Liberating Distributed Consensus

Heidi Howard
Systems Research Group @ Cambridge University

heidi.howard@cl.cam.ac.uk
@heidiann360
www.cl.cam.ac.uk/~hh360

Distributed Dream

- **Performant** - scalable, low latency, high throughput, geo-replicated, energy/cost efficient, versatile
- **Reliable** - fault-tolerant, dependable, highly available, AP of CAP, self-healing
- **Correct** - consistent, behaves as expected

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 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.

*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

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

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.

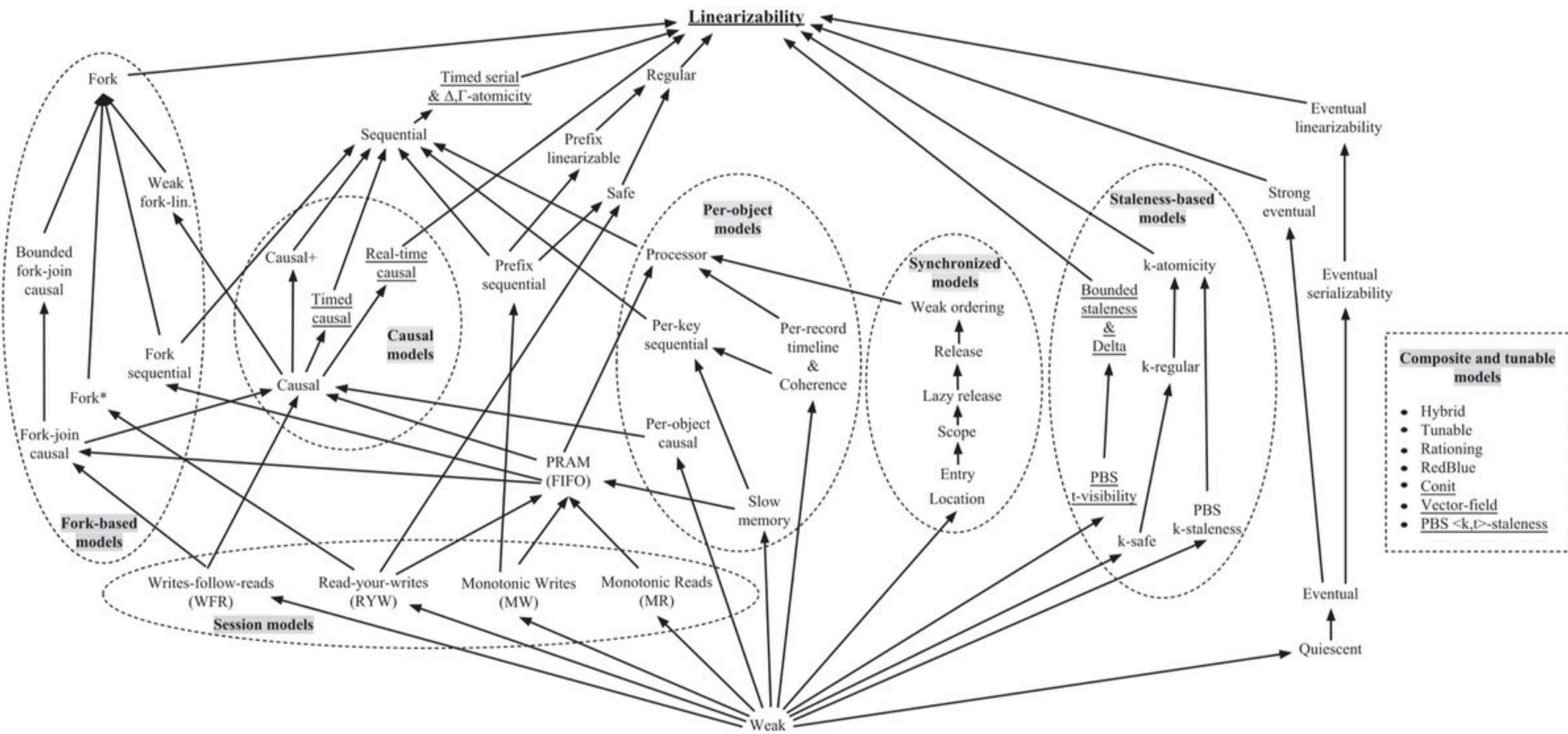
This work was supported in part by the Office of Naval Research under Contract N00014-82-K-0154, by the Office of Army Research under Contract DAAG29-79-C-0155, and by the National Science Foundation under Grants MCS-7924370 and MCS-8116678.

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

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1985 ACM 0004-5411/85/0400-0374 $00.75

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



Consistency in Non-Transactional Distributed Storage Systems

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 read/write the value

We assume an unreliable, asynchronous, non-Byzantine system.

Current Reality

Classic Paxos is a two phase, majority based, algorithm for reaching consensus.

Multi Paxos (including Zab & Raft) is a widely adopted optimisation of Classic Paxos, which works by electing one client as the “leader”.

Current Reality

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

“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

Theory community perspective

“There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system. ”

“Despite the existing literature on [Paxos], building a production system turned out to be a non-trivial task”

Engineering community perspective

Chandra et al, Paxos Made Live

“Paxos is exceptionally difficult to understand. The full explanation is notoriously opaque; 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

Systems community perspective

Limitations

Subtlety

Poorly understood thus difficult to implement correctly and to optimise

Poor Performance

- Majority agreement is slow thus scale limited
- In Classic Paxos, at least two round trips is needed to reached consensus, more in the case of conflict
- In Multi Paxos, the leader is the bottleneck. The capacity of the leader limits throughput and all decisions must go via the leader, adding latency.

Today's Talk

Instead of mitigating these issues, we rethink the underlying principles.

Part 1

We outline an abstract solution to consensus using immutable state.

Part 2

We generalise Classic Paxos and prove that it is conservative.

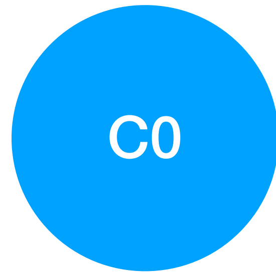
Part 3

We sketch three new algorithms made possible by our abstraction.

Part 1
Generalised solution to distributed consensus

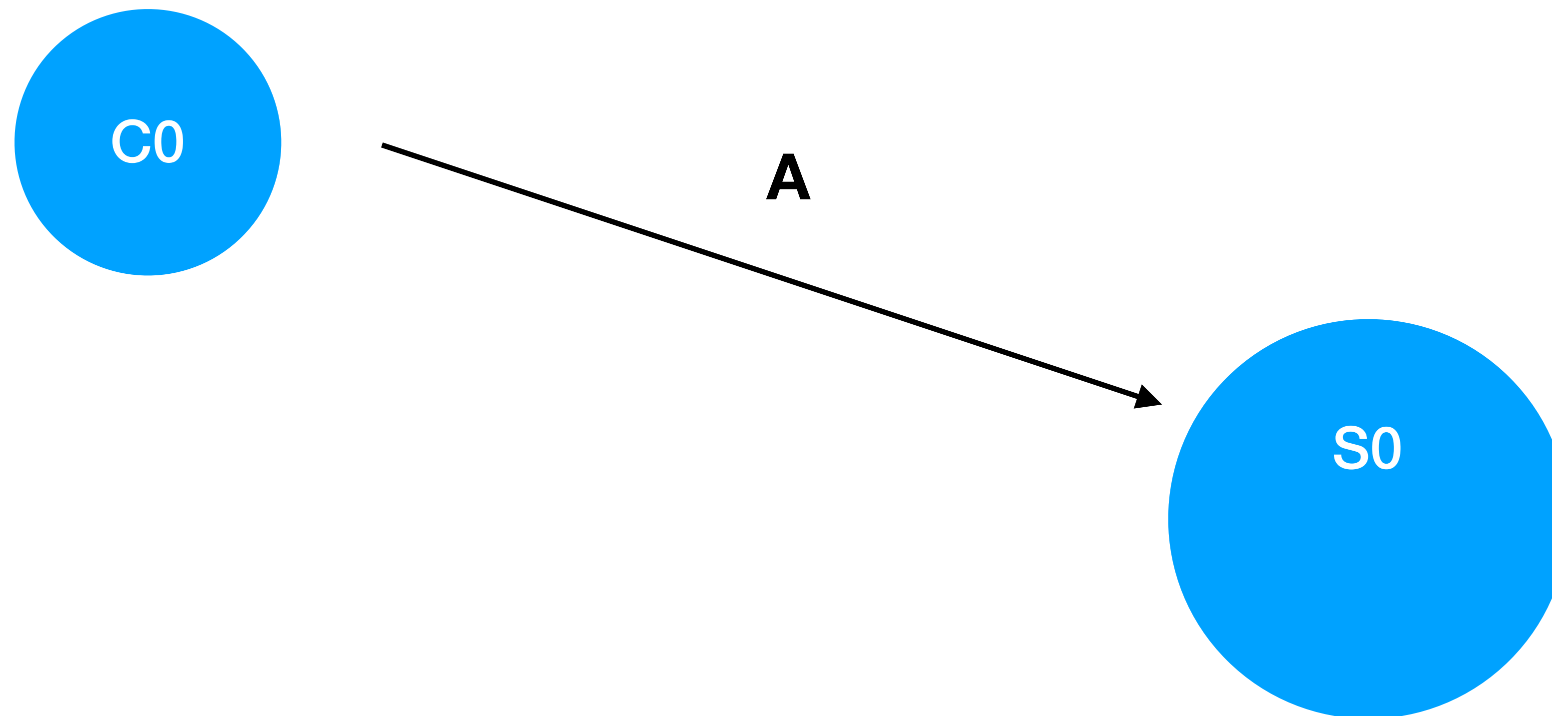
Single server

If we have one server, the algorithm is trivial.



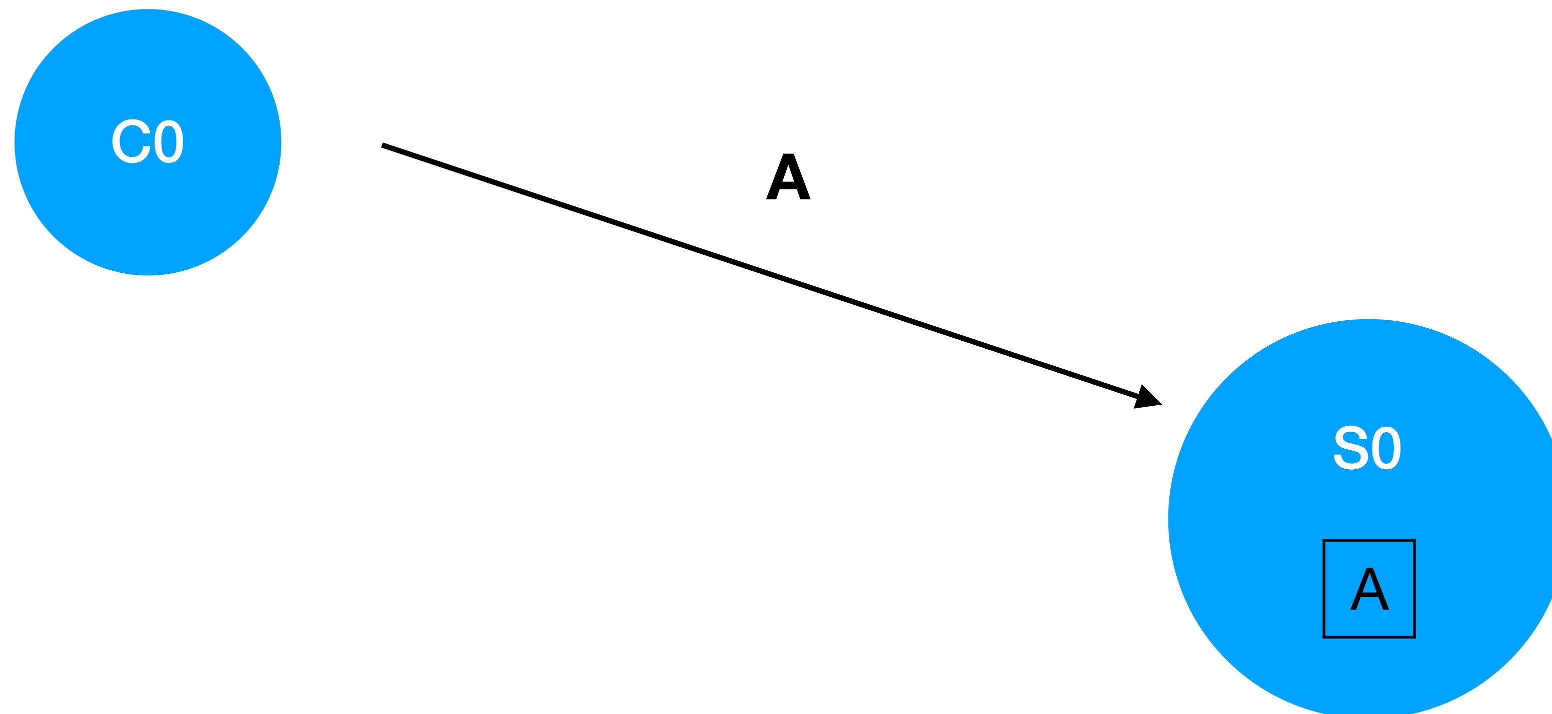
Single server

If we have one server, the algorithm is trivial.



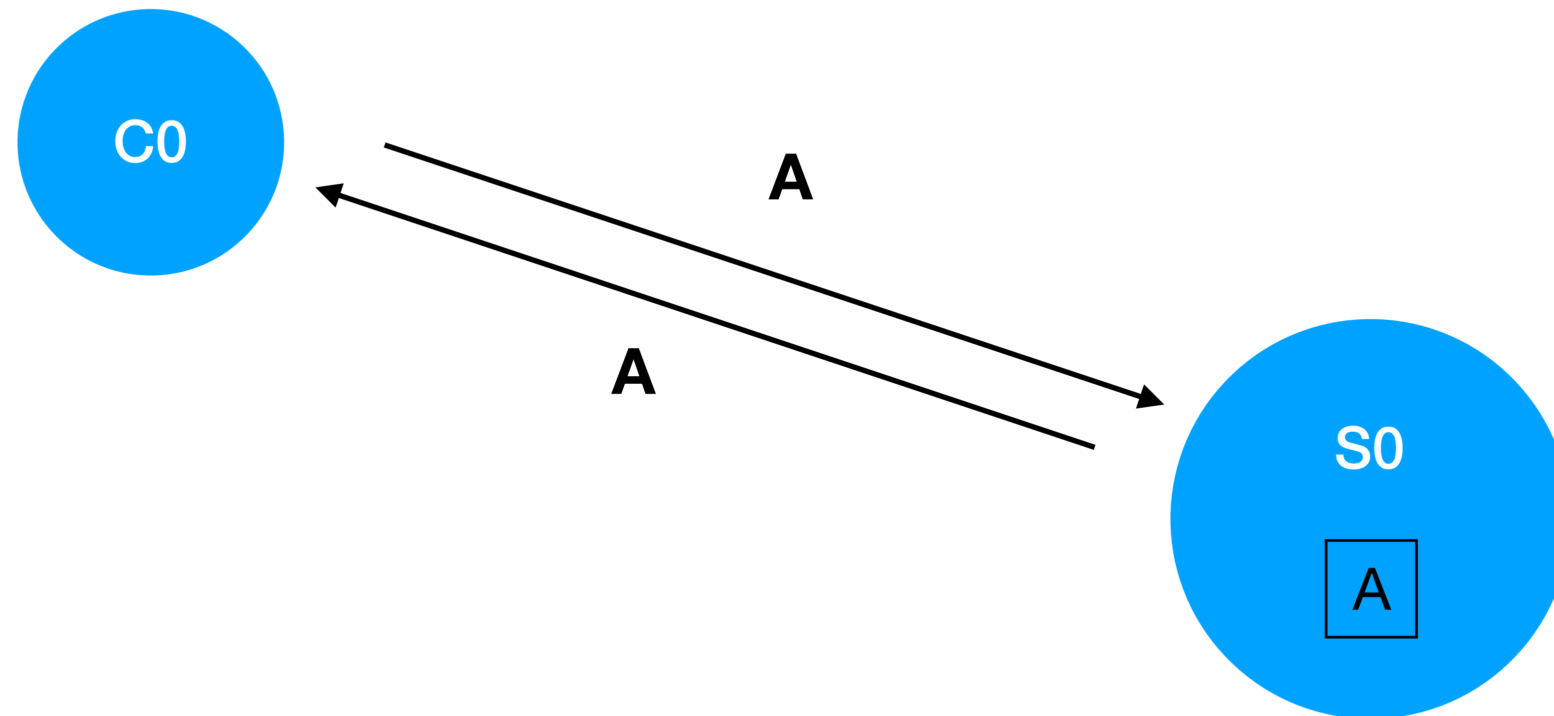
Single server

If we have one server, the algorithm is trivial.



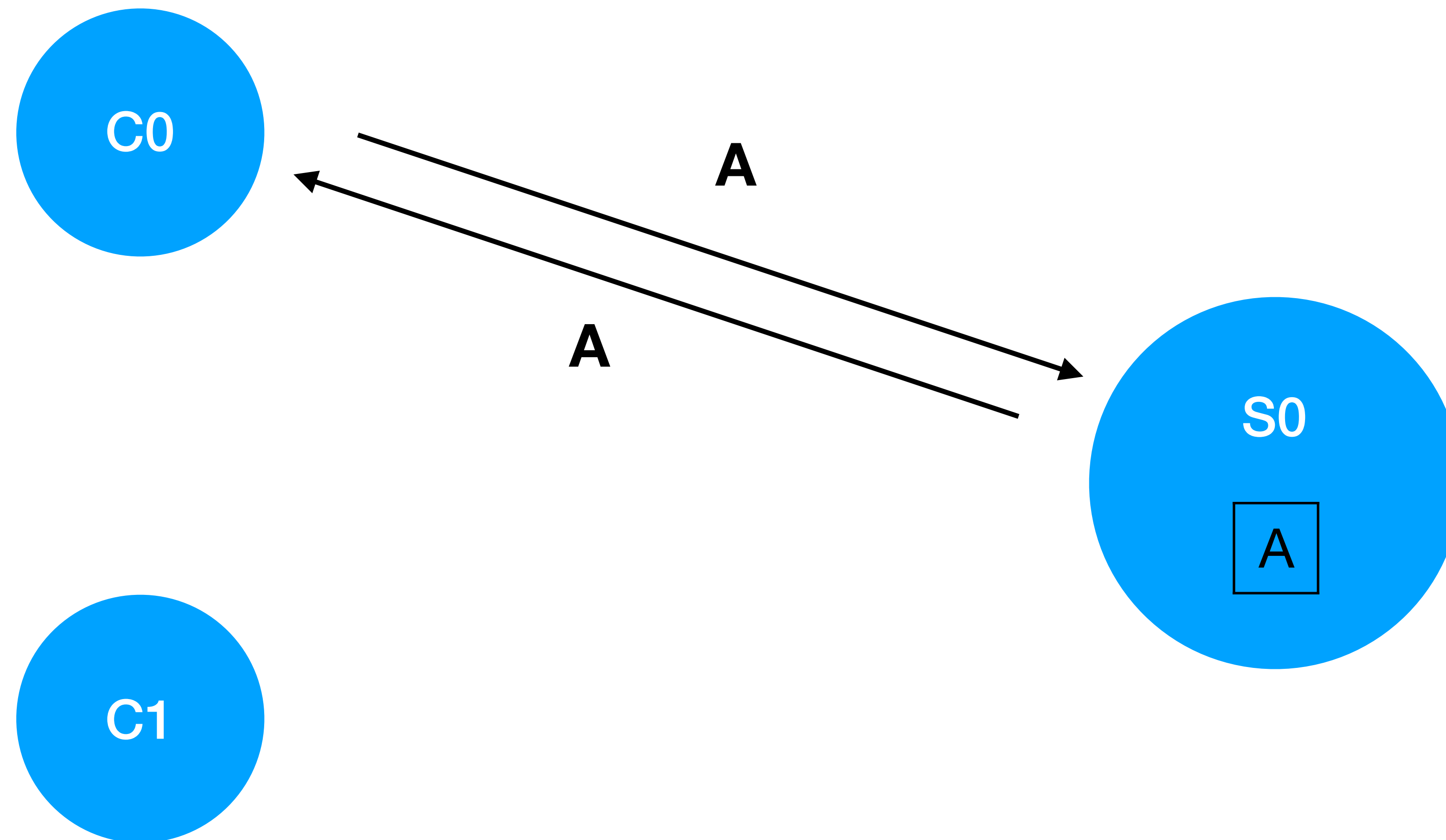
Single server

If we have one server, the algorithm is trivial.



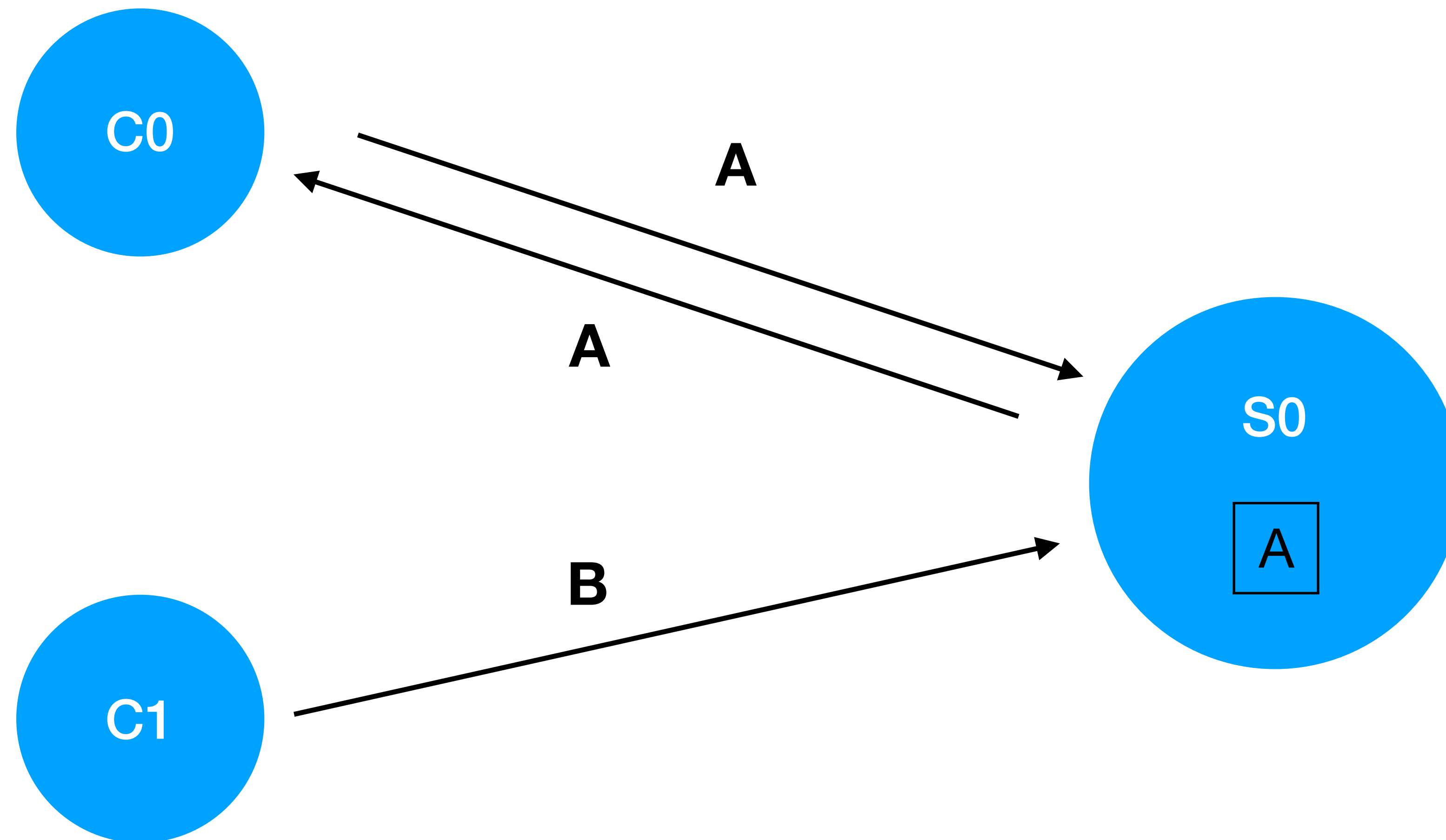
Single server

If we have one server, the algorithm is trivial.



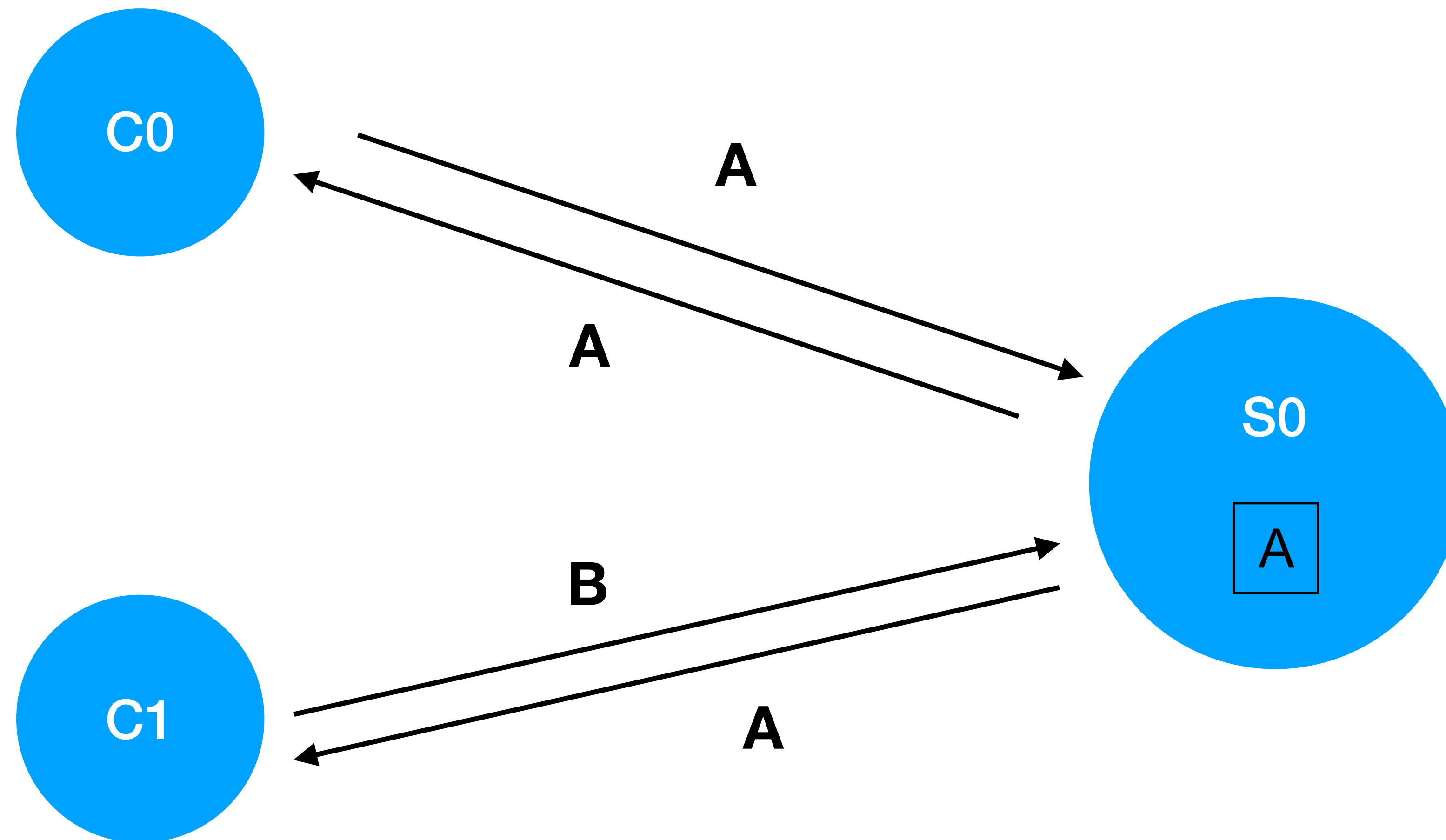
Single server

If we have one server, the algorithm is trivial.



Single server

If we have one server, the algorithm is trivial.



Multiple servers

- We could have multiple servers with copies of the register.

S0	S1	S2
A	B	C

Split vote
No decision

Multiple servers

- We could have multiple servers with copies of the register.

S0	S1	S2
A	B	C

Split vote
No decision

- Each server has a set of ordered write once persistent registers

	S0	S1	S2
0	-	A	A
1	-	-	A
2	A	A	A
3			-

Multiple servers

- We could have multiple servers with copies of the register.

S0	S1	S2
A	B	C

Split vote
No decision

- Each server has a set of ordered write once persistent registers

	S0	S1	S2
0	-	A	A
1	-	-	A
2	A	A	A
3			-

Server state table

Epochs

Nil value

Decision point

When has a client written sufficient copies of a value to say that this value has been decided?

Decision point

When has a client written sufficient copies of a value to say that this value has been decided?

To remain general, we say that a value is decided when it's written to specific subsets of servers at the same epoch.

We refer to these subsets as *quorums*.

Decision point

	S0	S1	S2
0	-	A	A
1	-	-	A
2	A	A	A
3	A		-

Configuration table maps epochs to quorums

e	Q
All	{{S0,S1},{S1,S2},{S0,S2}}

Decision point

	S0	S1	S2
0	-	A	A
1	-	-	A
2	A	A	A
3	A		-

A is decided

Configuration table maps epochs to quorums

e	Q
All	{{S0,S1},{S1,S2},{S0,S2}}

Decision point

	S0	S1	S2
0	-	A	A
1	-	-	A
2	A	A	A
3	A		-

A is decided

A is decided

Configuration table maps epochs to quorums

e	Q
All	{{S0,S1},{S1,S2},{S0,S2}}

Decision point

	S0	**S1**	**S2**	**S3**
0	B	B		A
1	-	-	A	A
2	A	A	A	
3	A			

e	**Q**
0	{{S0,S1,S2,S3}}
1+	{{S0,S1},{S2,S3}}

Decision point

	S0	S1	S2	S3
0	B	B		A
1	-	-	A	A
2	A	A	A	
3	A			

A is decided

A is decided

e	Q
0	{{S0,S1,S2,S3}}
1+	{{S0,S1},{S2,S3}}

However we can decide multiple values

	S0	S1	S2	S3
0	-	A	A	
1	C	C	A	A
2	A		A	

	S0	S1	S2
0	C	A	A
1	B	B	A
2	A	C	C
3	A		-

e	Q
0	{{S0,S1,S2,S3}}
1+	{{S0,S1},{S2,S3}}

e	Q
All	{{S0,S1},{S1,S2},{S0,S2}}

Safety

Only one value should ever be decided

Before a client writes a value in epoch *e* it must ensure that:

1. No other values are decided for epoch *e*

2. No other values are decided for epochs *0* to *e-1*

Epoch allocation rule

We choose between two modes for each epoch:

- **Exclusive value** requires only one value will be written to each epoch. For example, by allocating epochs to clients round robin.

- **Non-exclusive values** allows any value to written to any epoch. This requires that the quorums for a given epoch intersect.

Example: exclusive values

	S0	S1	S2	S3
0	-	-	-	A
1	B	B	-	-
2		-	-	
3	B	B	B	B

B is decided

Configuration table now includes epoch allocation

e	v	Q
0,2,…	C0	Any 2 of {S0,S1,S2,S3}
1,3,…	C1	

Example: non-exclusive values

	S0	S1	S2
0	B	A	A
1	-	-	A
2	A	A	A
3	A		-

A is decided

e	v	Q
All	Any	{{S0,S1},{S1,S2},{S0,S2}}

Example: hybrid values

	S0	S1	S2	S3
0	B	B		A
1	-	-	A	A
2	A	A	A	
3	A			

e	v	Q
0	Any	{{S0,S1,S2,S3}}
1,4,…	C0	{{S0,S1},{S2,S3}}
2,5,…	C1	
3,6,…	C2	

Example: hybrid values

	S0	S1	S2	S3
0	B	B		A
1	-	-	A	A
2	A	A	A	
3	A			

A is decided

A is decided

e	v	Q
0	Any	{{S0,S1,S2,S3}}
1,4,…	C0	
2,5,…	C1	{{S0,S1},{S2,S3}}
3,6,…	C2	

However we can decide multiple values

	S0	S1	S2	S3
0	-	A	A	
1	C	C	A	A
2	A		A	

	S0	S1	S2
0	C	A	A
1	B	B	A
2	A	C	C
3	A		-

e	Q
0	{{S0,S1,S2,S3}}
1+	{{S0,S1},{S2,S3}}

e	Q
All	{{S0,S1},{S1,S2},{S0,S2}}

However we can decide multiple values

	S0	S1	S2	S3
0	-	A	A	
1	C	C	A	A
2	A		A	

e	Q
0	{{S0,S1,S2,S3}}
1+	{{S0,S1},{S2,S3}}

	S0	S1	S2
0	C	A	A
1	B	B	A
2	A	C	C
3	A		-

e	Q
All	{{S0,S1},{S1,S2},{S0,S2}}

Safety

Only one value should ever be decided

Before a client writes a value in epoch *e* it must ensure that:

1. No other values are decided for epoch *e*

Epoch allocation rule

2. No other values are decided for epochs *0* to *e-1*

Client state

Each client maintains their own copy of the state table. The clients construct this state table using two operations:

- Clients can **read** values from any register
- Clients can **write nil** values to any register

From the state table, the client can track decisions.

When a client learns that value *v* has been decided then the client can return *v.*

Client write rule

Before a client writes a value v to epoch e it must ensure that either:

- No decisions are reached for epochs *0* to *e-1, or*
- All decisions which are reached for epochs *0* to *e-1* are for value v

However we can decide multiple values

	S0	S1	S2	S3
0	-	A	A	
1	C	C	A	A
2	A		A	

	S0	S1	S2
0	C	A	A
1	B	B	A
2	A	C	C
3	A		-

e	Q
0	{{S0,S1,S2,S3}}
1+	{{S0,S1},{S2,S3}}

e	Q
All	{{S0,S1},{S1,S2},{S0,S2}}

However we can decide multiple values

	S0	S1	S2	S3
0	-	A	A	
1	C	C	A	A
2	A		A	

e	Q
0	{{S0,S1,S2,S3}}
1+	{{S0,S1},{S2,S3}}

	S0	S1	S2
0	C	A	A
1	B	B	A
2	A	C	C
3	A		-

e	Q
All	{{S0,S1},{S1,S2},{S0,S2}}

However we can decide multiple values

	S0	S1	S2	S3
0	-	A	A	
1	C	C	A	A
2	A	[illegible]	A	

e	Q
0	{{S0,S1,S2,S3}}
1+	{{S0,S1},{S2,S3}}

	S0	S1	S2
0	C	A	A
1	B	B	A
2	A	C	C
3	[illegible]		-

e	Q
All	{{S0,S1},{S1,S2},{S0,S2}}

Safety

Only one value should ever be decided

Before a client writes a value in epoch *e* it must ensure that:

1. No other values are decided for epoch *e*
2. No other values are decided for epochs *0* to *e-1*

Part 1 - Summary

We have proposed an abstract algorithm for reaching agreement over a single value

- **Safety** - The immutable state allows us to easily reason about the safety and correctness of algorithms.
- **Flexibility** - Implementations may choose their own configuration and algorithm, provided they follow the epoch allocation and client write rules.

Part 2

Generalising Classic Paxos

Classic Paxos

We can implement Classic Paxos using our abstract consensus algorithm.

All epochs are exclusive and allocated round robin to clients. We use majorities for quorums.

e	**v**	**Q**
0,2,…	C0	{{S0,S1},{S0,S2},{S1,S2}}
1,3,…	C1	

Classic Paxos - Phase one

- The client chooses an allocated epoch *e* and sends *prepare(e)* to all servers.
- Provided register e is unwritten, each server writes nil in any unwritten registers from *0* to *e-1* and replies with the epoch *f* and value *w* of the greatest non-nil register using *promise(e,f,w)*

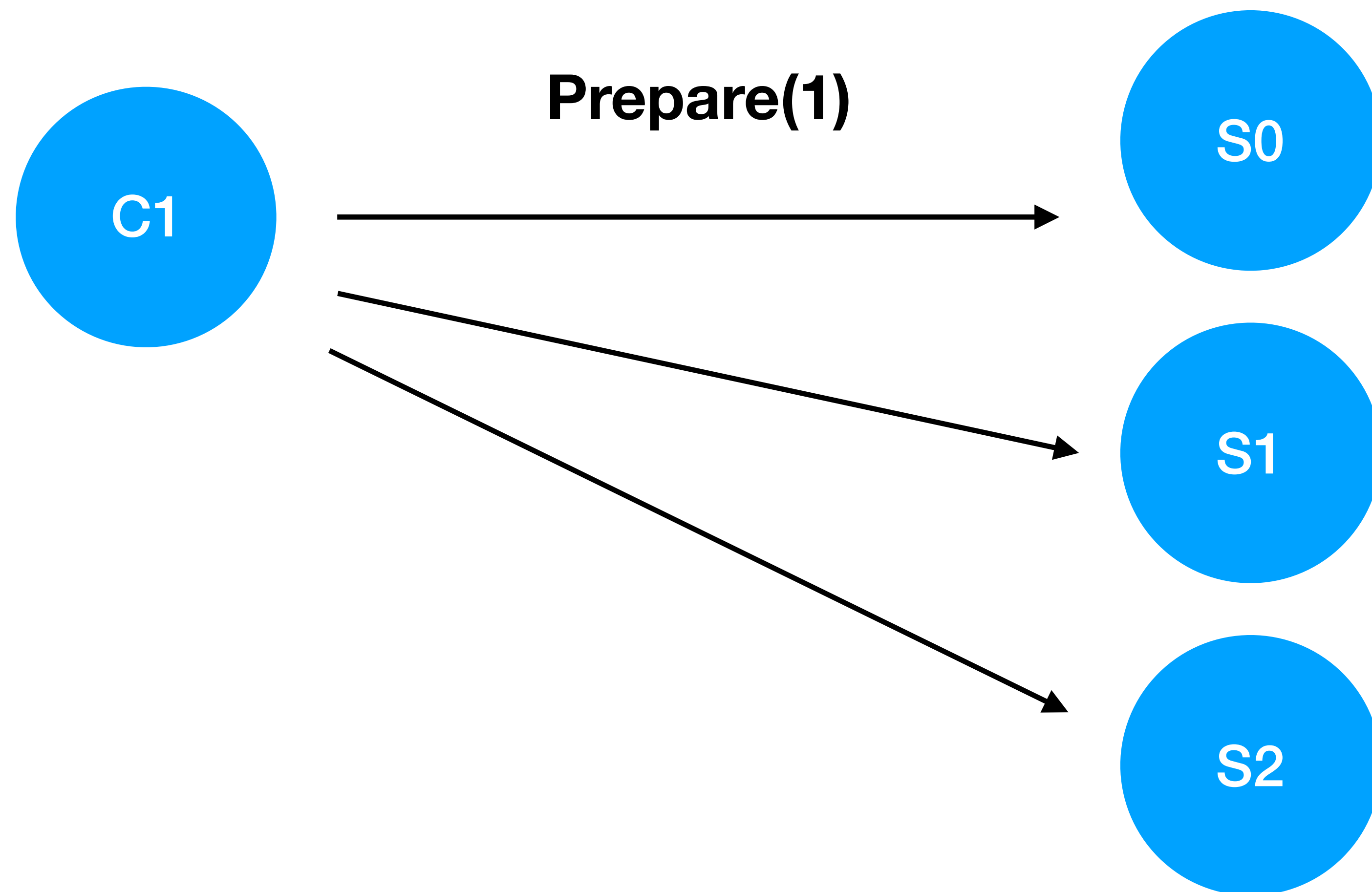
Classic Paxos - Phase two

- After a majority of servers reply, the client chooses the value *v* with the greatest epoch or its own value if none. Client sends *propose(e,v)* to all servers.

- Provided e is unwritten, each server writes nil to any unwritten registers from *0* to *e-1 and* value *v* to the register at epoch *e*. The server replies to the client using *accept(e)*

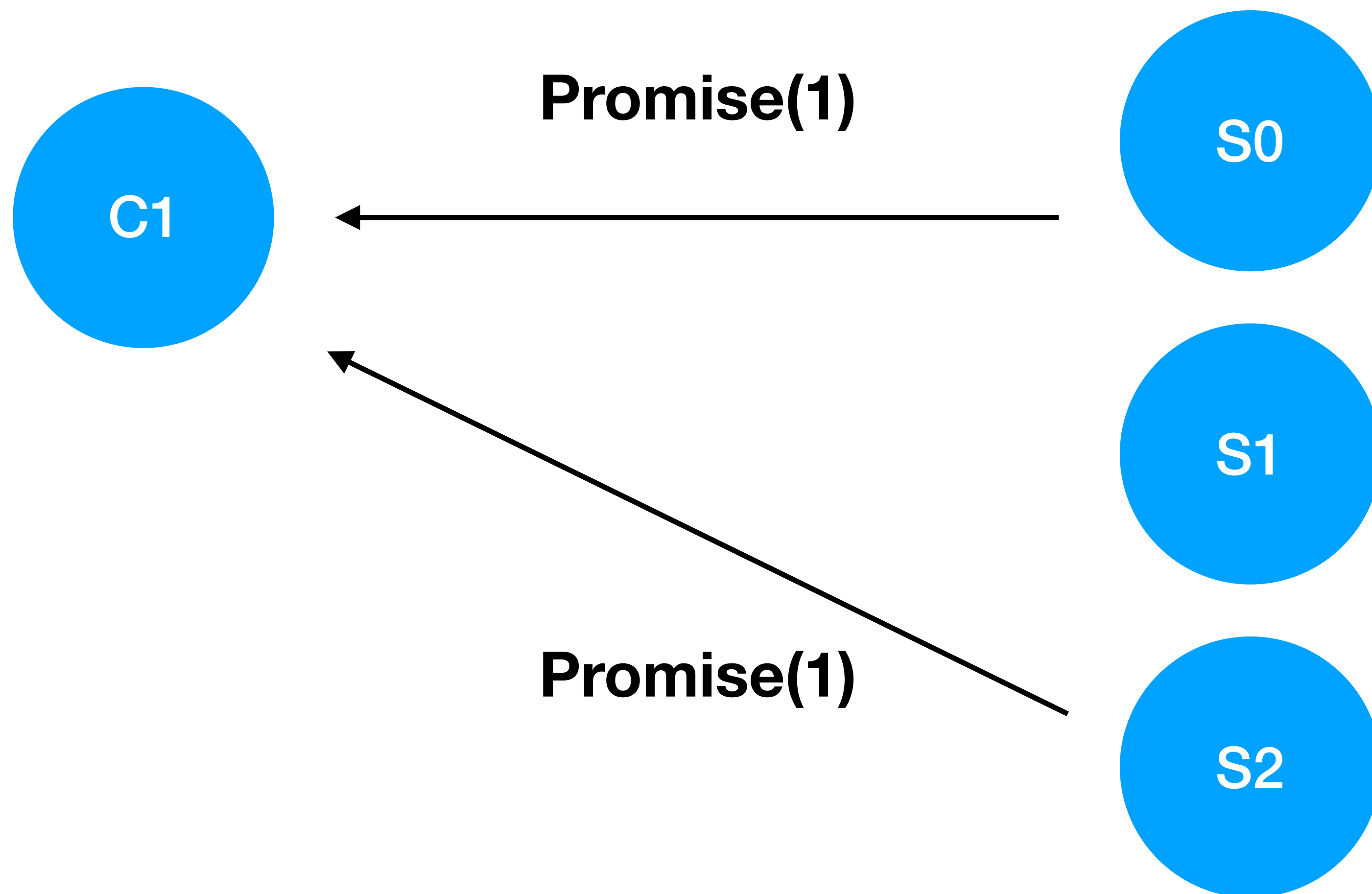
- The client terminates when *accept(e) is received from the majority of servers.*

Example - Phase one



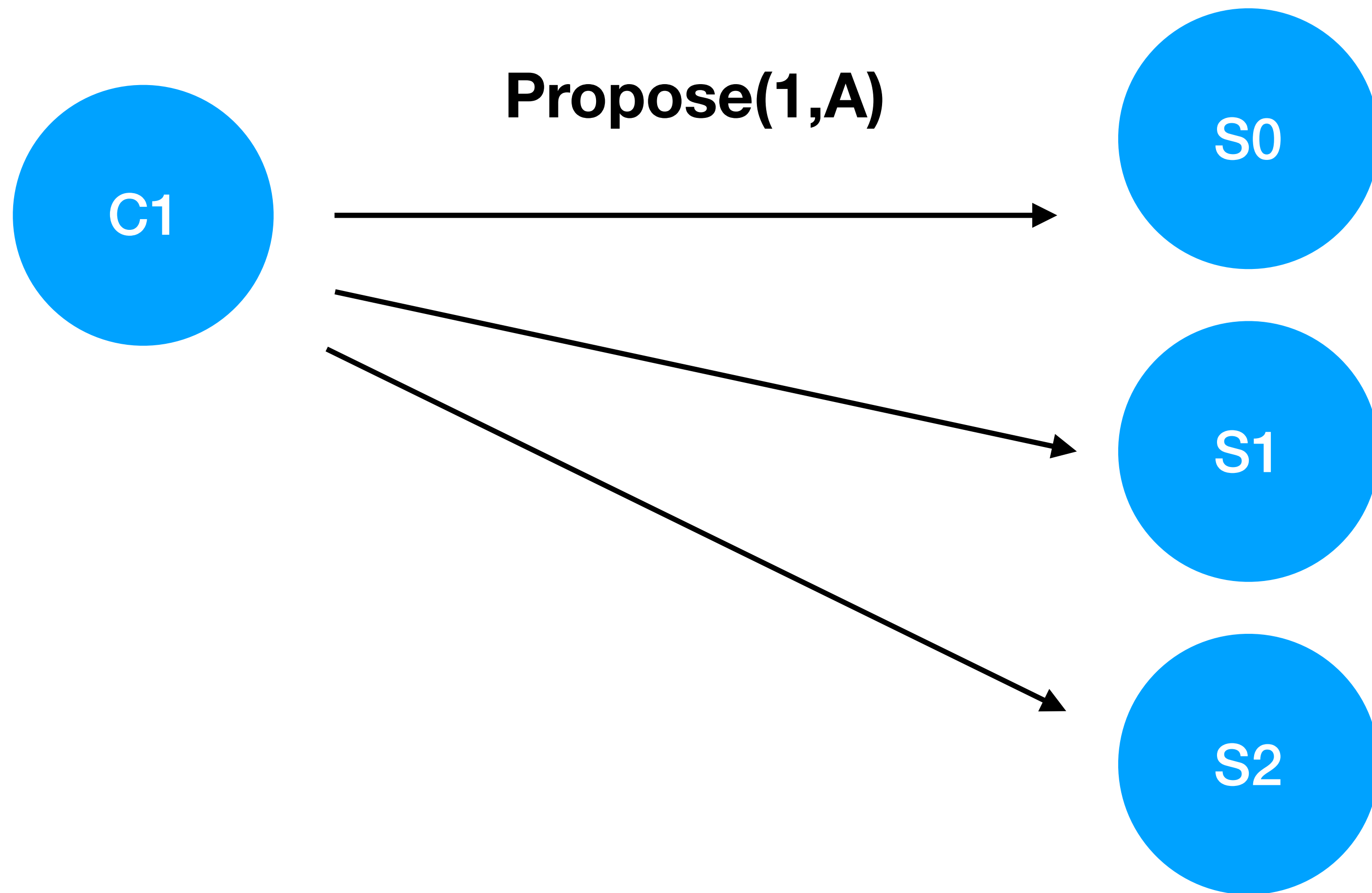
	S0	S1	S2
0			
1			
2			
3			

Example - Phase one



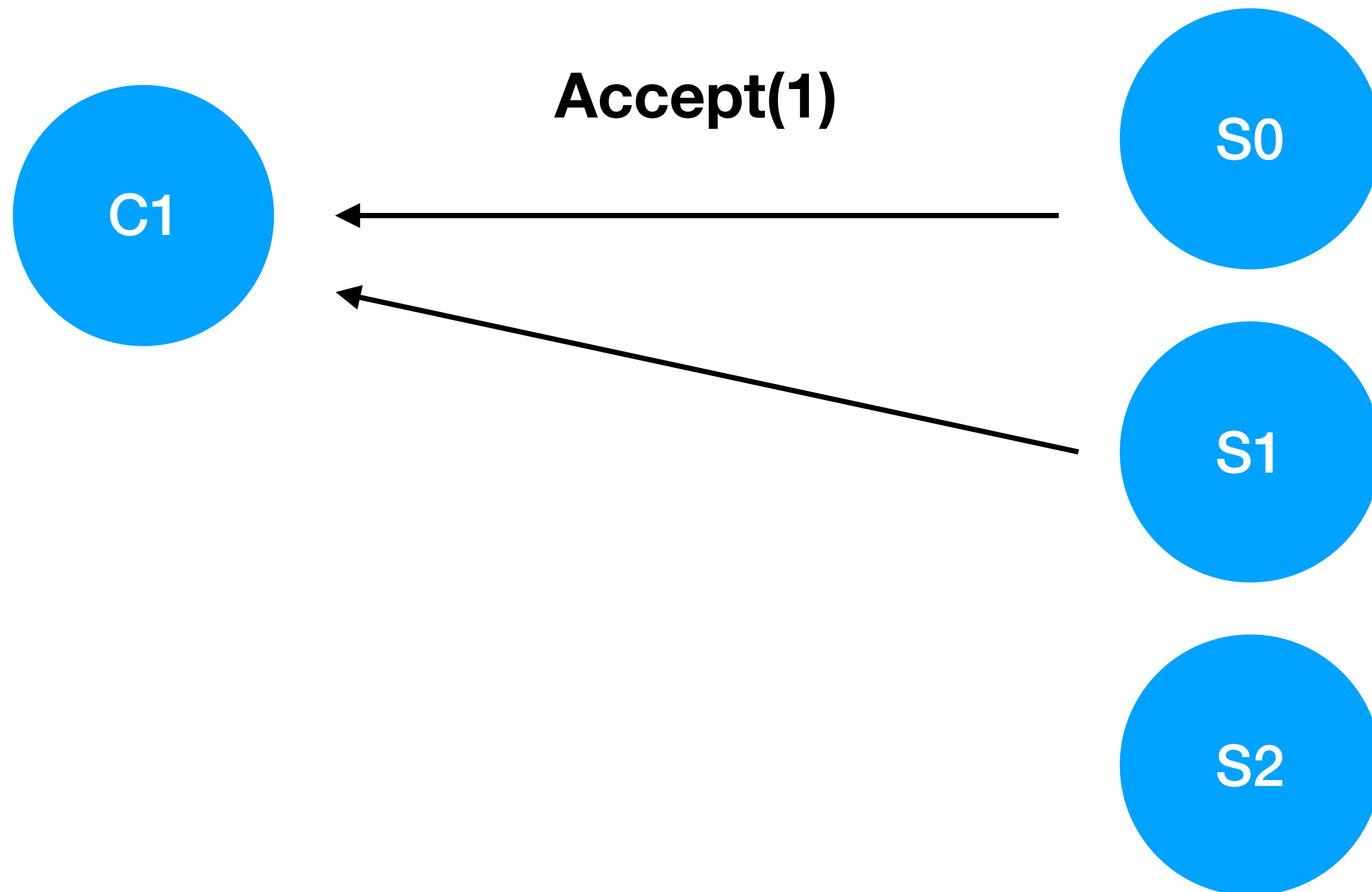
	S0	S1	S2
0	-	-	-
1			
2			
3			

Example - Phase two



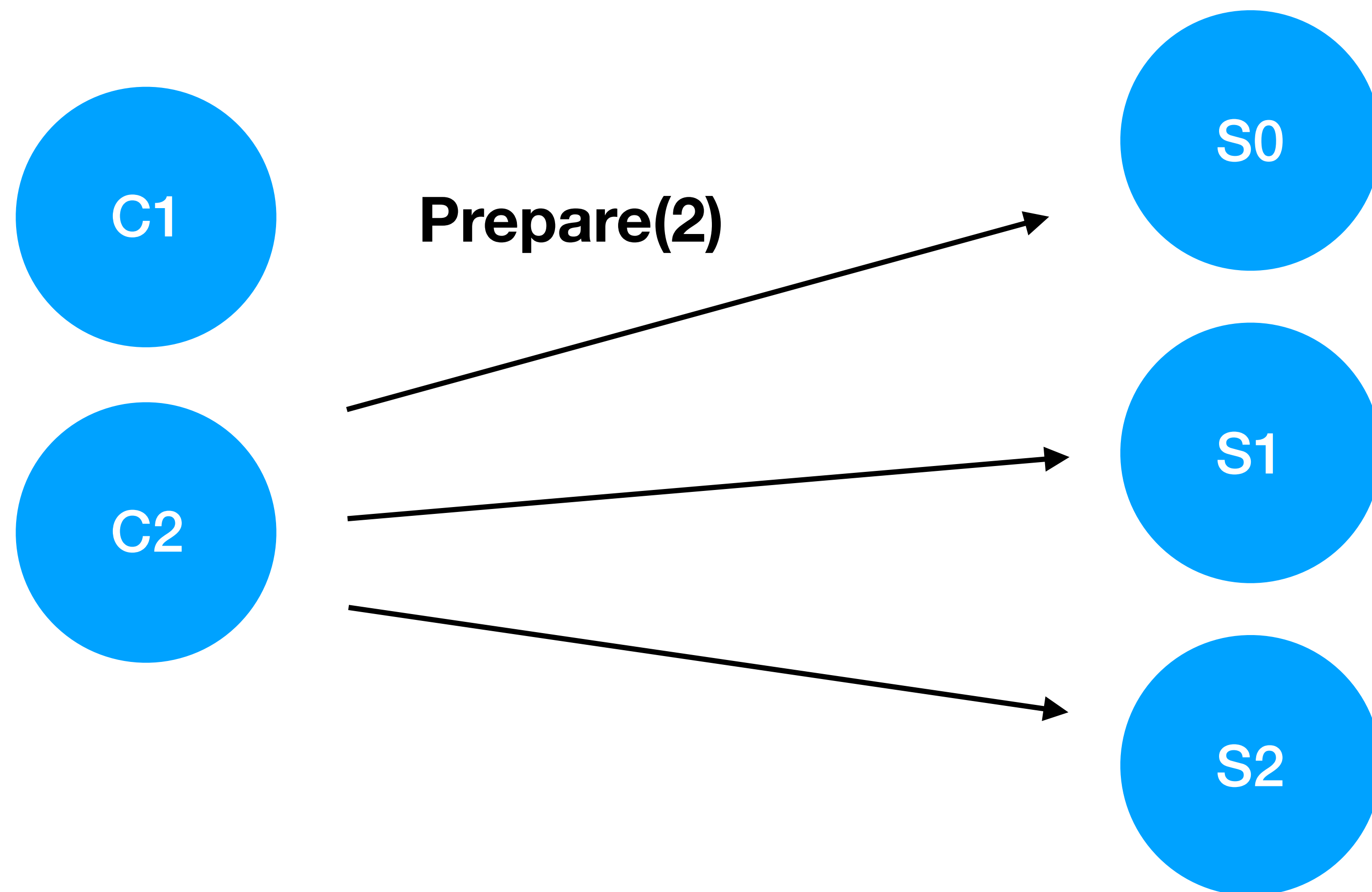
	S0	S1	S2
0	-	-	-
1			
2			
3			

Example - Phase two



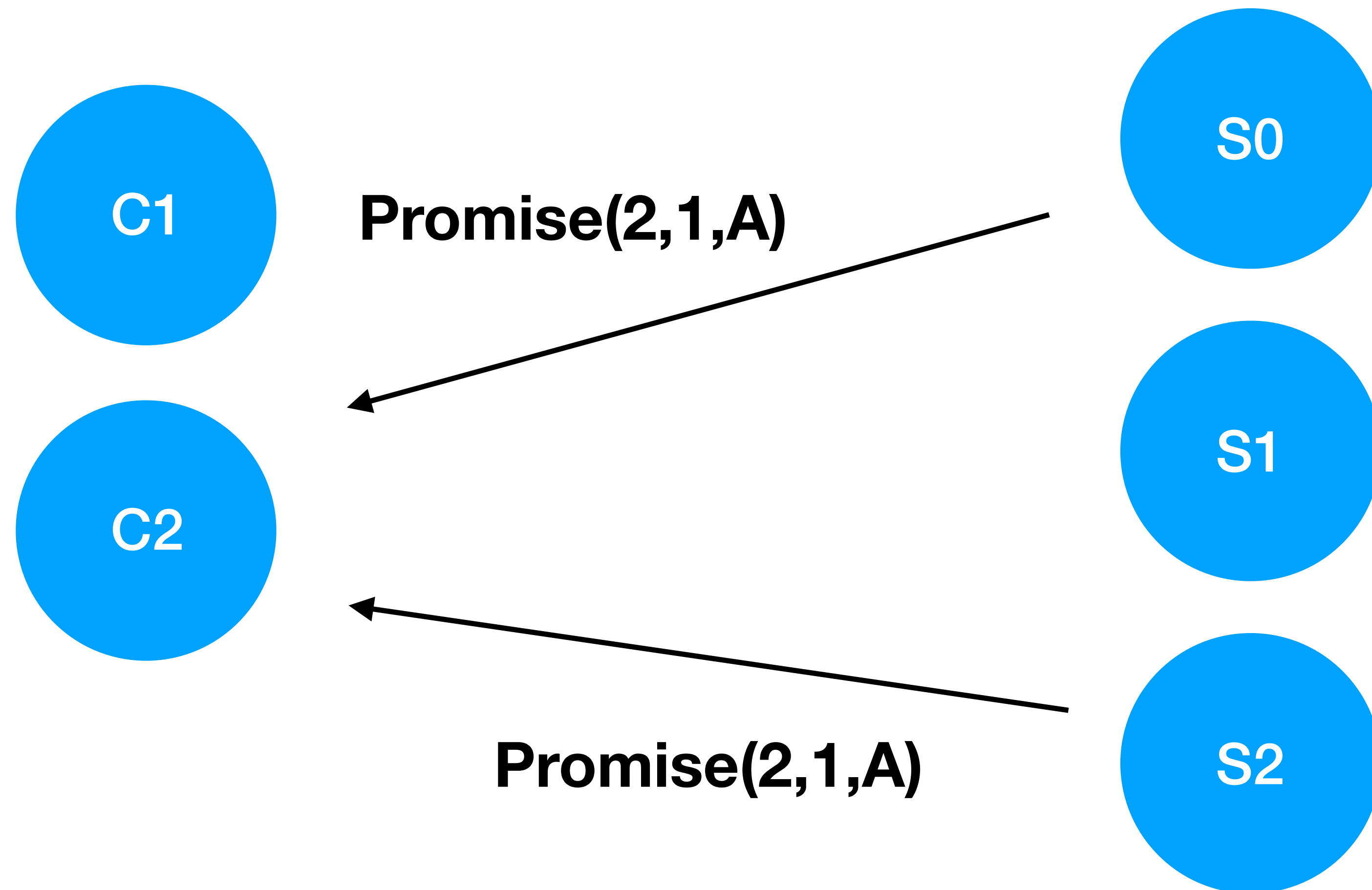
	S0	S1	S2
0	-	-	-
1	A	A	A
2			
3			

Example - Phase one



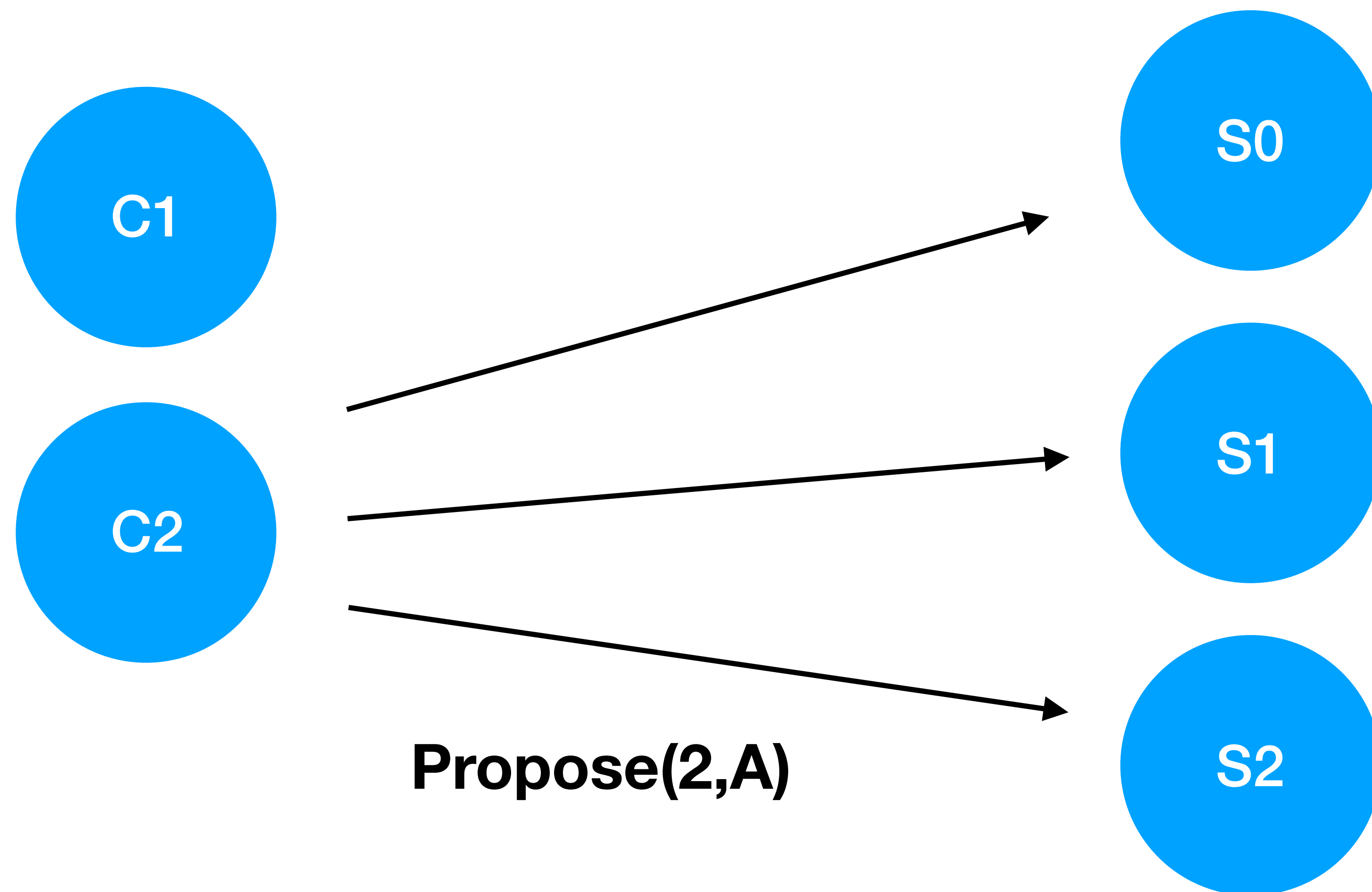
	S0	S1	S2
0	-	-	-
1	A	A	A
2			
3			

Example - Phase one



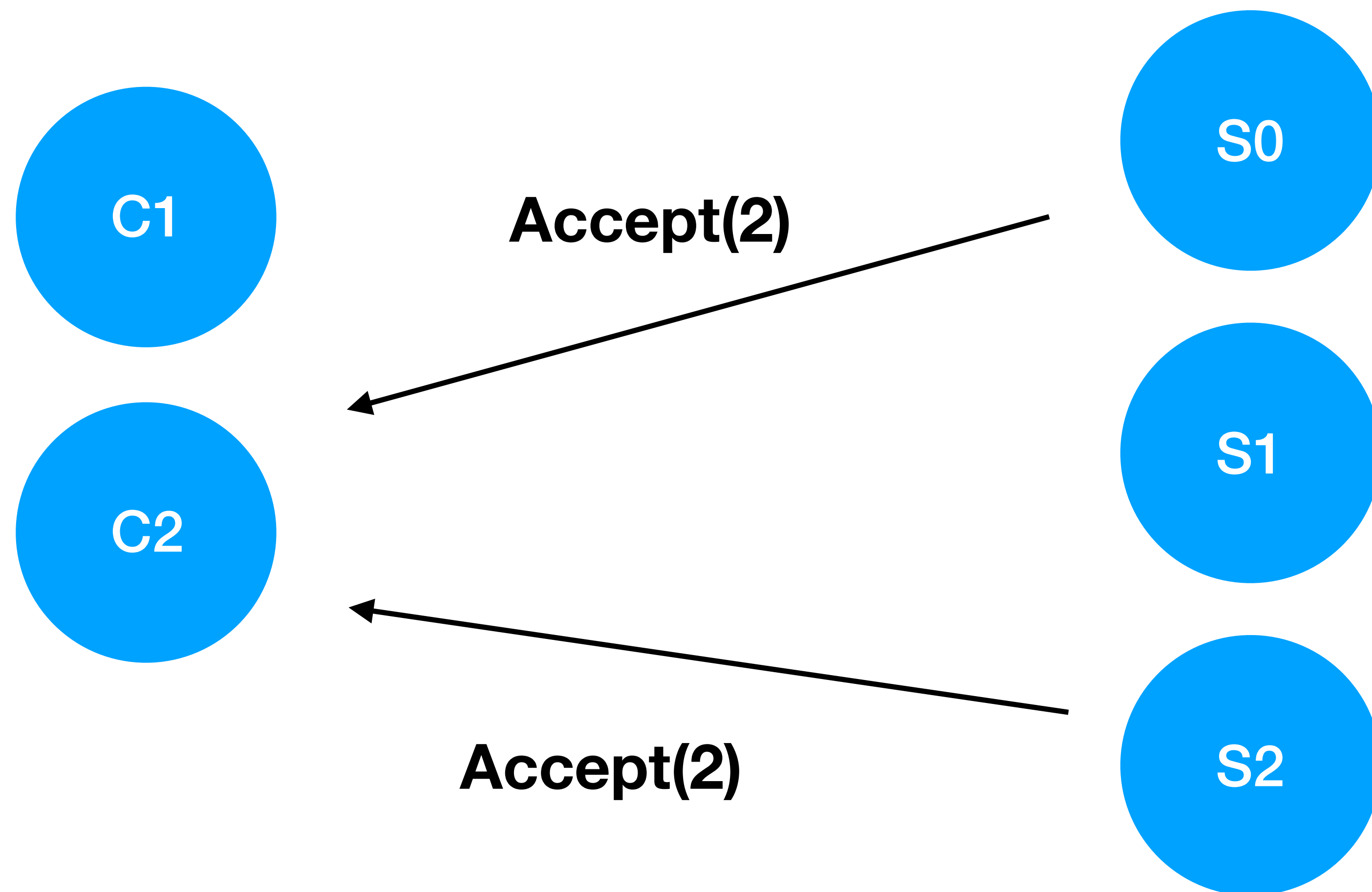
	S0	S1	S2
0	-	-	-
1	A	A	A
2			
3			

Example - Phase two



	S0	S1	S2
0	-	-	-
1	A	A	A
2			
3			

Example - Phase two



	S0	S1	S2
0	-	-	-
1	A	A	A
2	A	A	A
3			

Safety of Classic Paxos

Only one value should ever be decided

Before a client writes a value in epoch *e* it must ensure that:

1. No other values are decided for epoch *e*

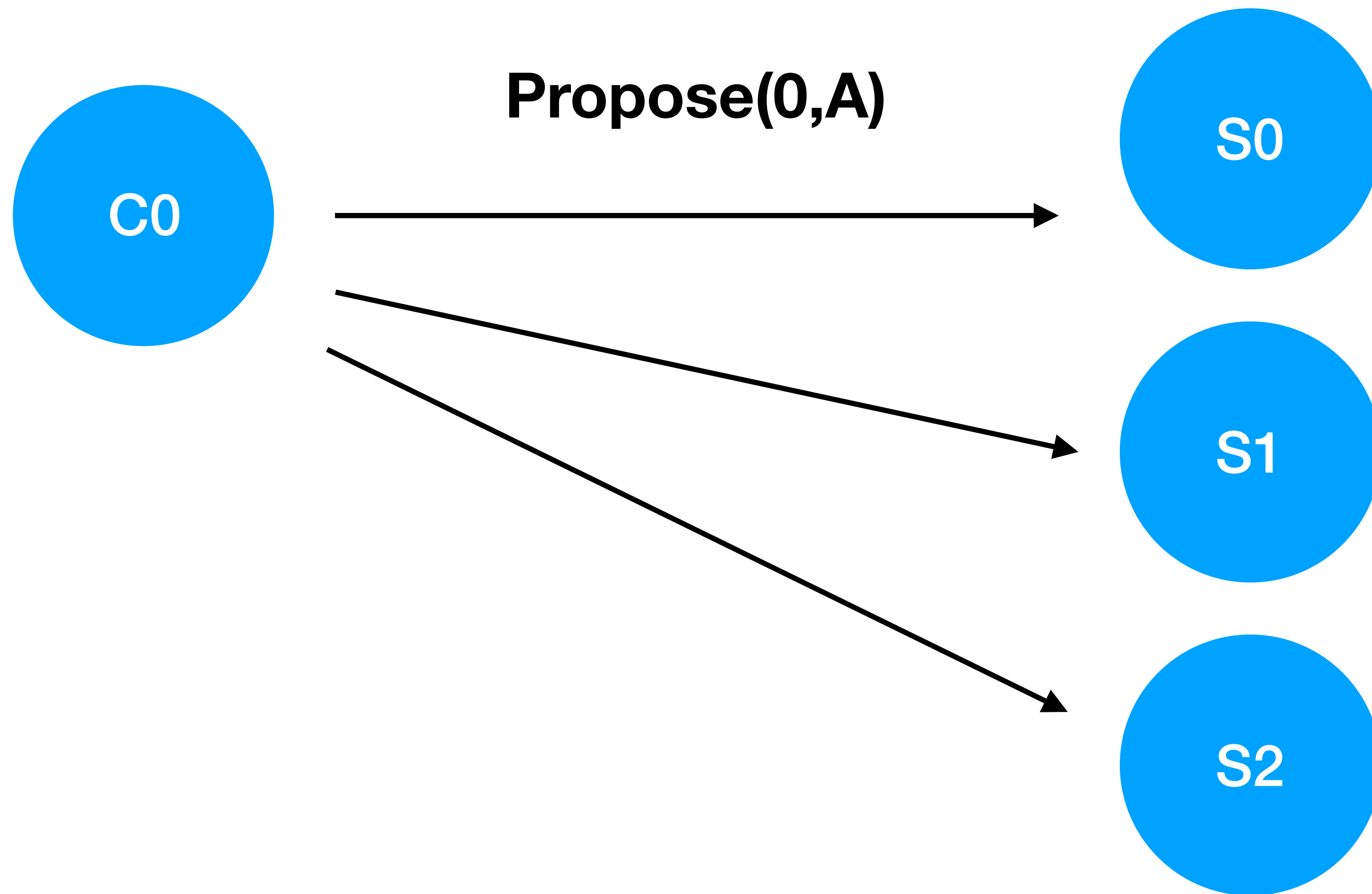
2. No other values are decided for epochs *0* to *e-1*

Quorum intersection

Original requirement - Paxos requires that each of its two phases use a quorum of servers and that any two quorums must intersect.

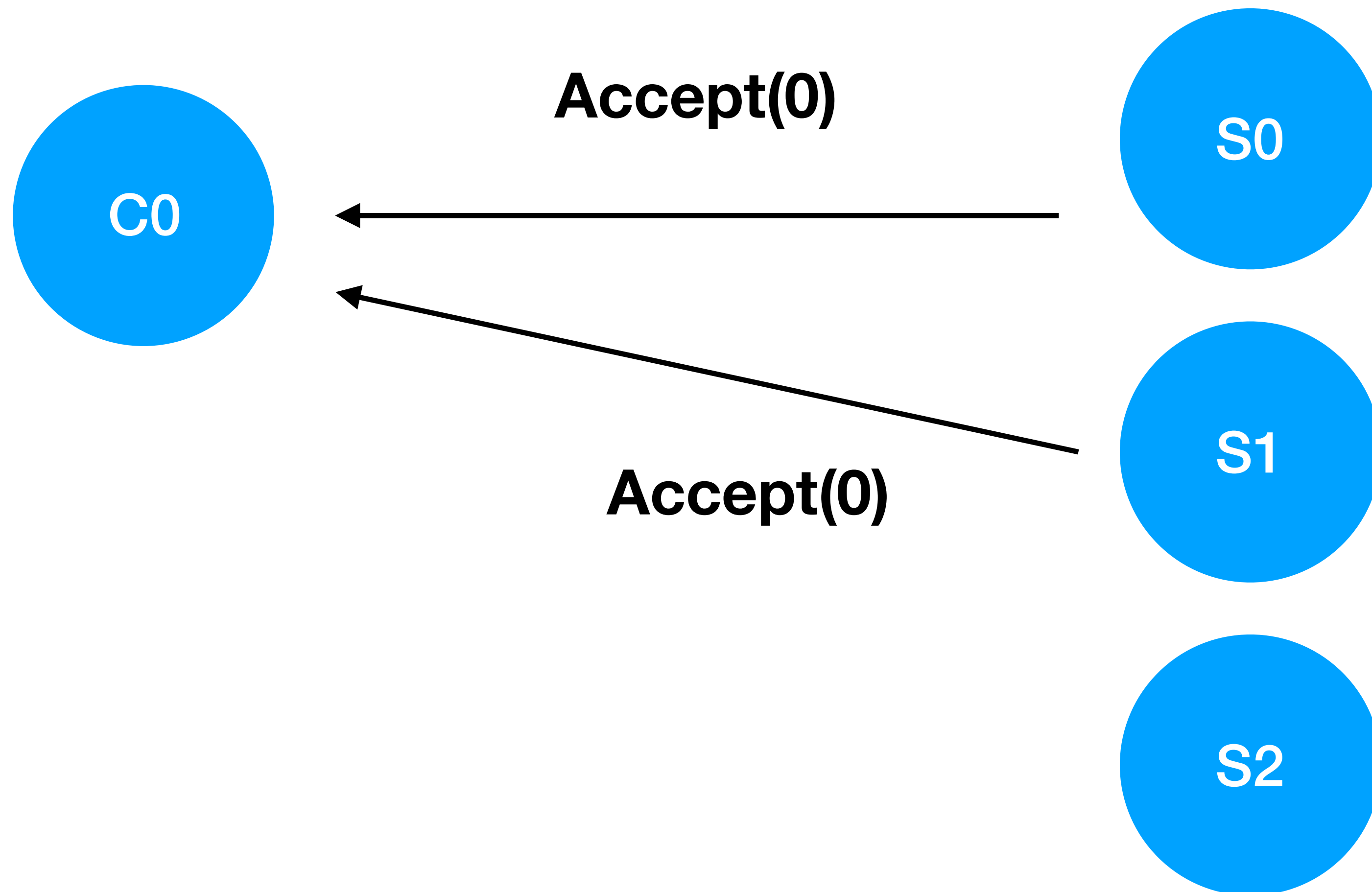
Revised requirement - A client in epoch *e* must get at least one server from each quorum of epochs *0* to *e-1* to participate in phase one.

Example - Phase two



	S0	S1	S2
0			

Example - Phase two



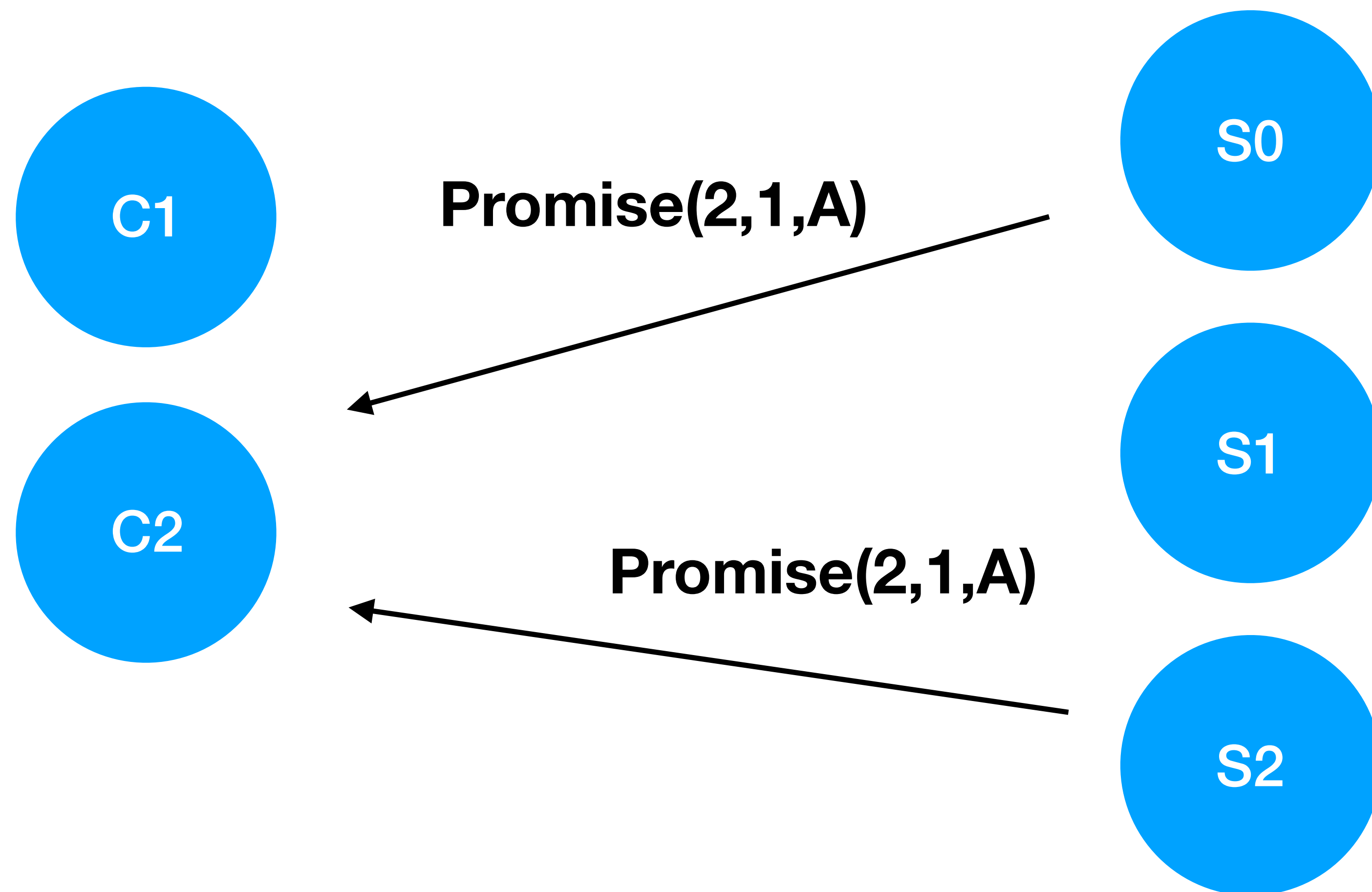
	S0	S1	S2
0	A	A	A

Early completion of phase one

Original requirement - Paxos requires that a phase one quorum of servers always participant in phase one.

Revised requirement - If a client reads a non-nil value from epoch *e* then it no longer needs to intersect with epochs *0* to *e.*

Example - Phase one



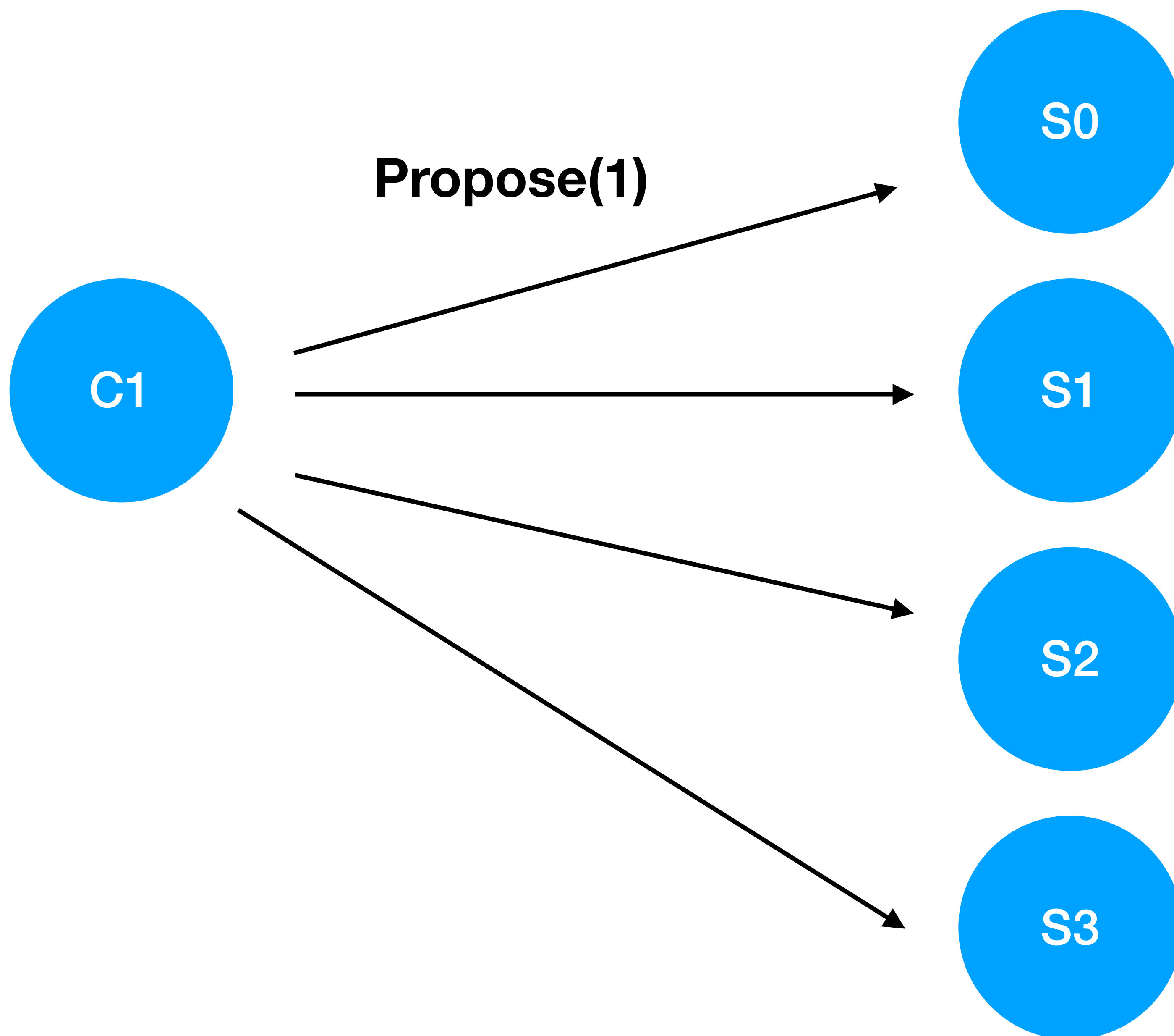
	S0	S1	S2
0	-	-	-
1	A	A	A
2			
3			

Value Selection

Original requirement - Paxos requires that the value with the greatest epoch is proposed in phase two. Otherwise, if no values were returned in phase two, then any value may be proposed.

Revised requirement - The client need only propose a value if it may have been chosen by a quorum

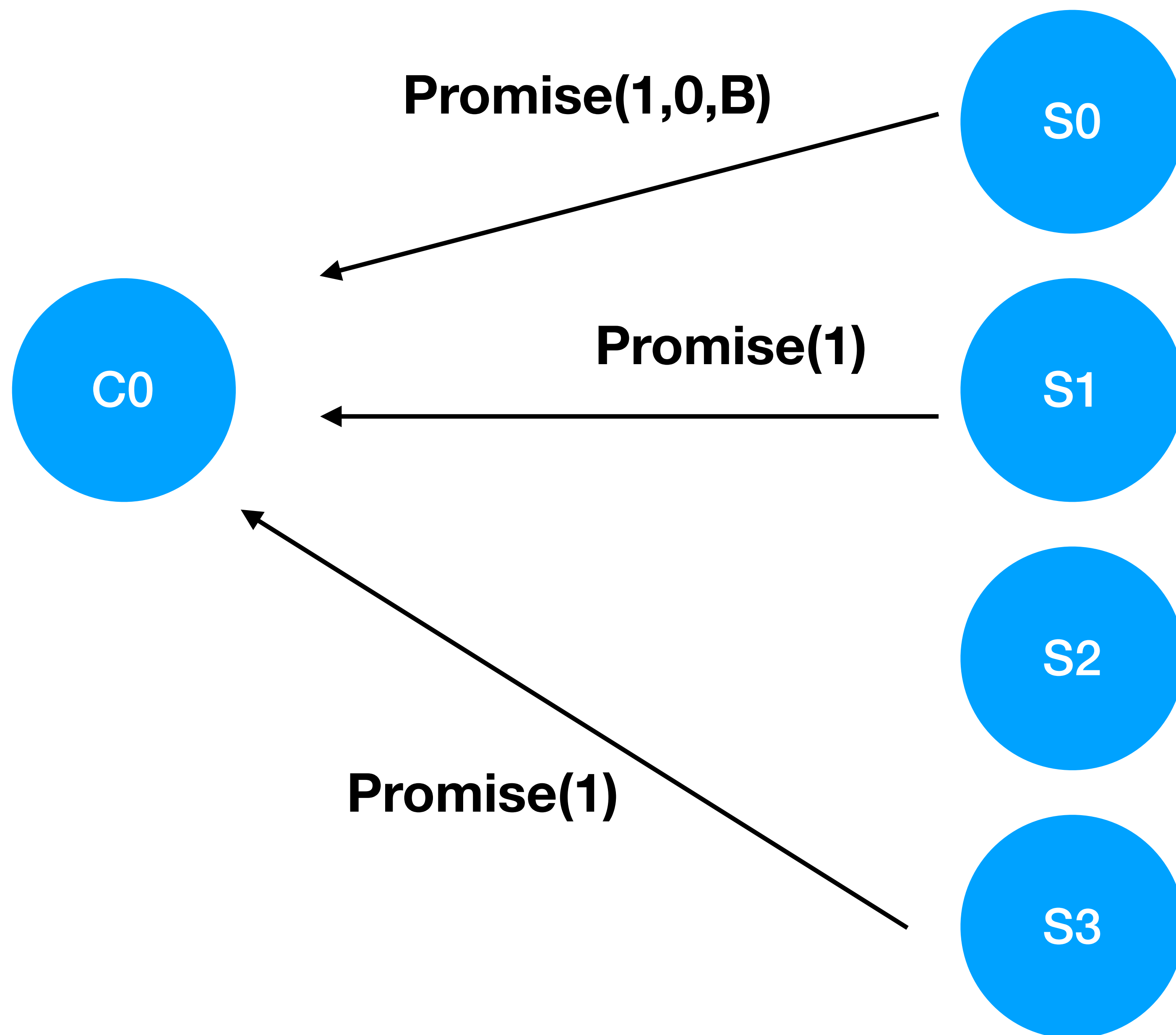
Example: Phase one



	S0	S1	S2	S3
0	B			

e	v	Q
0,2,…	C0	{{S0,S1}, {S1,S2}}
1,3,…	C1	

Example: Phase one



	S0	S1	S2	S3
0	B	-	-	-

e	v	Q
0,2,…	C0	{{S0,S1}, {S1,S2}}
1,3,…	C1	

Part 2 - Summary

We can relax the requirements of Classic Paxos in the following three 3 areas:

- Quorum intersection
- Phase one completion
- Value selection

Part 3
Examples

Current Reality

	Classic Paxos	**Multi Paxos**
Minimum number of 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***?

Co-located consensus

Goal: In a co-located system, allow any client to decide a value in 1 RTT and tolerate any minority failure.

Co-located consensus

Round robin allocation of epochs to servers

e	v
0, 3, …	C0
1, 5, …	C1
2, 6, …	C2

Epochs partitioned at 20

e	Q
0 to 19	{{S0,S1,S2}}
20+	2 of {S0,S1,S2}

Co-located consensus

Fast path (epochs 0-19)

Execute phase one locally, followed by phase two with all participants.

If unsuccessful, try slow path.

Slow path (epochs 20+)

Classic two phase paxos with majorities

Co-located consensus

Pros

- If all servers are up then all clients can terminate in 1 RTT
- If two clients collide, one will succeed and the other will retry.

Cons

- Requires co-location
- 2 RTTs are needed if a server is slow/unavailable
- Clients proposing the same value can collide

Supermajority consensus

Goal: Allow any client to decide a value in 1 RTT and tolerate any minority failure.

Supermajority consensus

e	v	Q
0	Any	4 of {S0,S1,S2,S3,S4}
1, 4, …	C0	3 of {S0,S1,S2,S3,S4}
2, 5, …	C1	
3, 6, …	C2	

Supermajority consensus

Fast path (epoch 0)

Execute phase two with client value and epoch 0.

If unsuccessful, try slow path.

Slow path (epochs 1+)

Classic two phase paxos with majorities

Supermajority consensus

Pros

- If at least 4 of 5 servers are up and no collisions occur then all clients can terminate in 1 RTT.
- Clients proposing the same value do not collide.

Cons

- 2 RTTs are needed if 2 or more servers are slow/ unavailable or a collision occurs

Supermajority consensus

Pros

- If at least 4 of 5 servers are up and no collisions occur then all clients can terminate in 1 RTT.
- Clients proposing the same value do not collide.

Cons

- 2 RTTs are needed if 2 or more servers are slow/ unavailable or a collision occurs

Note: This algorithm is generalised Fast Paxos

Binary consensus

Goal: A binary decision algorithm in which any client can decided value 0 in 1 RTT and tolerate any minority failure

Binary consensus

e	v	Q
0, 2, …	0	2 of {S0,S1,S2}
1, 3, …	1	

Epochs allocated to values round robin

Binary consensus

Fast path (epoch 0)

If client value is 0, then execute phase two for value 0.

If unsuccessful, try slow path.

Slow path (epochs 1+)

Classic two phase paxos with majorities. If proposed value does not match epoch then restart.

Binary consensus

Pros

- Client proposing value 0 can complete in 1 RTT.
- Clients proposing the same value do not collide.

Cons

- Clients proposing value 1 need 2 RTTs to complete.
- Only works for reaching consensus over a binary value

Binary consensus

Pros

- Client proposing value 0 can complete in 1 RTT.
- Clients proposing the same v
collide

Cons

- Clients proposing value 1 need 2 RTTs to complete.
- Only works for reaching
a binary

Note: This algorithm generalises to any set of values, provided one value is known.

Part 3 - Summary

In this part, we have sketched three example algorithms which achieve consensus in 1 round trip and tolerate any minority failure:

- Co-located consensus
- Supermajority consensus
- Binary consensus

Closing Remarks

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

Any questions?

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