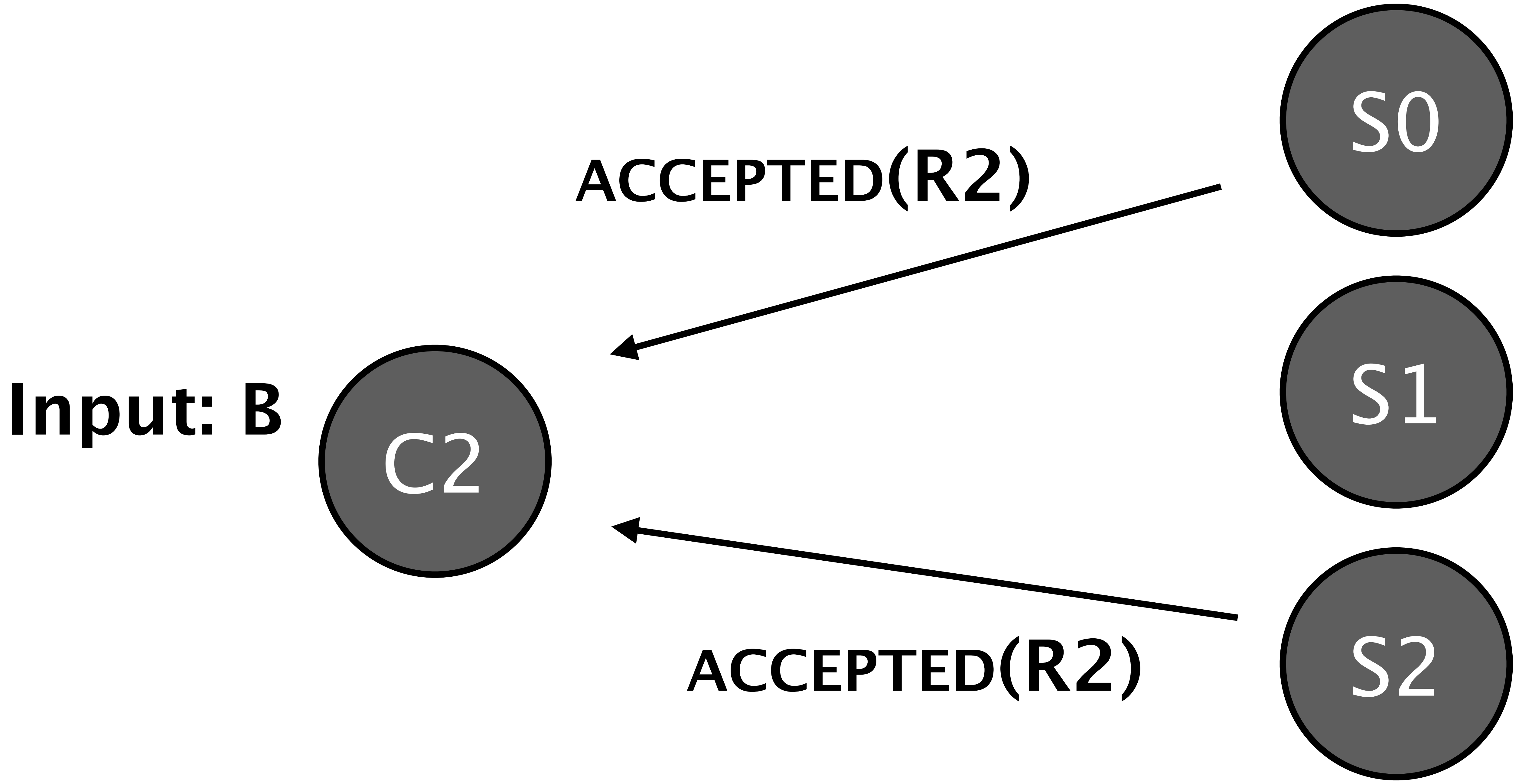
Input: B




| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | – | – |
| R2 | A | A | A |
| R3 | | | |

*Bonus Slide

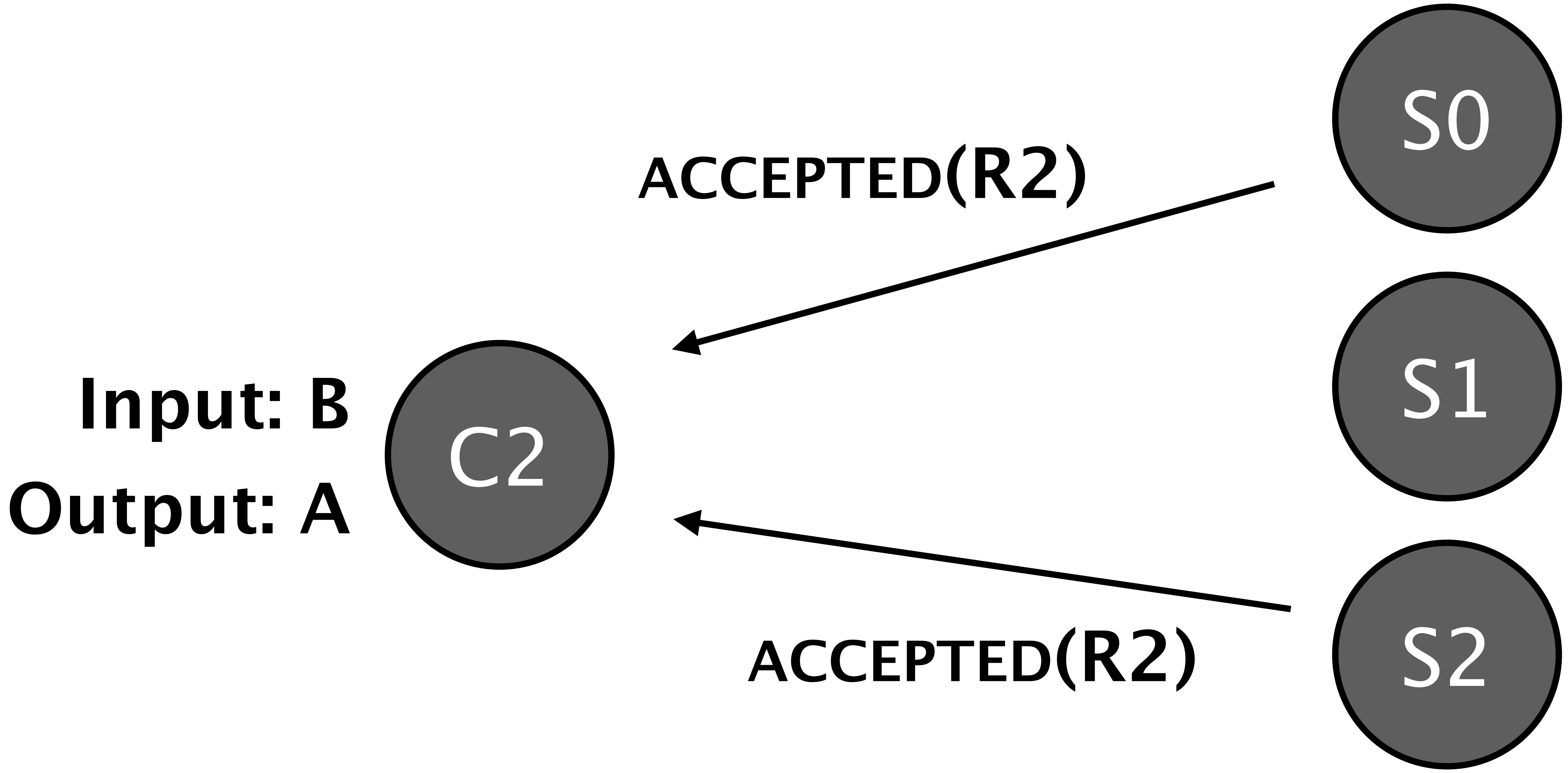
# Example – Phase two




| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | – | – |
| R2 | A | A | A |
| R3 | | | |

*Bonus Slide

# Example – Phase two




| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | – | – |
| R2 | A | A | A |
| R3 | | | |

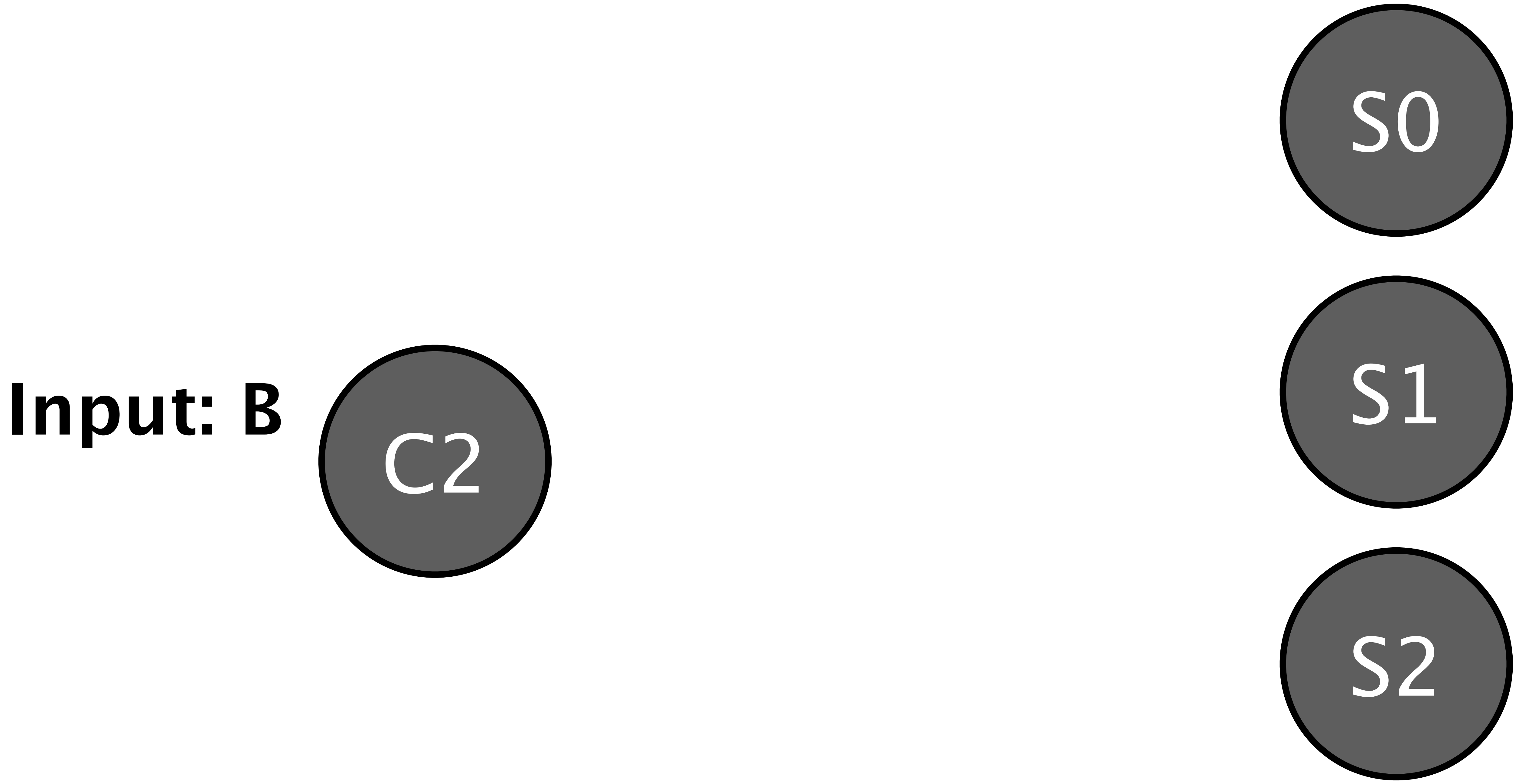
*Bonus Slide

# Example – Phase one


| | S0 | S1 | S2 |
|---|---|---|---|
| **R0** | – | – | – |
| **R1** | A | | |
| **R2** | | | |
| **R3** | | | |

***Bonus Slide**

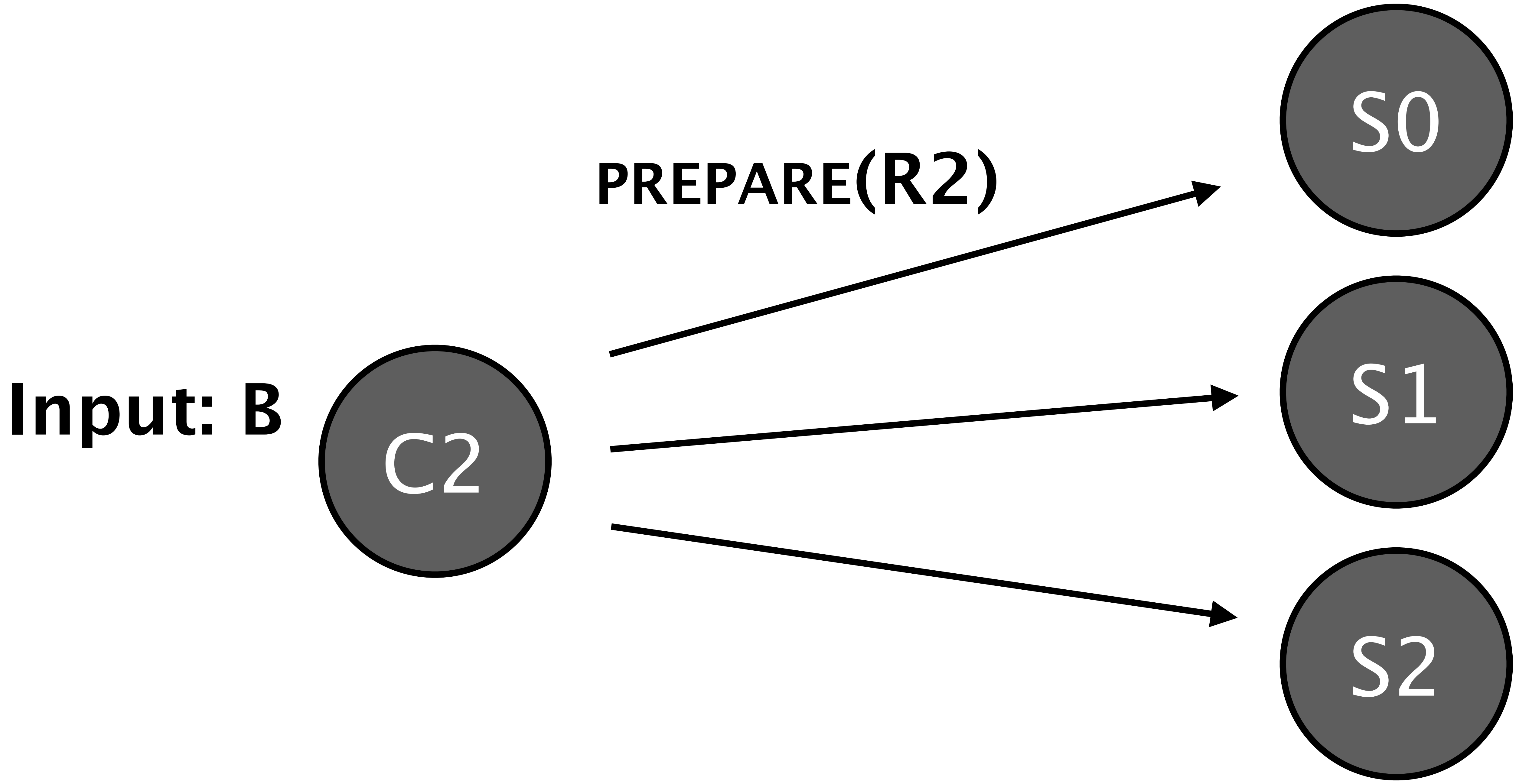
# Example – Phase one




| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | | |
| R2 | | | |
| R3 | | | |

*Bonus Slide

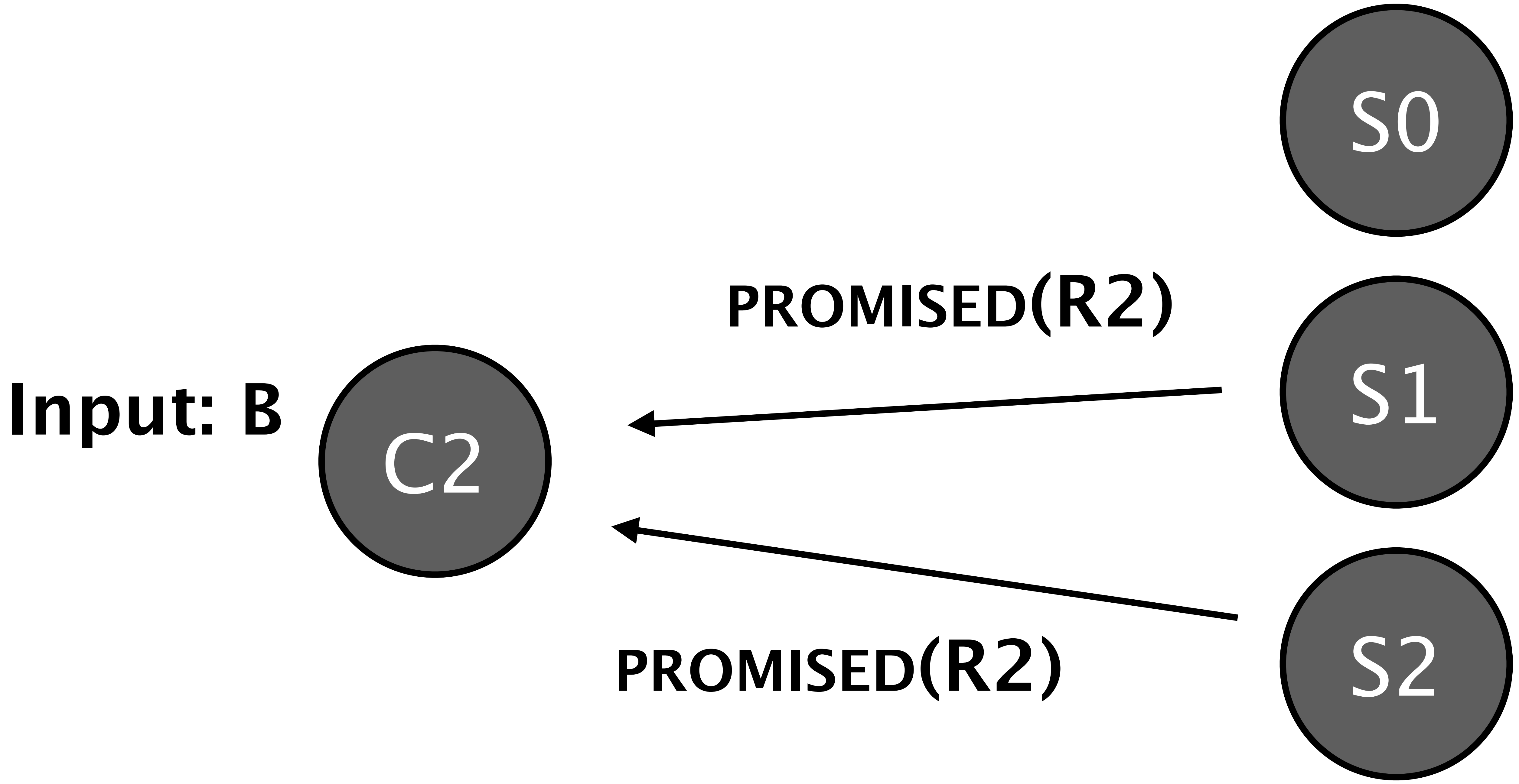
# Example – Phase one




| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | | |
| R2 | | | |
| R3 | | | |

*Bonus Slide

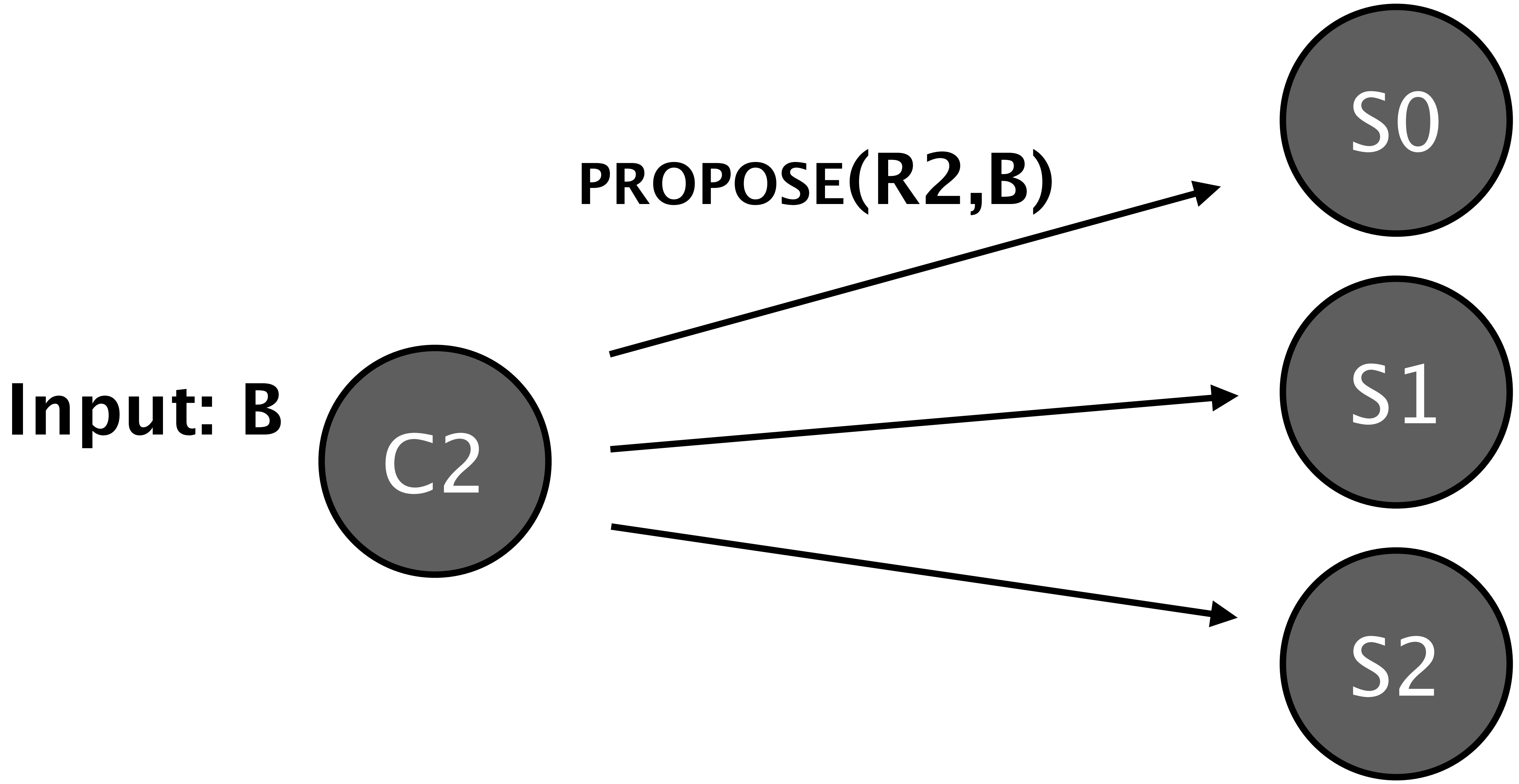
# Example – Phase one




| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | – | – |
| R2 | | | |
| R3 | | | |

*Bonus Slide

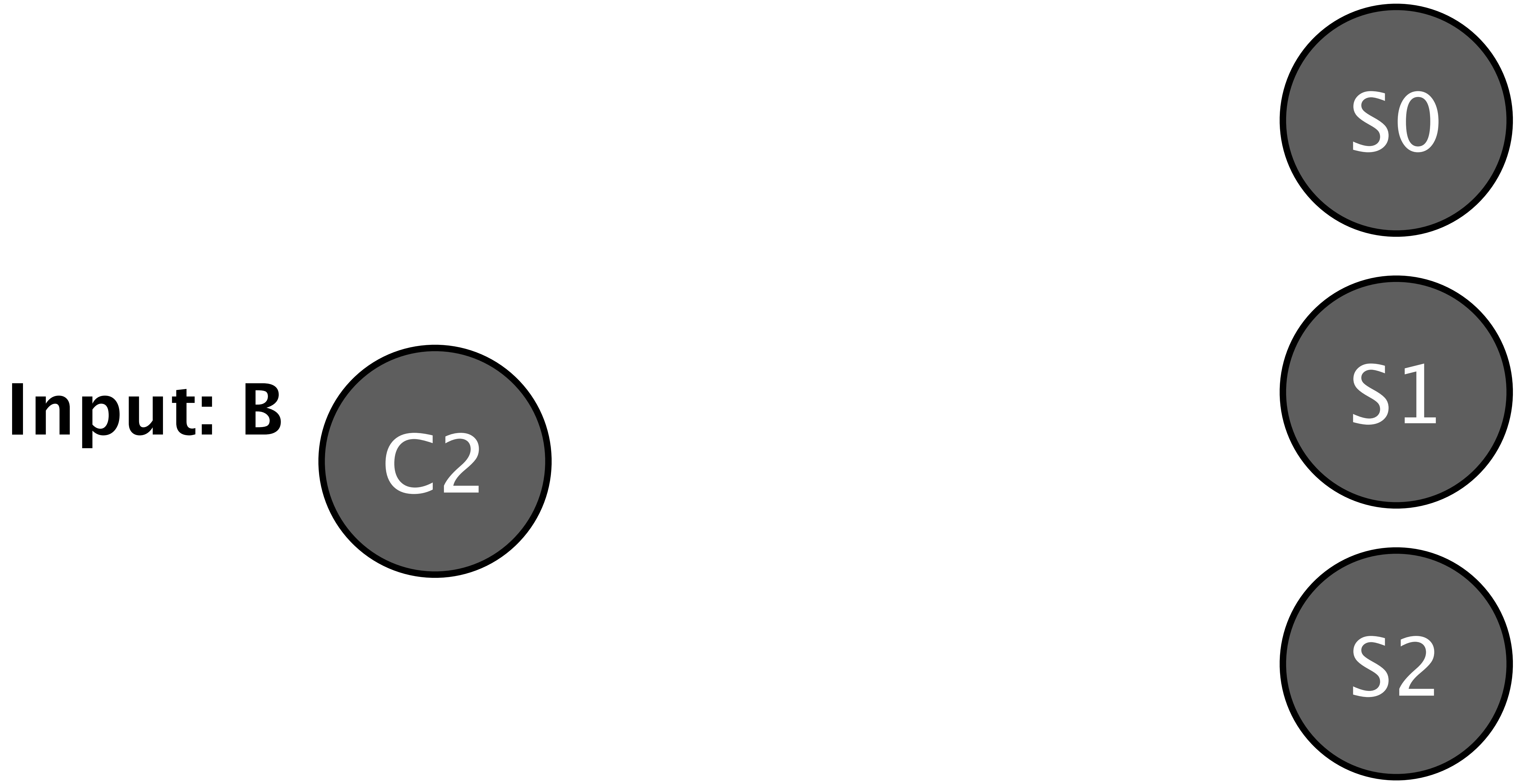
# Example – Phase two




| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | – | – |
| R2 | | | |
| R3 | | | |

*Bonus Slide

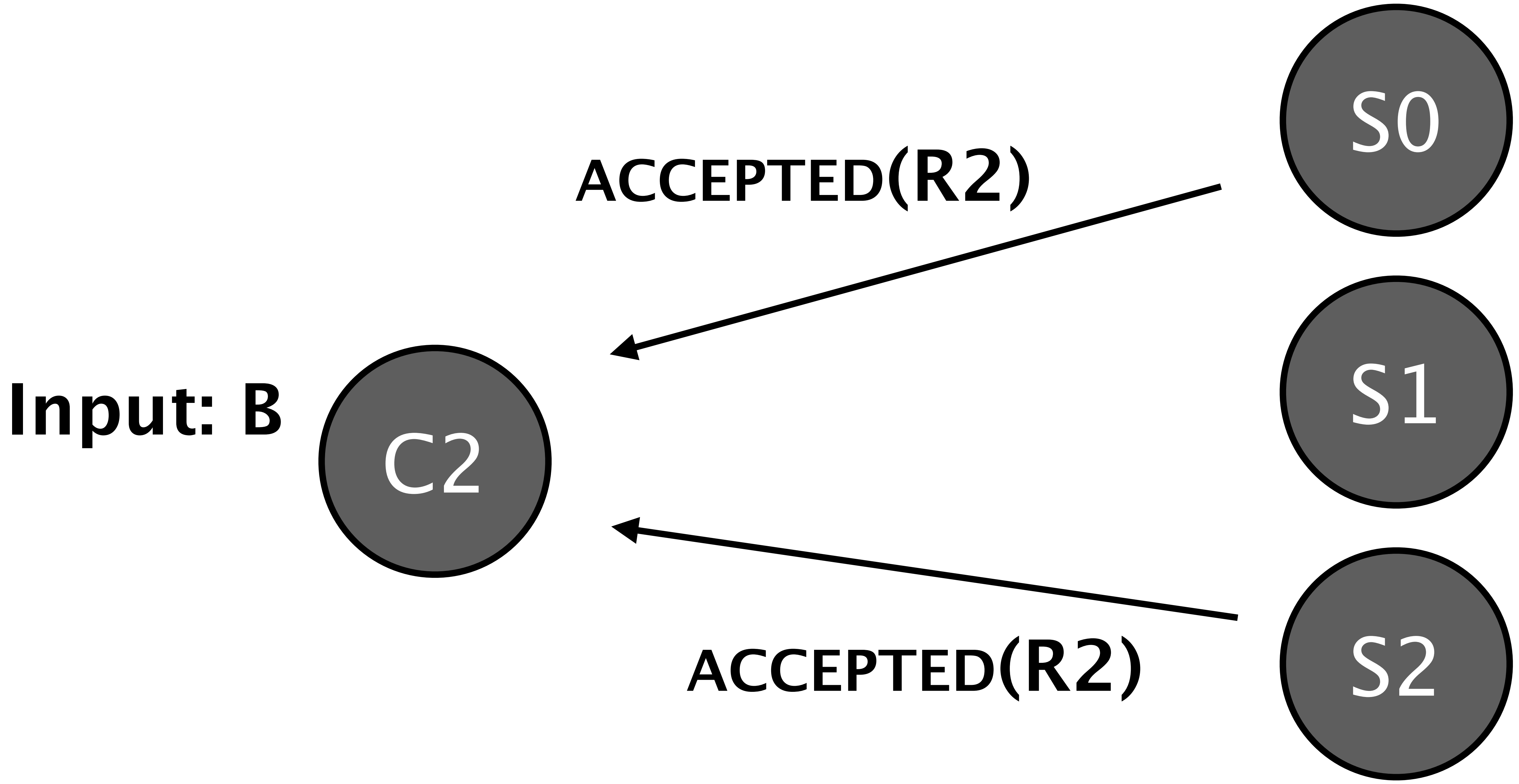
# Example – Phase two




| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | – | – |
| R2 | B | B | B |
| R3 | | | |

*Bonus Slide

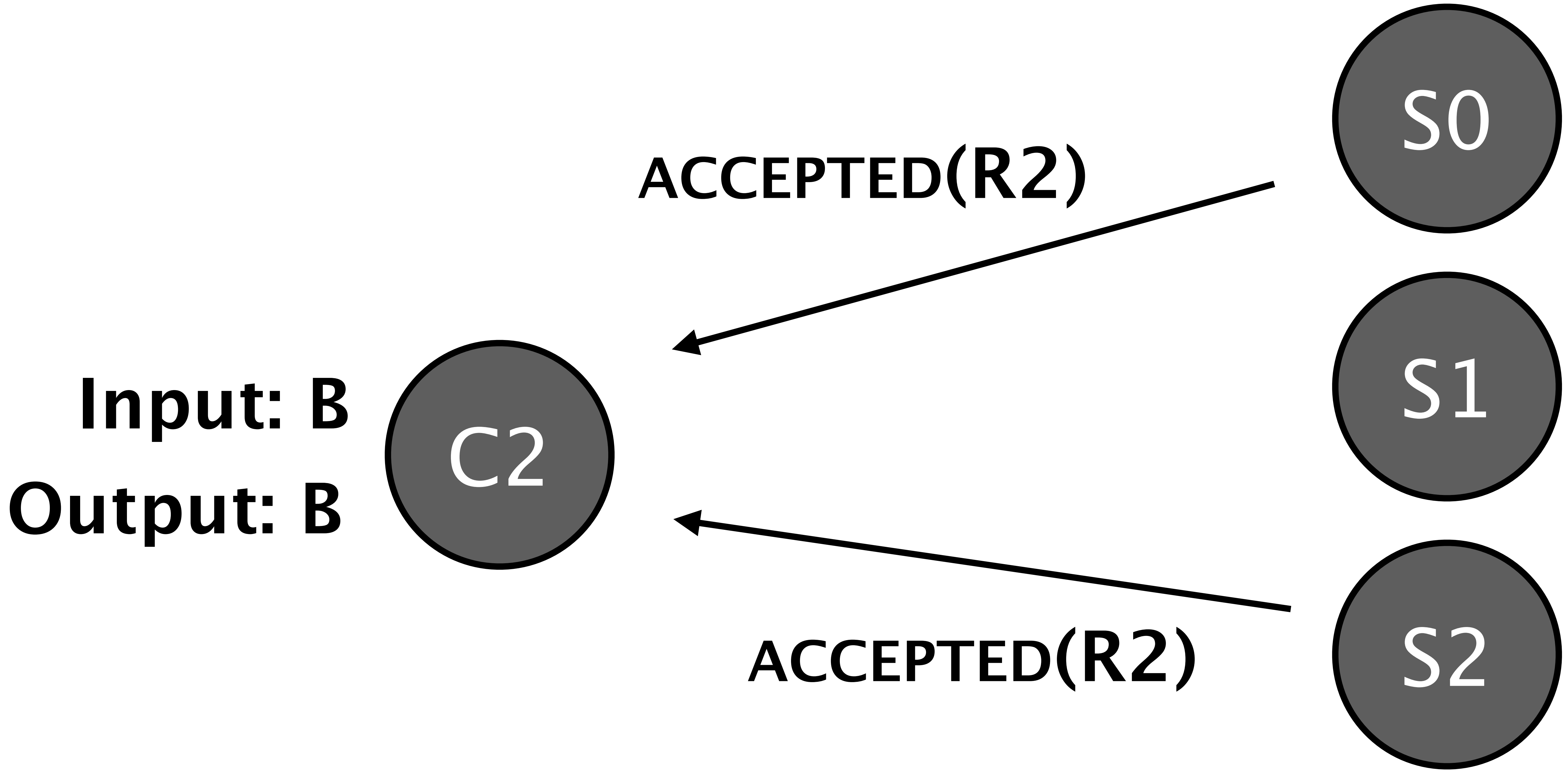
# Example – Phase two




| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | – | – |
| R2 | B | B | B |
| R3 | | | |

*Bonus Slide

# Example – Phase two




| | S0 | S1 | S2 |
|---|---|---|---|
| R0 | – | – | – |
| R1 | A | – | – |
| R2 | B | B | B |
| R3 | | | |

*Bonus Slide

# Quorum intersection

# Quorum intersection

**Original requirement**

Paxos requires that a quorum of servers participate in each of its two phases and that any two quorums must intersect.

# Quorum intersection

**Original requirement**

Paxos requires that a quorum of servers participate in each of its two phases and that any two quorums must intersect.

**Revised requirement**

A client using register i must get at least one server from each quorum of registers 0 to i–1 to participate in phase one.

# Part 3
# All aboard consensus

# Current Reality

| | Classic Paxos | Multi Paxos |
|---|---|---|
| **Minimum round trips?** | 2 | 1 |
| **Which client can decide the value?** | Any | Leader only |

# Current Reality

| | Classic Paxos | Multi Paxos |
|---|---|---|
| **Minimum round trips?** | 2 | 1 |
| **Which client can decide the value?** | Any | Leader only |

Can we design an algorithm in which ***any client*** can achieve consensus in just ***1 round trip***?

# Designing for today

# Designing for today

1. Failures are rare.

# Designing for today

1. Failures are rare.

2. Each host is a client and server.

# All aboard – Quorum table

Registers partitioned at R2

All servers

| | Quorums |
|---|---|
| **R0, R1, R2** | {S0,S1,S2} |
| **R3+** | {S0,S1} {S1,S2} {S0,S2} |

Majority quorums

# All aboard – Algorithm

# All aboard – Algorithm

**Fast path [R0 – R2]**

Client executes phase one locally, followed by phase two with all servers.

# All aboard – Algorithm

**Fast path [R0 – R2]**

Client executes phase one locally, followed by phase two with all servers.

**Slow path [R3+]**

Client executes classic Paxos with majority quorums for both phases.

# All aboard – Summary

# All aboard – Summary

**Pros**

- Any clients can terminate in just one round trip (provided all servers are up).

# All aboard – Summary

**Pros**

- Any clients can terminate in just one round trip (provided all servers are up).

**Cons**

- The fast path has increased the quorum size from majority to all.
- More round trips are needed if a server is slow/unavailable.

# Lessons learned

# Lessons learned

Immutability and generality can change our perspective on distributed consensus.

# Lessons learned

Immutability and generality can change our perspective on distributed consensus.

Paxos can relax its quorum intersection requirements. Utilising different quorums tables can produce different tradeoffs.

# Lessons learned

Immutability and generality can change our perspective on distributed consensus.

Paxos can relax its quorum intersection requirements. Utilising different quorums tables can produce different tradeoffs.

Paxos with majorities is a single point on a broad and diverse spectrum of consensus algorithms.

# Q & A

Heidi Howard
heidi.howard@cl.cam.ac.uk
@heidiann360
heidihoward.co.uk