FAULT TOLERANCE THROUGH RANDOMNESS

Erik Selin

Background

We will study the design of fault tolerant distributed algorithms made resilient to node failures in asynchronous environments through randomization.

Background

We will study the design of fault tolerant distributed algorithms made resilient to node failures in asynchronous environments through randomization.

Fault Tolerant

Distributed algorithm capable of continuing execution in the event of faults.

Fault examples:

- **Crash**
- **Send/Receive Omission**
- **Byzantine**

Background

We will study the design of fault tolerant distributed algorithms made resilient to node failures in asynchronous environments through randomization.

Node Failures

Simply the possibility that an entity can fail during the computation.

Intuitively, if all nodes fail no computation is possible. However, can we handle some nodes failing?

Background

We will study the design of fault tolerant distributed algorithms made resilient to node failures in asynchronous environments through randomization.

Asynchronous Environments

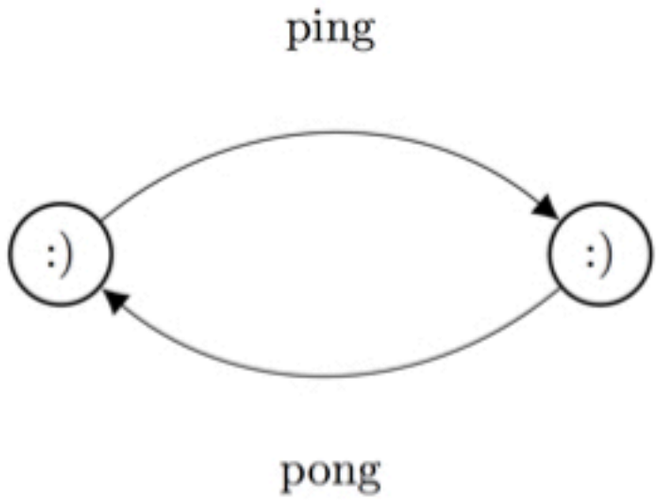
A distributed environment where we cannot distinguish between a link experiencing a very long timeout versus a fault.

Background

We will study the design of fault tolerant distributed algorithms made resilient to node failures in asynchronous environments through randomization.

Asynchronous Environments

A distributed environment where we cannot distinguish between a link experiencing a very long timeout versus a fault.

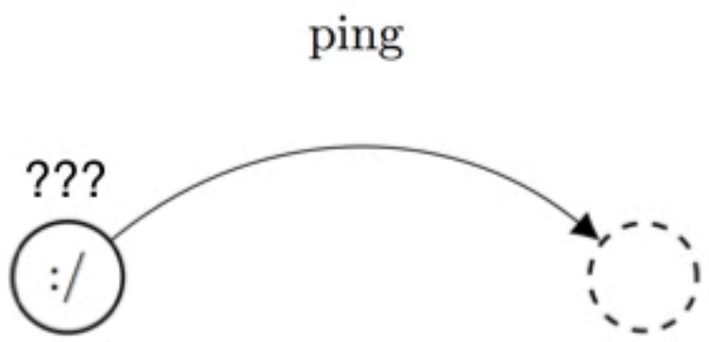


Background

We will study the design of fault tolerant distributed algorithms made resilient to node failures in asynchronous environments through randomization.

Asynchronous Environments

A distributed environment where we cannot distinguish between a link experiencing a very long timeout versus a fault.



Background

We will study the design of fault tolerant distributed algorithms made resilient to node failures in asynchronous environments through randomization.

Randomization

Using a random component to problems associated with deterministic algorithms.

Pitfalls with randomization:

- **Execution time becomes probabilistic**
- **Incorrect results might be possible (Monte Carlo)**
- **Executions might never terminate (Las Vegas)**

The Problem

Common knowledge can only be achieved if and only if we can achieve consensus.

The Problem

Common knowledge can only be achieved if and only if we can achieve consensus.

Binary Consensus

N asynchronous nodes wish to agree about a binary value.

In addition, if for all nodes P, Xp = v then the decision must be v. Ruling out the trivial solution (i.e. always pick 0)

The Problem

Common knowledge can only be achieved if and only if we can achieve consensus.

The Single-Fault Disaster

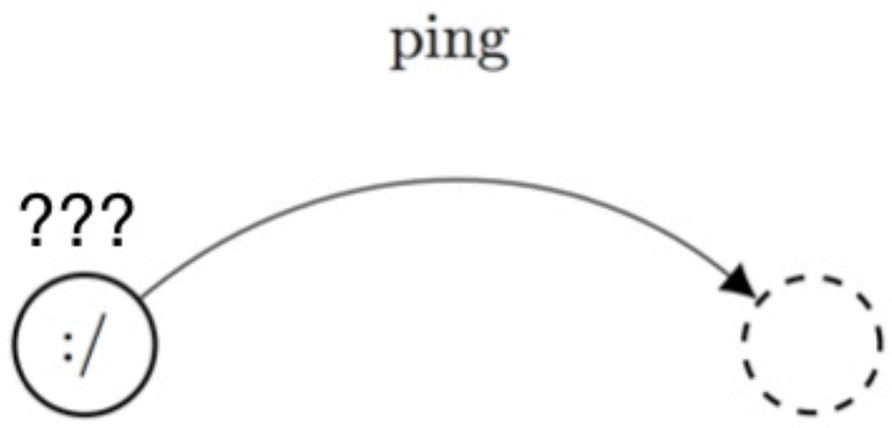
Deterministic algorithms cannot achieve fault-tolerant consensus in asynchronous environments even under the best of conditions.

Often referred to as the Fischer, Lynch, Paterson impossibility proof.

The Problem

Common knowledge can only be achieved if and only if we can achieve consensus.

The Single-Fault Disaster



Solution

A solution:

Introduce non-determinism through randomness.

Additional assumptions:

- **Bidirectional links**
- **Each node has a unique ID**
- **The network is a complete graph (N is known)**
- **Each node has access to a fair coin**
- **Message ordering (ie. FIFO)**

Consensus Protocol

Consensus Protocol

- **Operates in rounds.**
- **Las Vegas Style Algorithm**
  - **Terminates with high-probability**
  - **Upon termination the solution is always correct**
- **Called Rand-Omit in the book, Ben-Or is more common and appropriate after creator Dr. Ben-Or**
- **Proceedings of the second annual ACM symposium on Principles of distributed computing, 1983**
- **Only concerned with crash-type faults**
- **The goal is to establish binary consensus assuming at most t < N/2 nodes crash.**

Consensus Protocol

Code Example

Consensus Protocol

```
x = initial value 0 or 1
r = 1
```

Consensus Protocol

```
x = initial value 0 or 1
r = 1
STEP_1:
  send VOTE(r, x) to all
```

Consensus Protocol

```
x = initial value 0 or 1
r = 1
STEP_1:
  send VOTE(r, x) to all
STEP_2:
  wait for N - t messages of type VOTE(r, v)
  if more than N/2 messages have same v:
    send DECIDE(r, v) to all
  else:
    send DECIDE(r, ?) to all
```

Consensus Protocol

```
x = initial value 0 or 1
r = 1
STEP_1:
  send VOTE(r, x) to all
STEP_2:
  wait for N - t messages of type VOTE(r, v)
  if more than N/2 messages have same v:
    send DECIDE(r, v) to all
  else:
    send DECIDE(r, ?) to all
STEP_3:
  wait for N - t messages of type DECIDE(r, *)
  if one or more DECIDE(r, v) message:
    x = v
    if t+1 or more DECIDE(r, v) messages with same v:
      decide(v)
  else:
    x = 0 or 1 with p = 0.5
```

Consensus Protocol

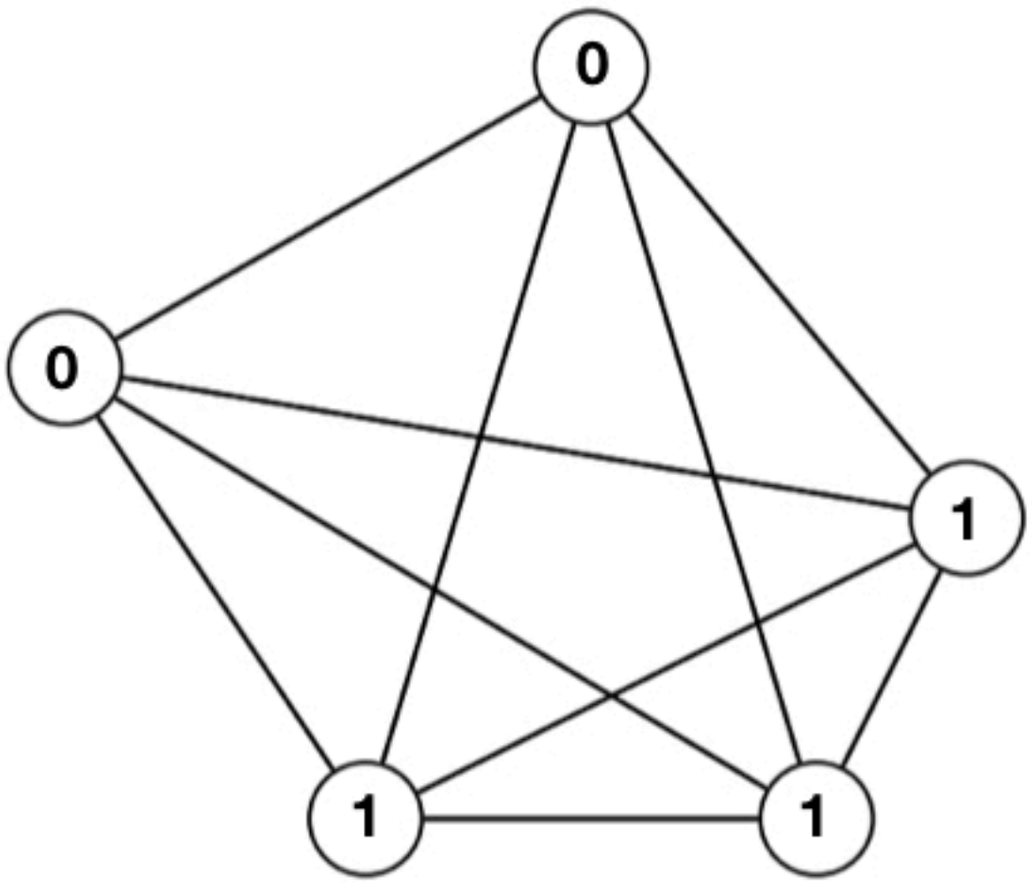
```
x = initial value 0 or 1
r = 1
STEP_1:
  send VOTE(r, x) to all
STEP_2:
  wait for N - t messages of type VOTE(r, v)
  if more than N/2 messages have same v:
    send DECIDE(r, v) to all
  else:
    send DECIDE(r, ?) to all
STEP_3:
  wait for N - t messages of type DECIDE(r, *)
  if one or more DECIDE(r, v) message:
    x = v
    if t+1 or more DECIDE(r, v) messages with same v:
      decide(v)
  else:
    x = 0 or 1 with p = 0.5
STEP_4:
  r = r + 1
  GOTO STEP_1
```

Consensus Protocol

Execution Example

Consensus Protocol

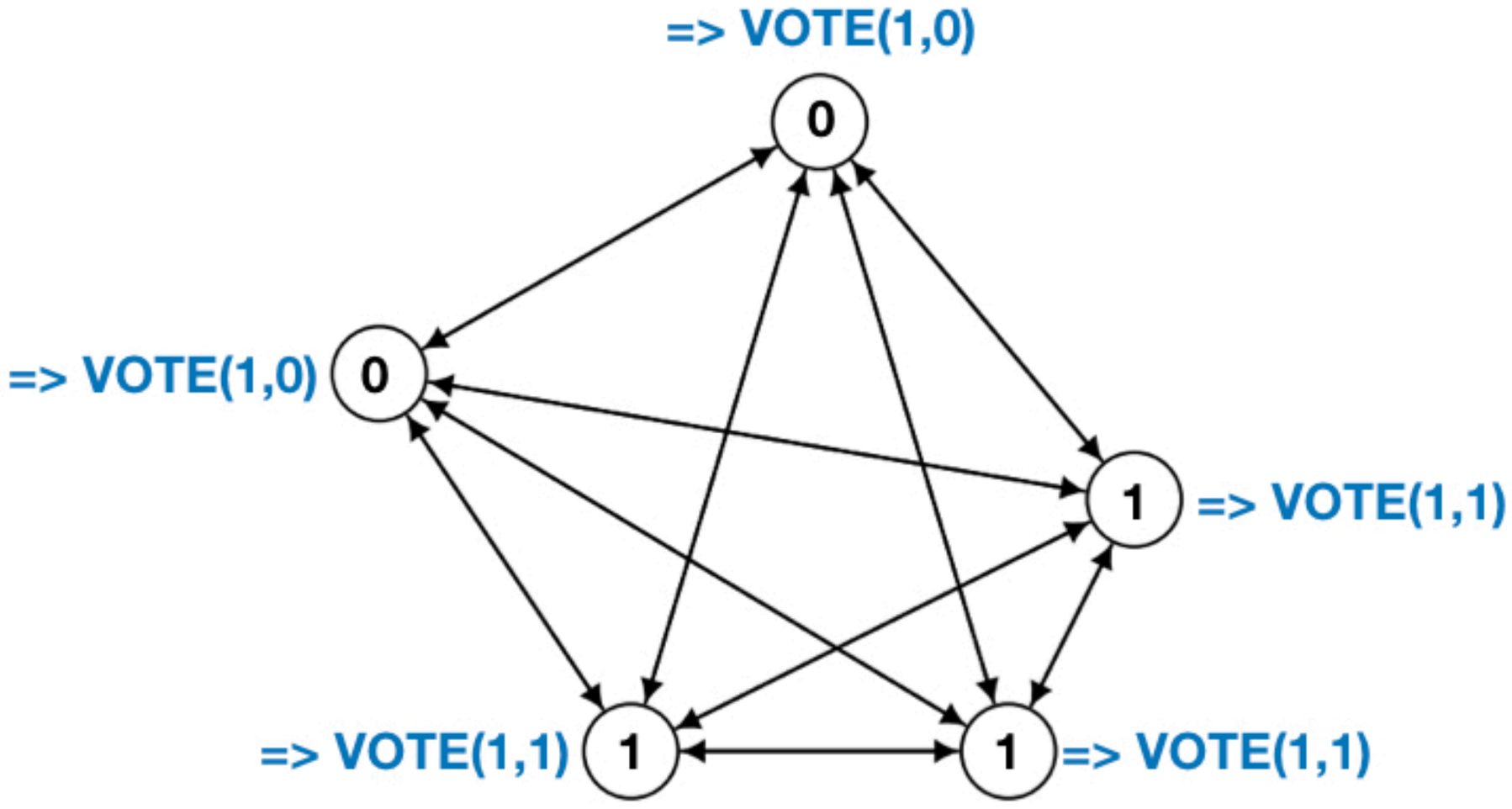
Example for t=2, that is, tolerate up to two crashed nodes



r=1

Consensus Protocol

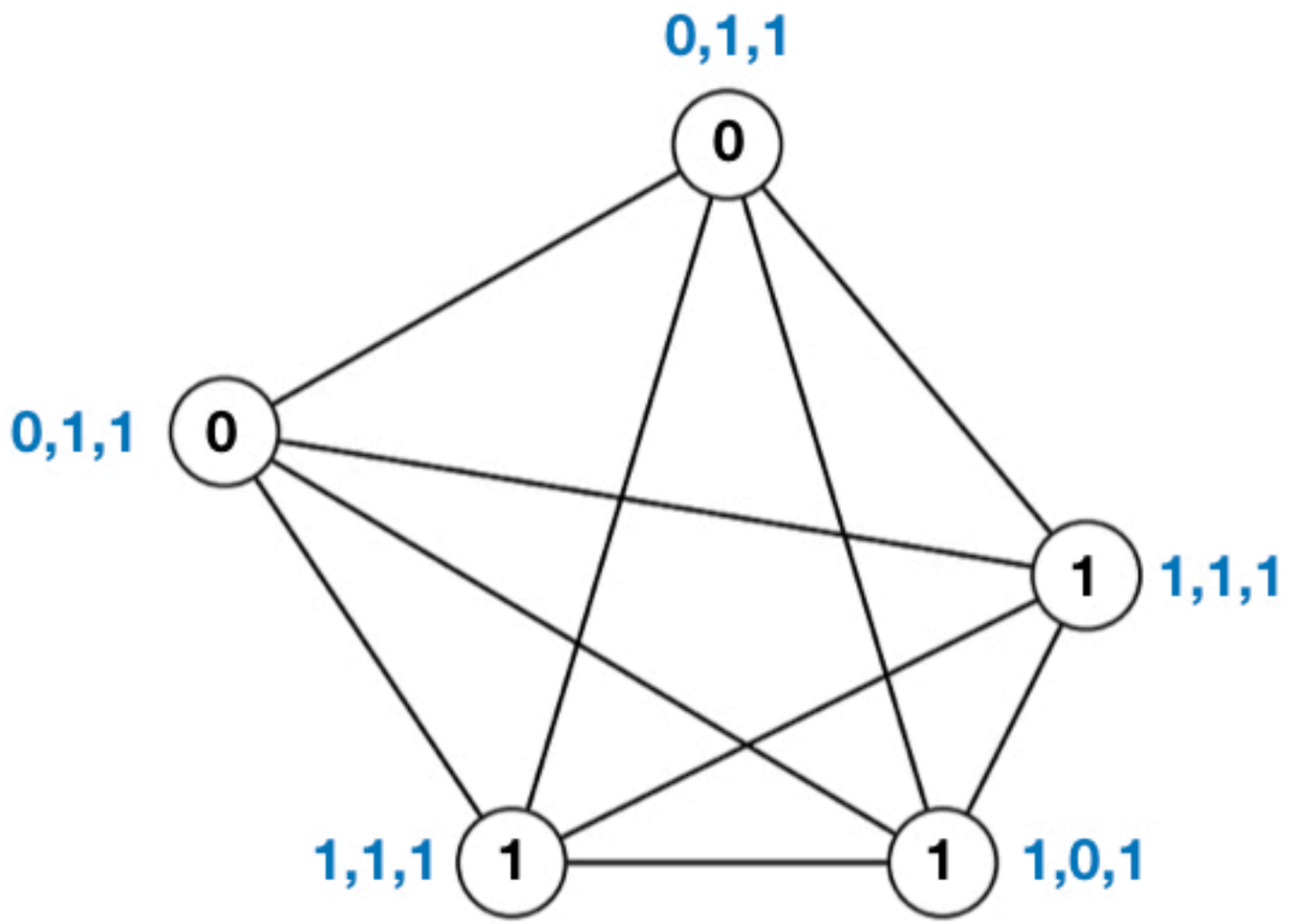
Step 1: send VOTE to all nodes



r=1

Consensus Protocol

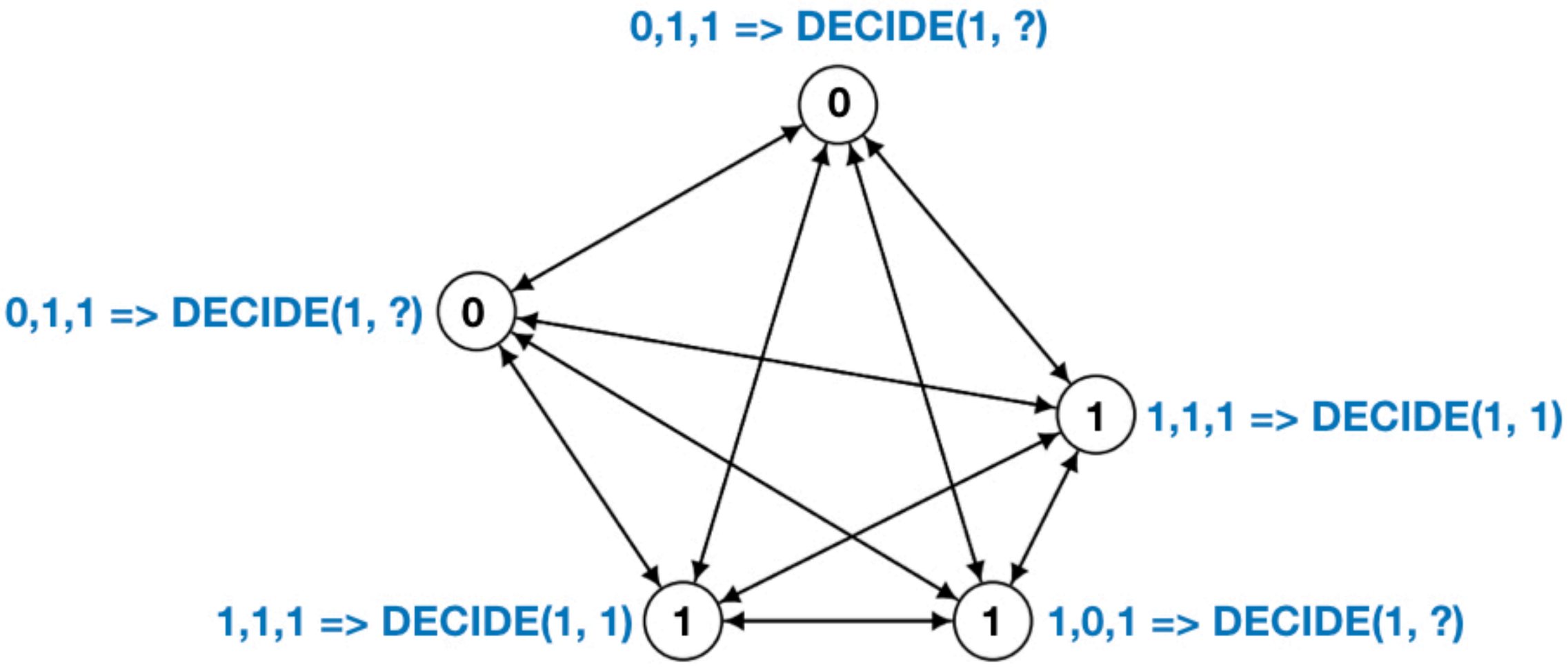
Step 2: wait for N-t VOTE



r=1

Consensus Protocol

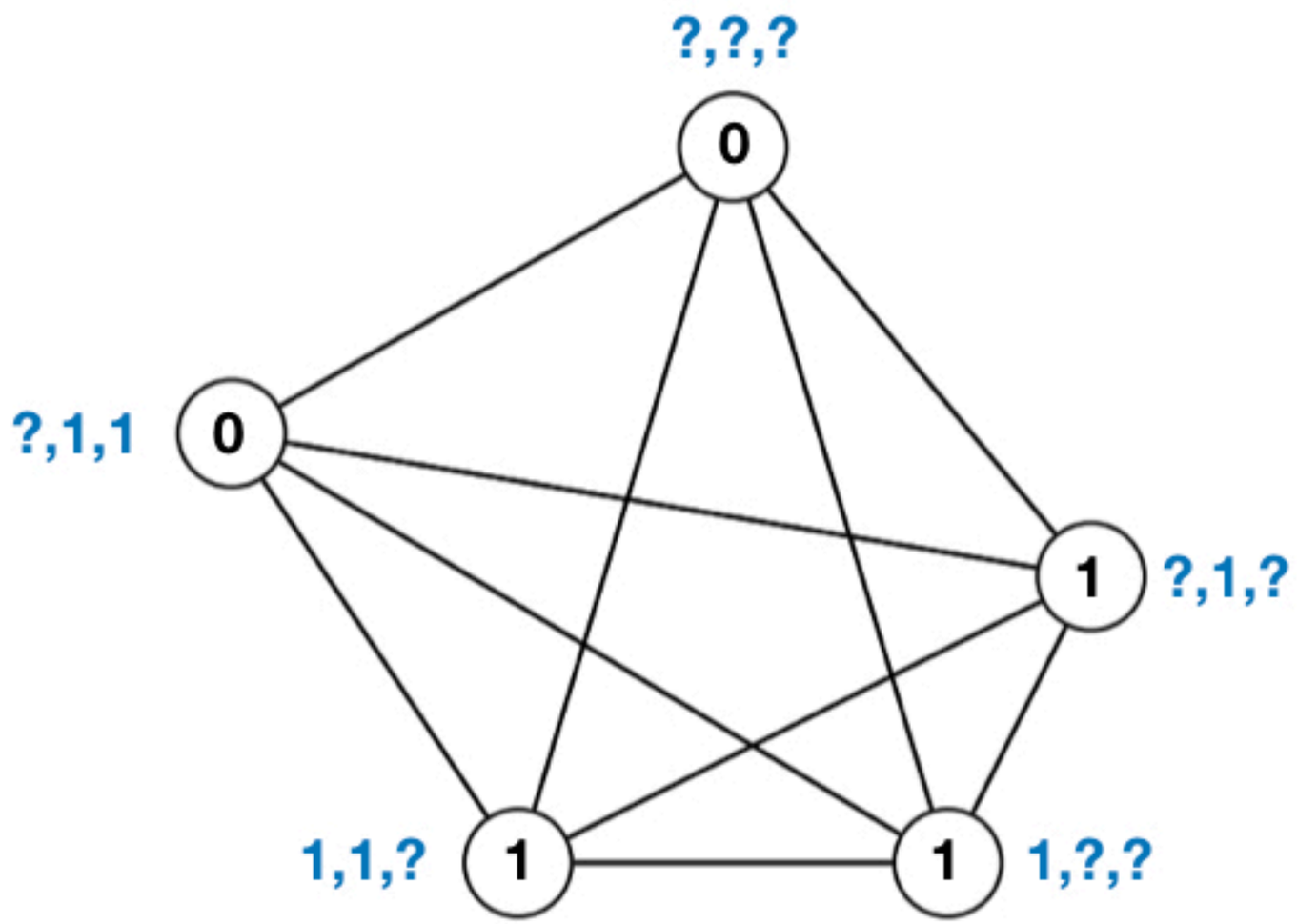
Step 2: send DECIDE based on vote messages



r=1

Consensus Protocol

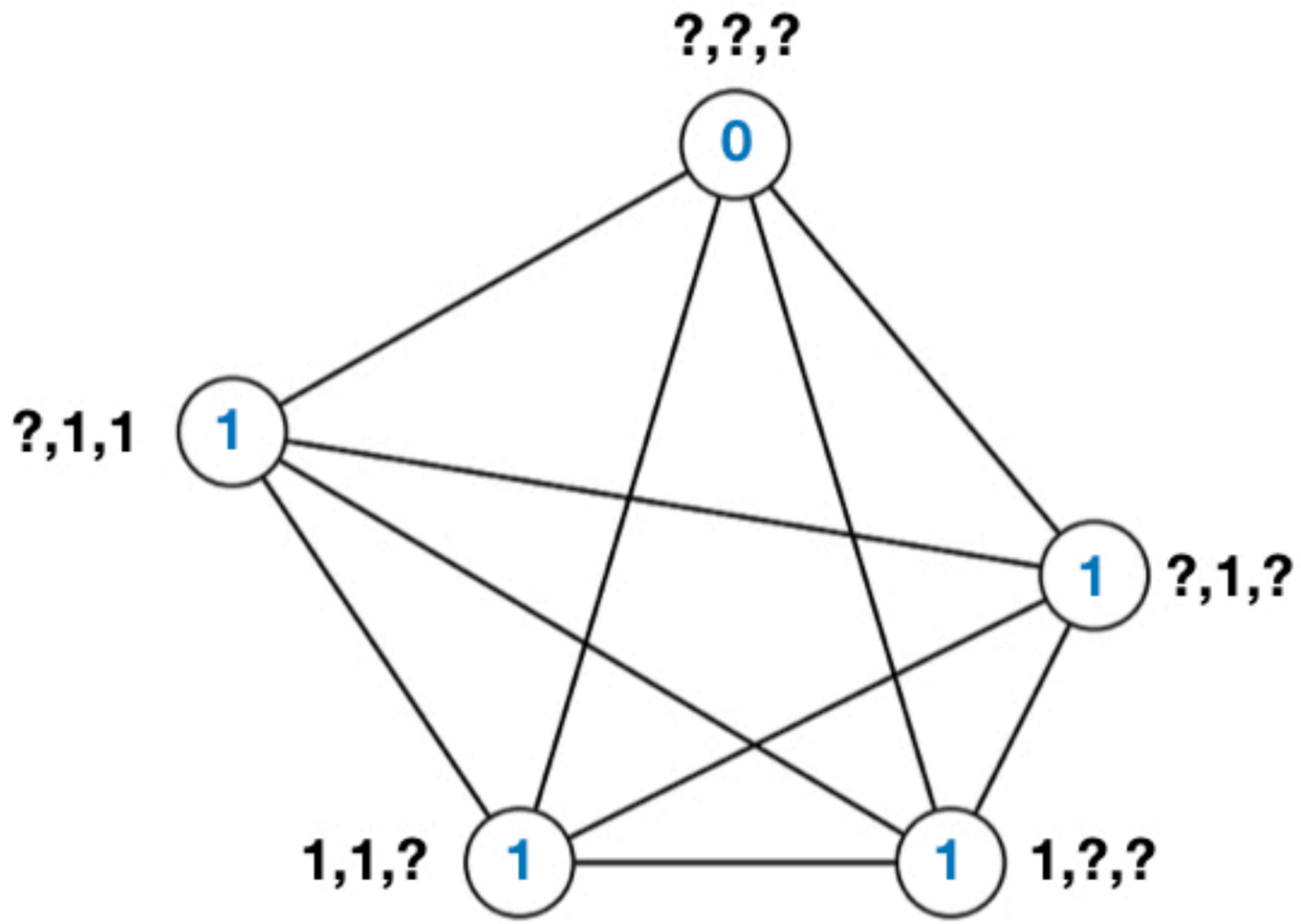
Step 3: wait for N-t DECIDE



r=1

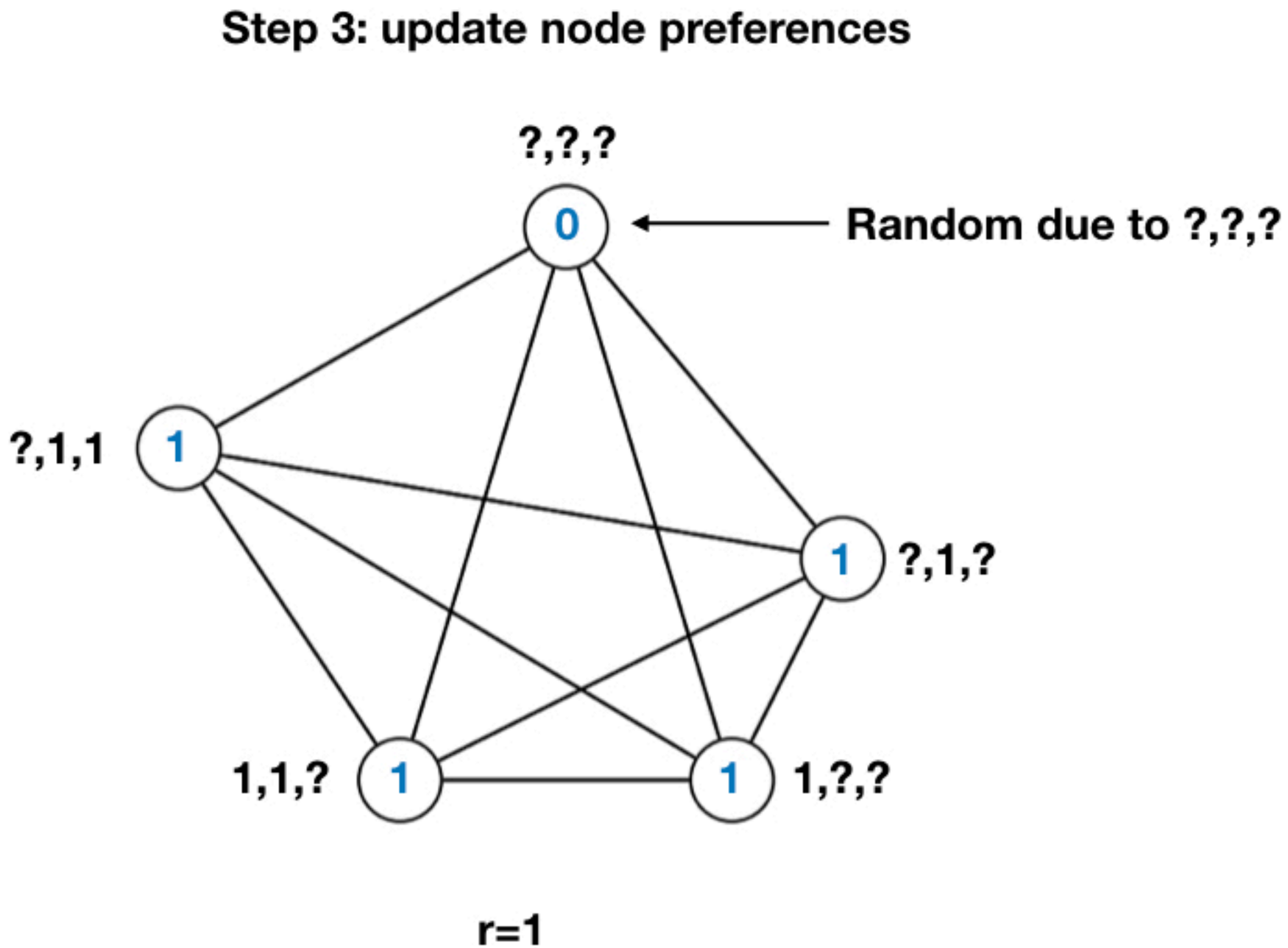
Consensus Protocol

Step 3: update node preferences

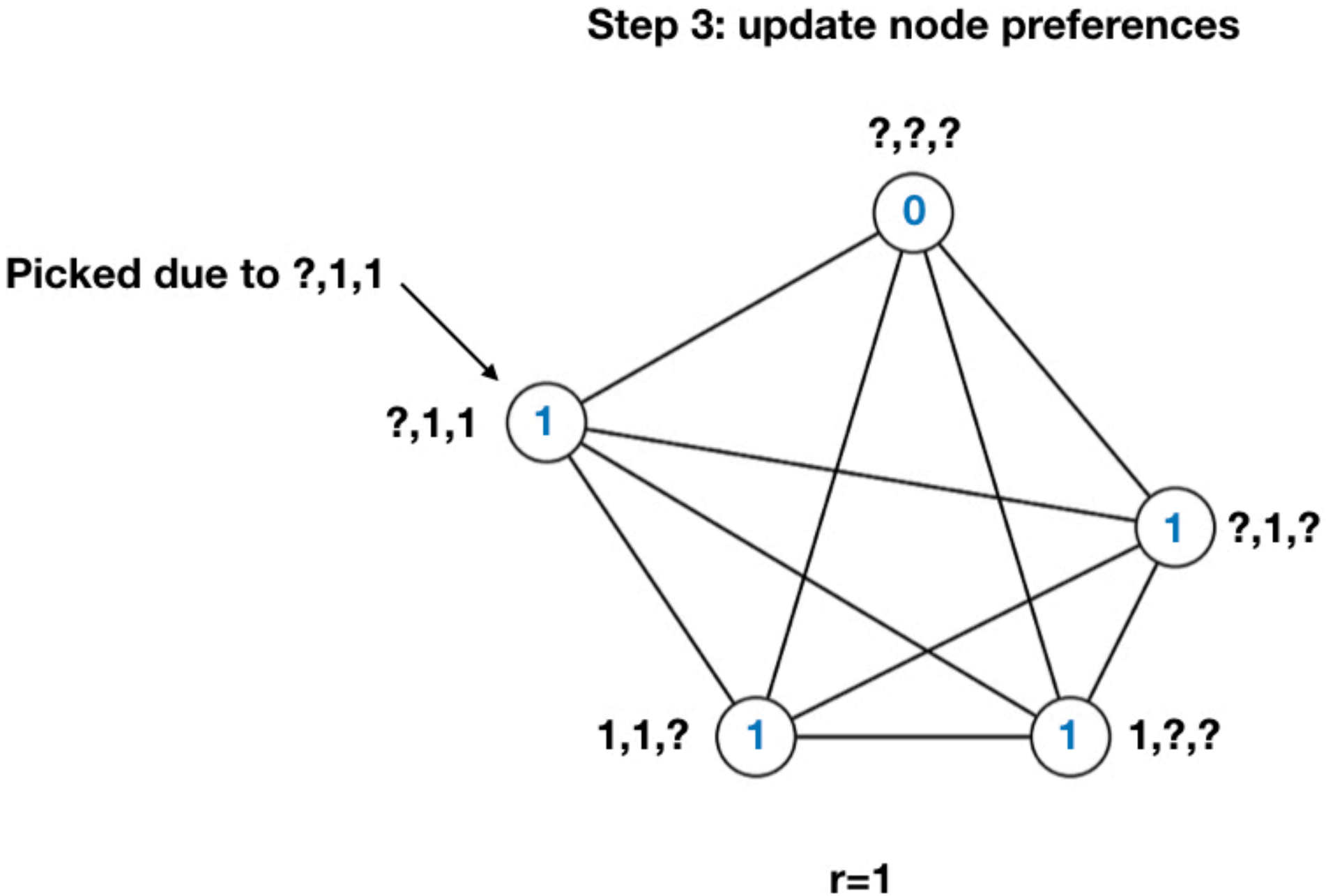


r=1

Consensus Protocol

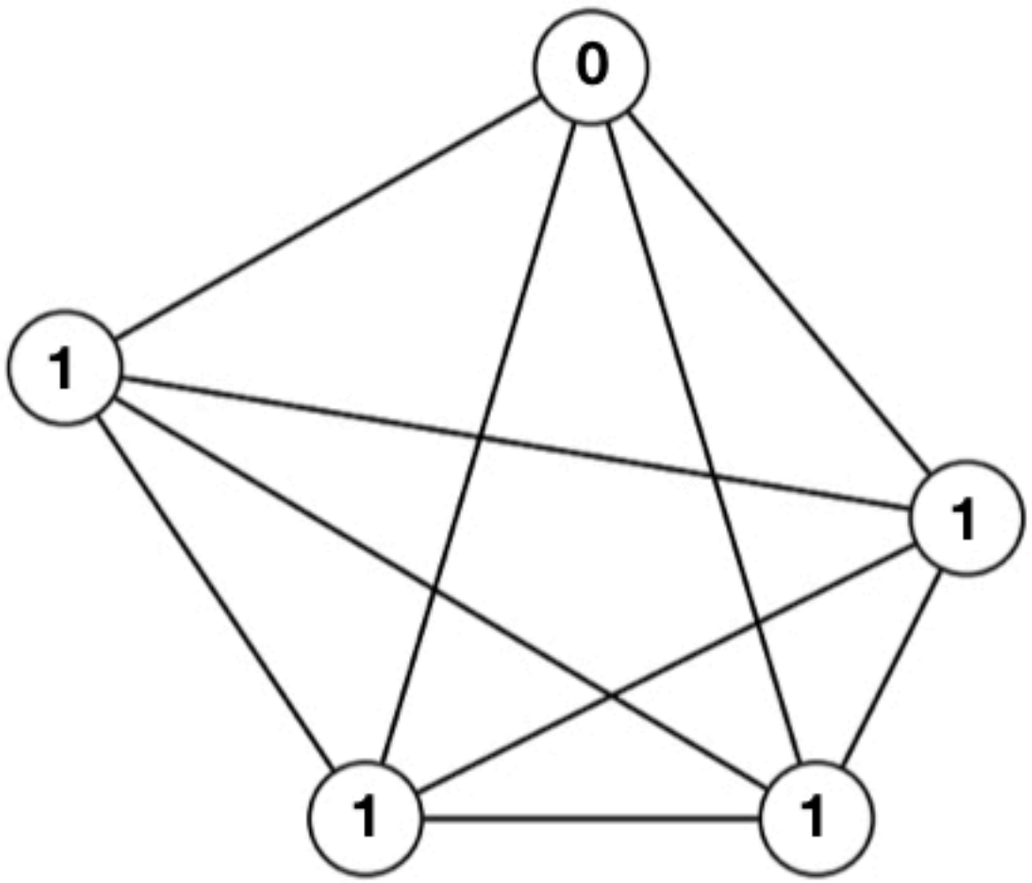


Consensus Protocol



Consensus Protocol

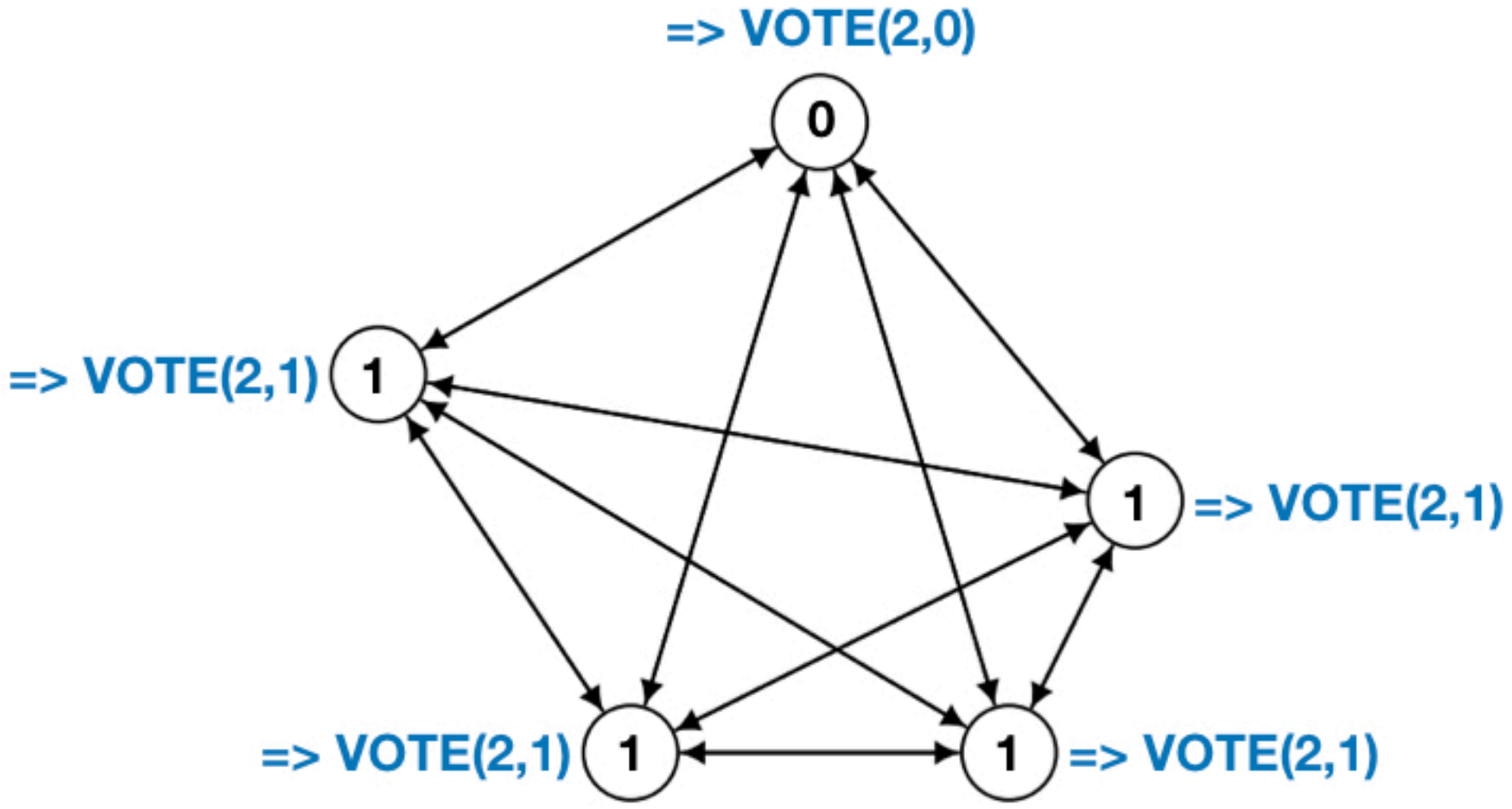
Step 4: next round and go to step 1



r=2

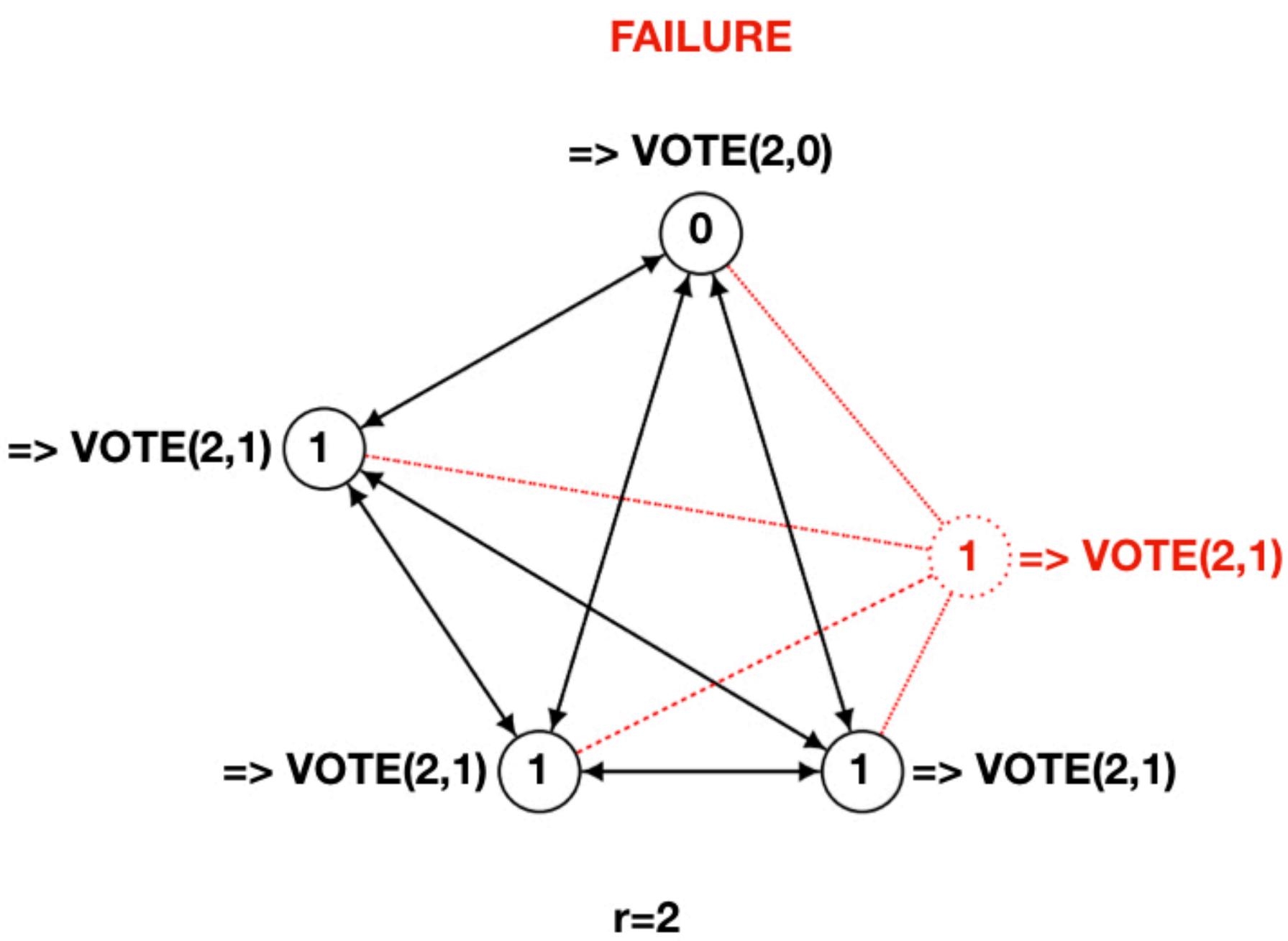
Consensus Protocol

Step 1: send VOTE to all nodes



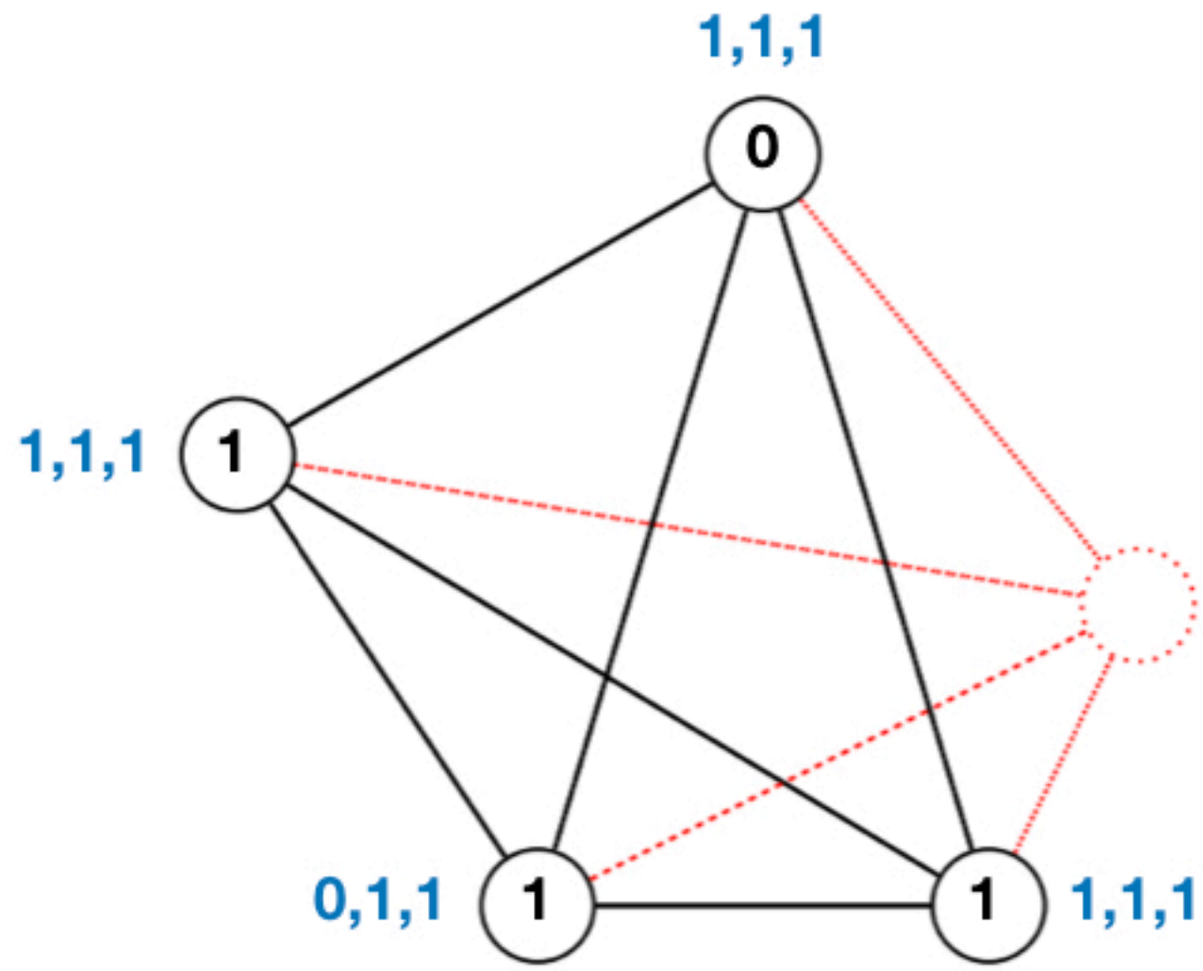
r=2

Consensus Protocol



Consensus Protocol

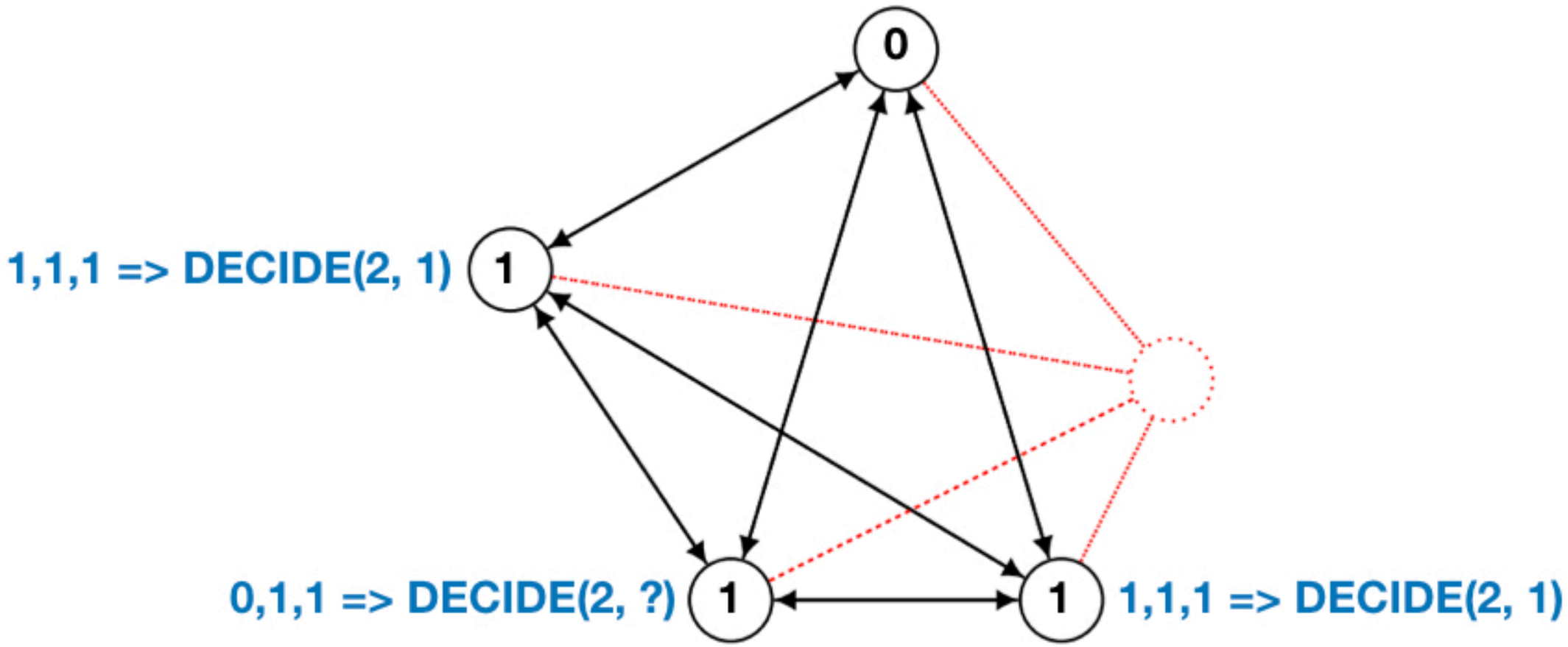
Step 2: wait for N-t VOTE



r=2

Consensus Protocol

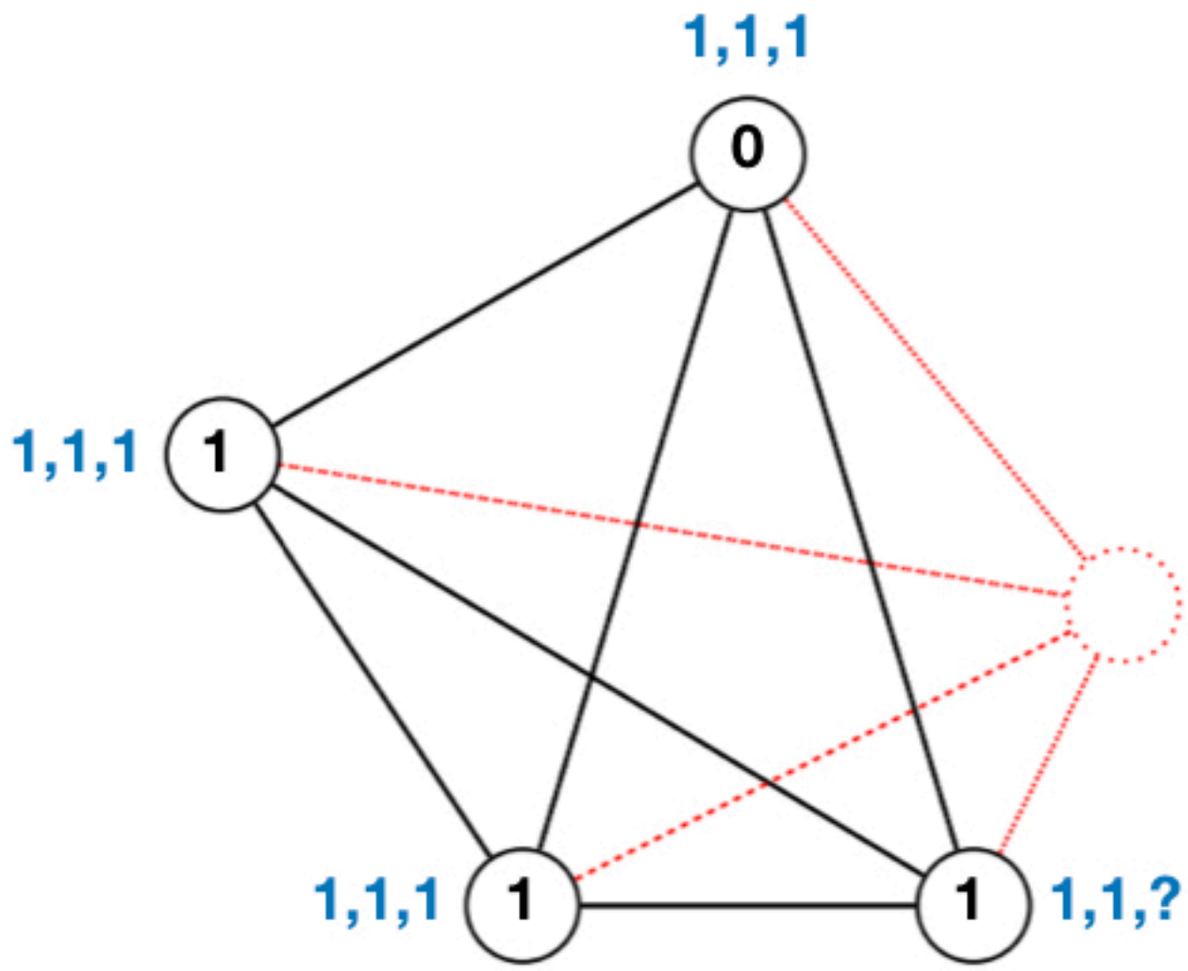
Step 2: send DECIDE based on vote messages



r=2

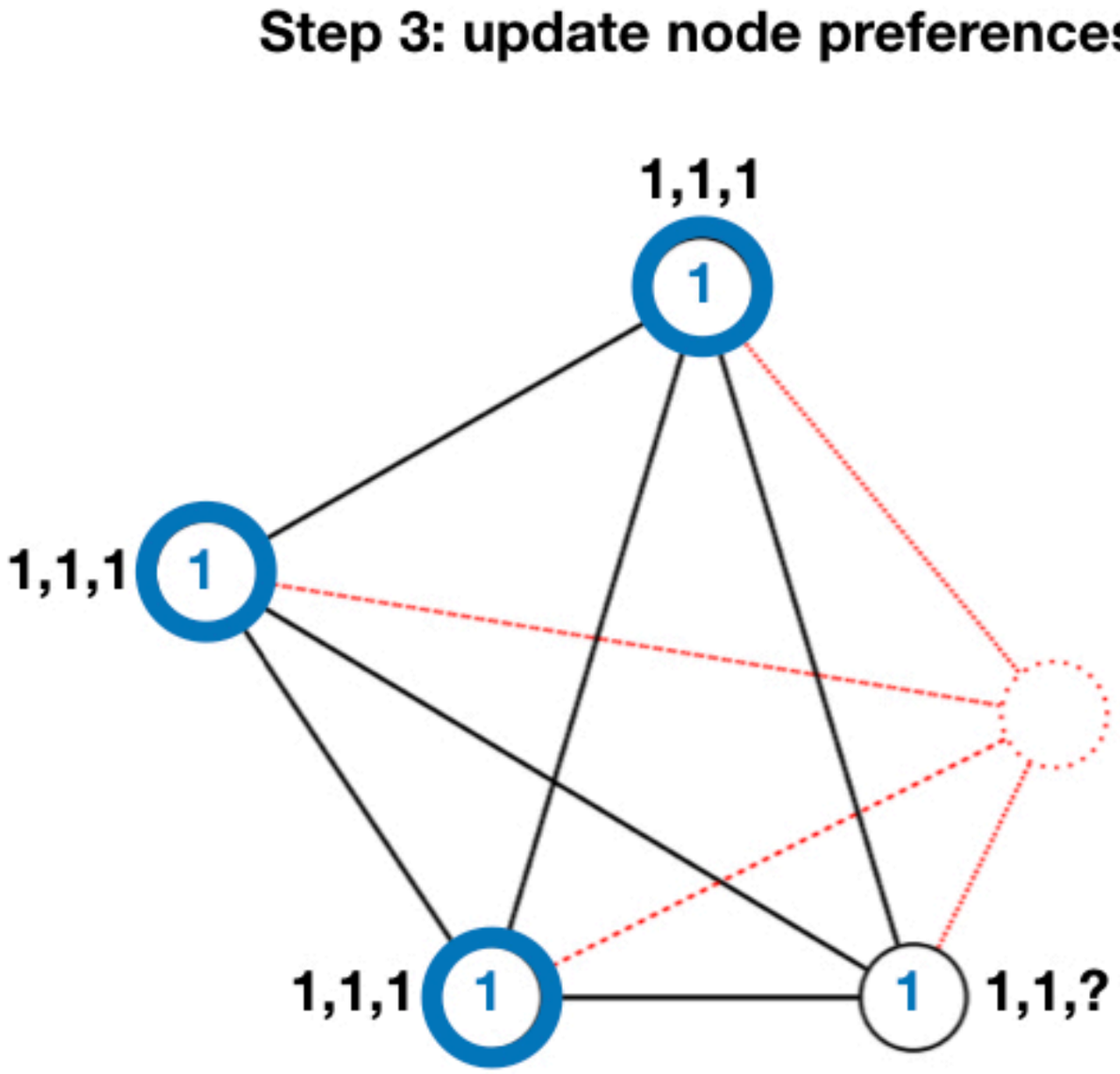
Consensus Protocol

Step 3: wait for N-t DECIDE

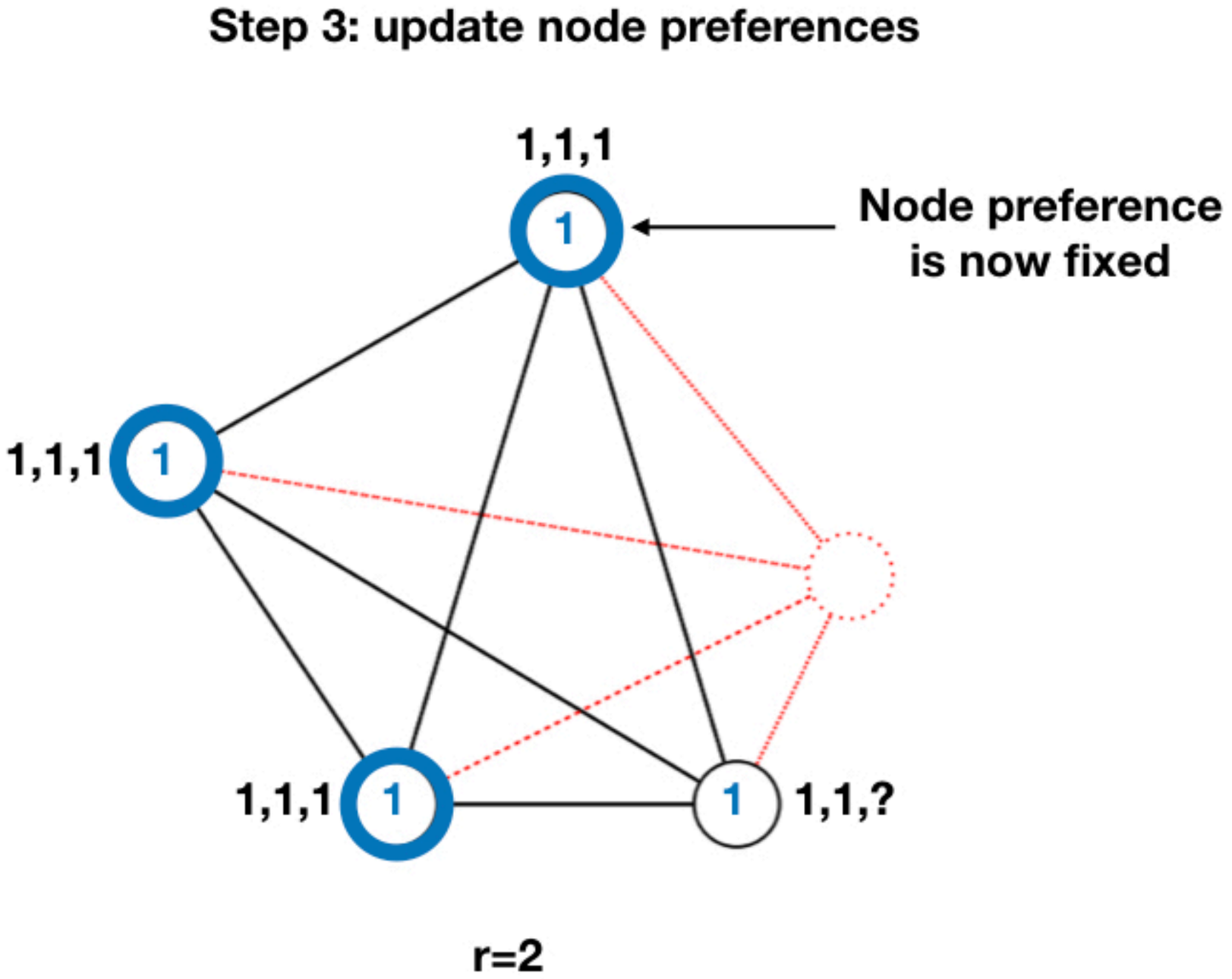


r=2

Consensus Protocol

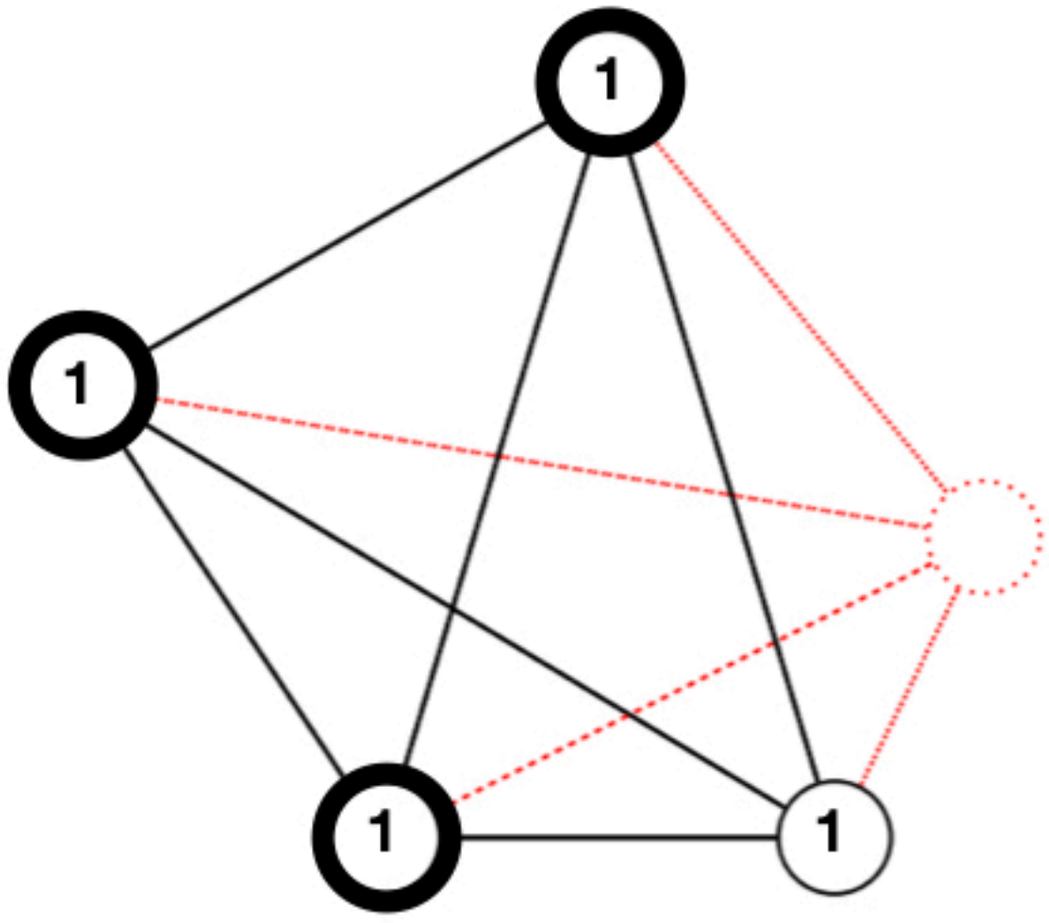


Consensus Protocol



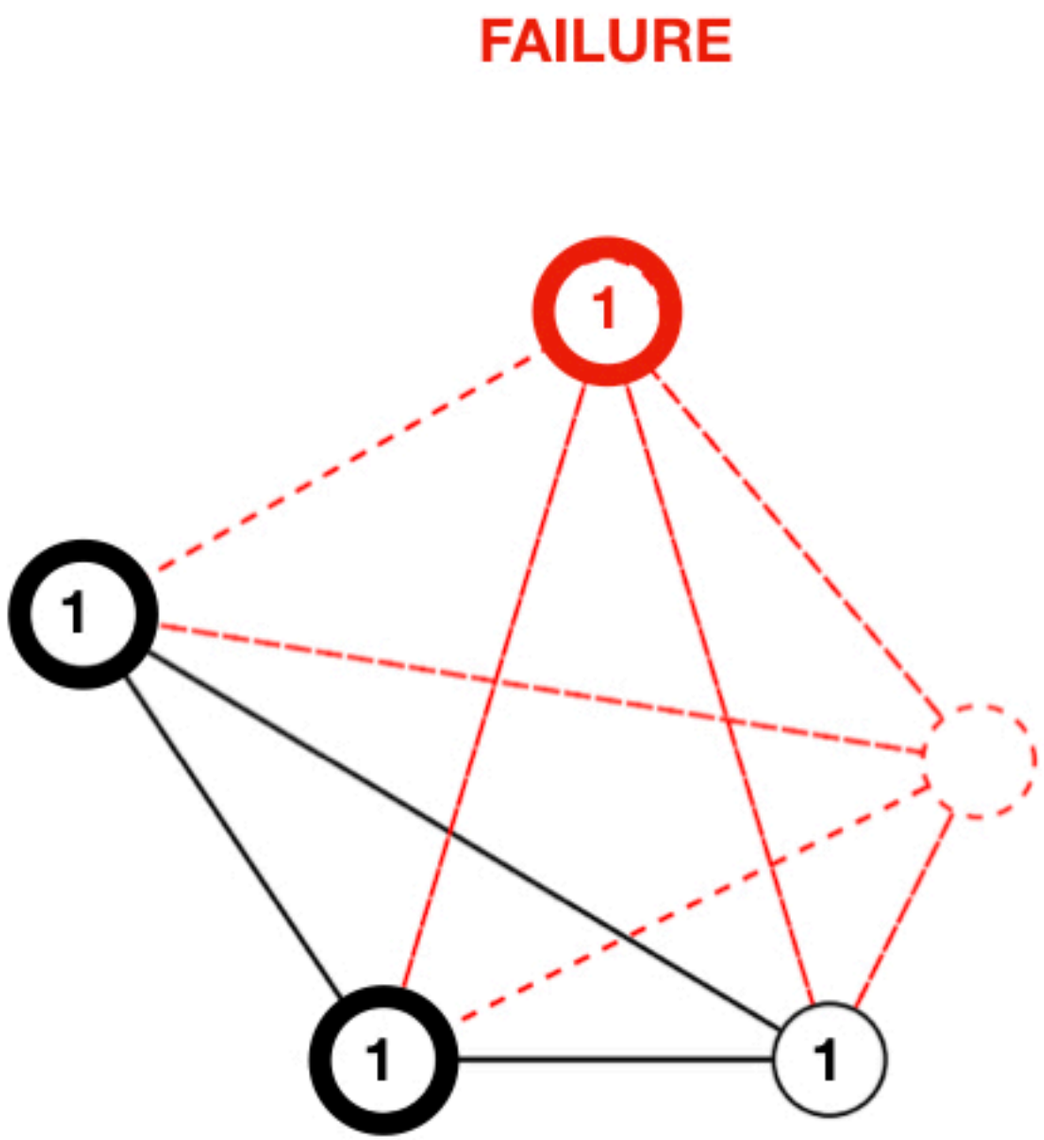
Consensus Protocol

Step 4: next round and go to step 1



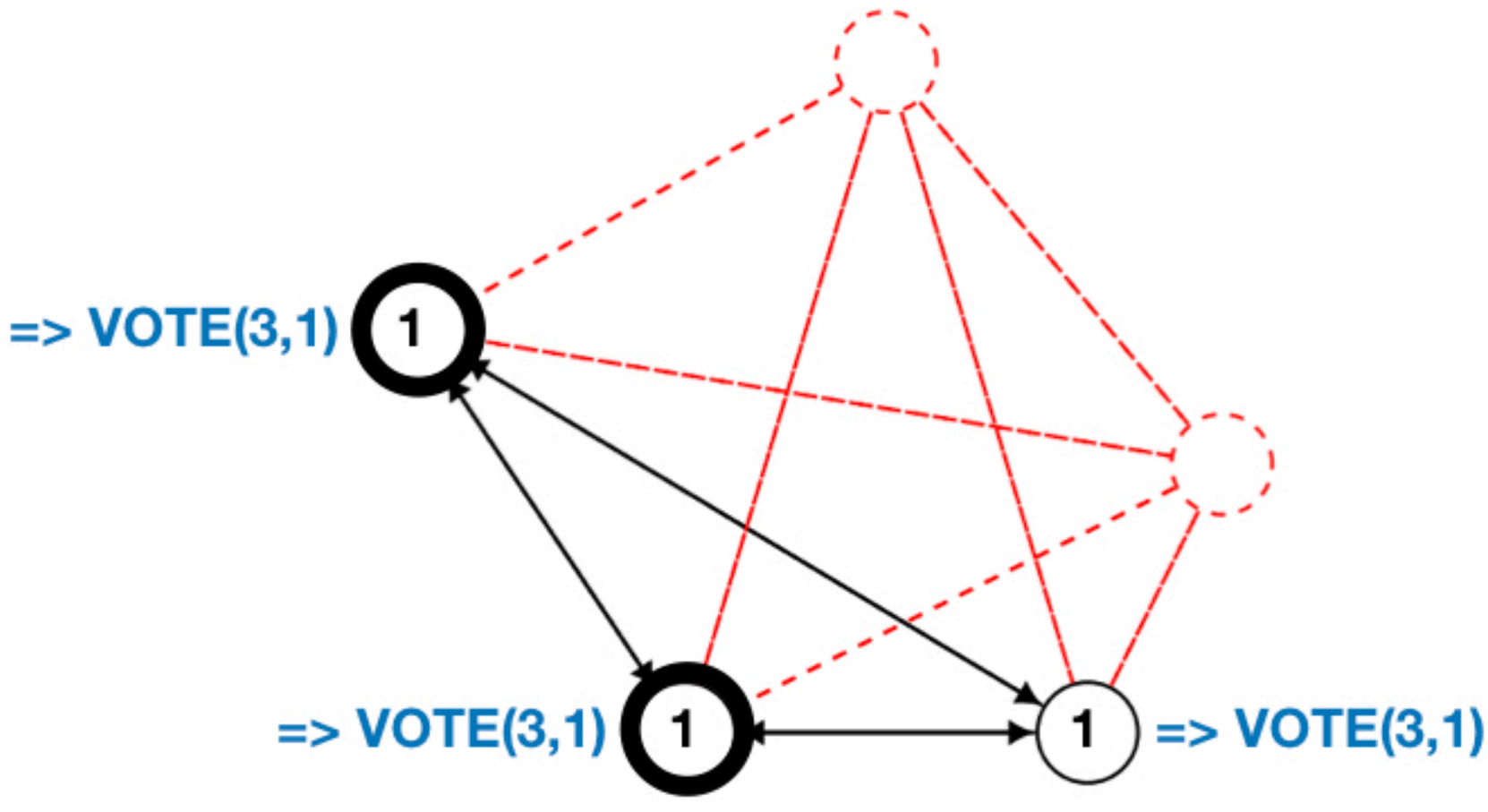
r=3

Consensus Protocol



Consensus Protocol

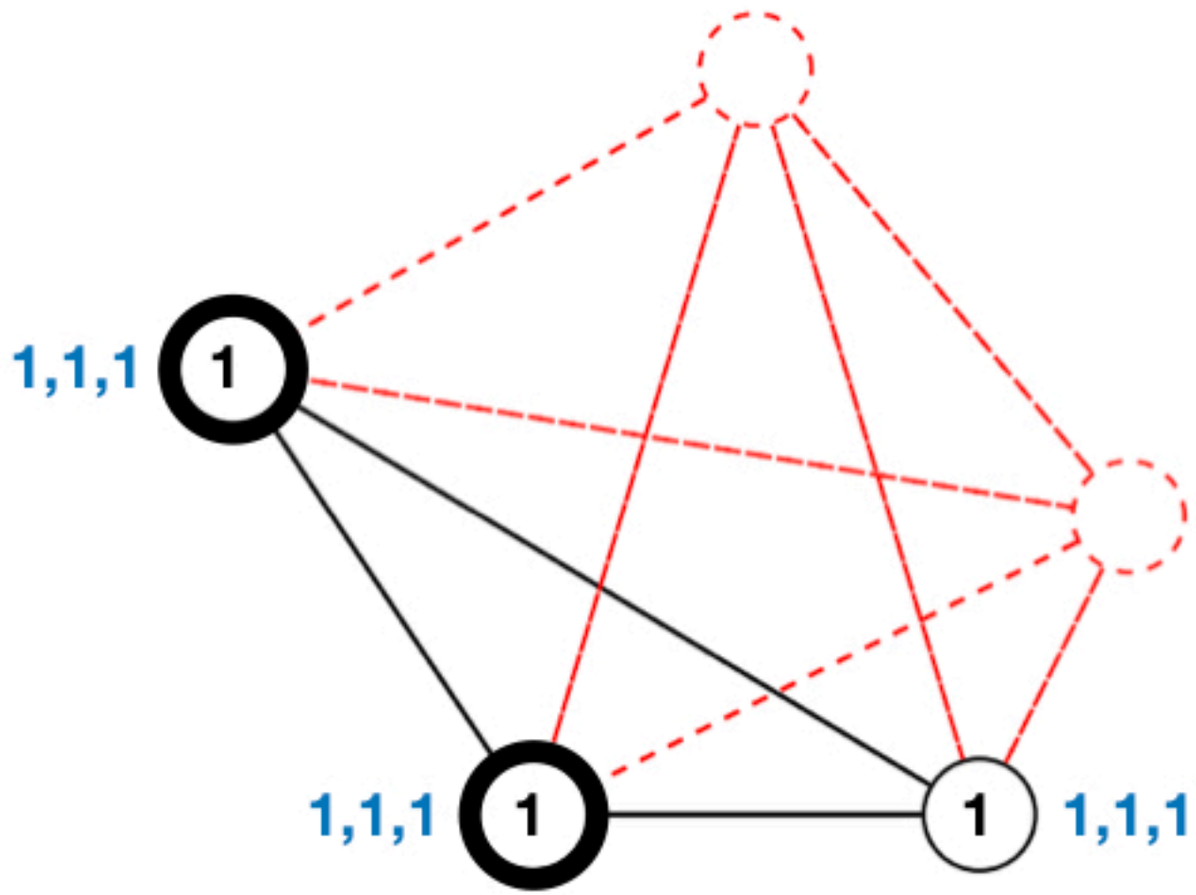
Step 1: send VOTE to all nodes



r=3

Consensus Protocol

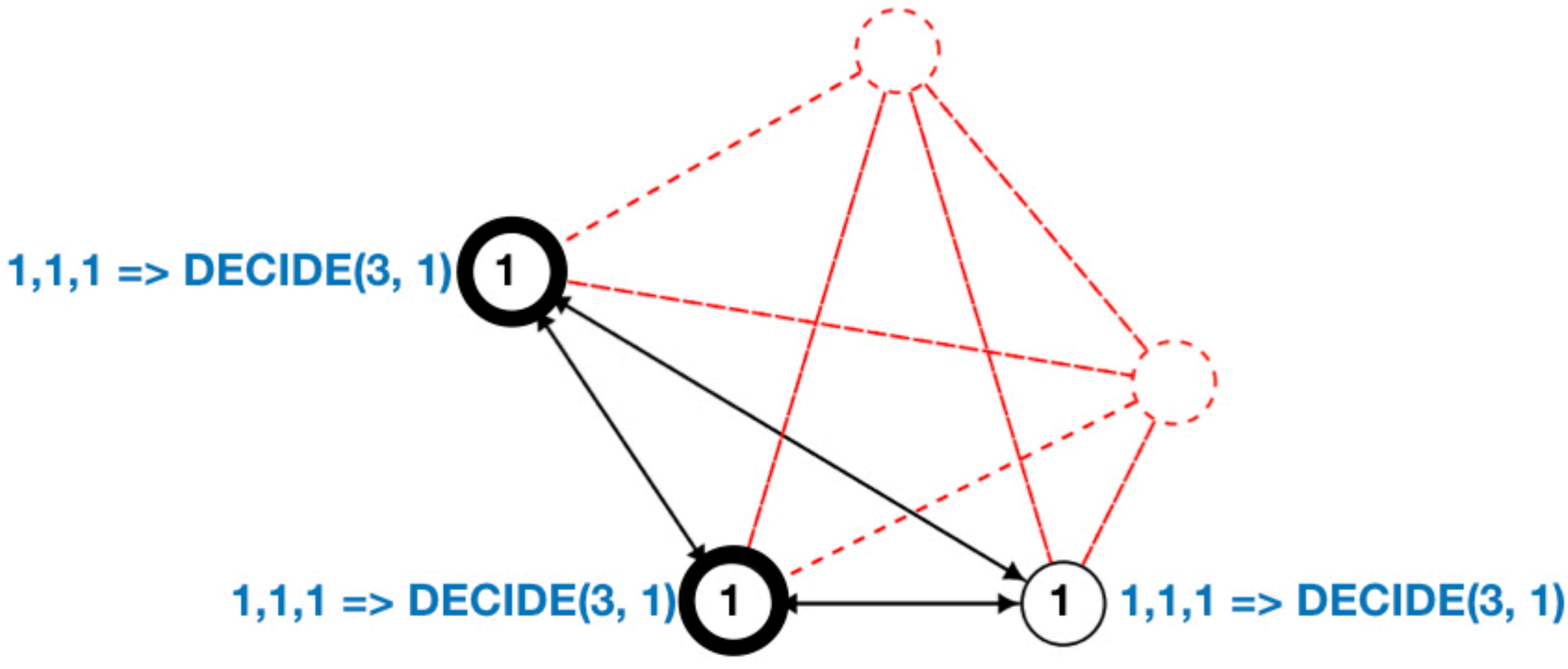
Step 2: wait for N-t VOTE



r=3

Consensus Protocol

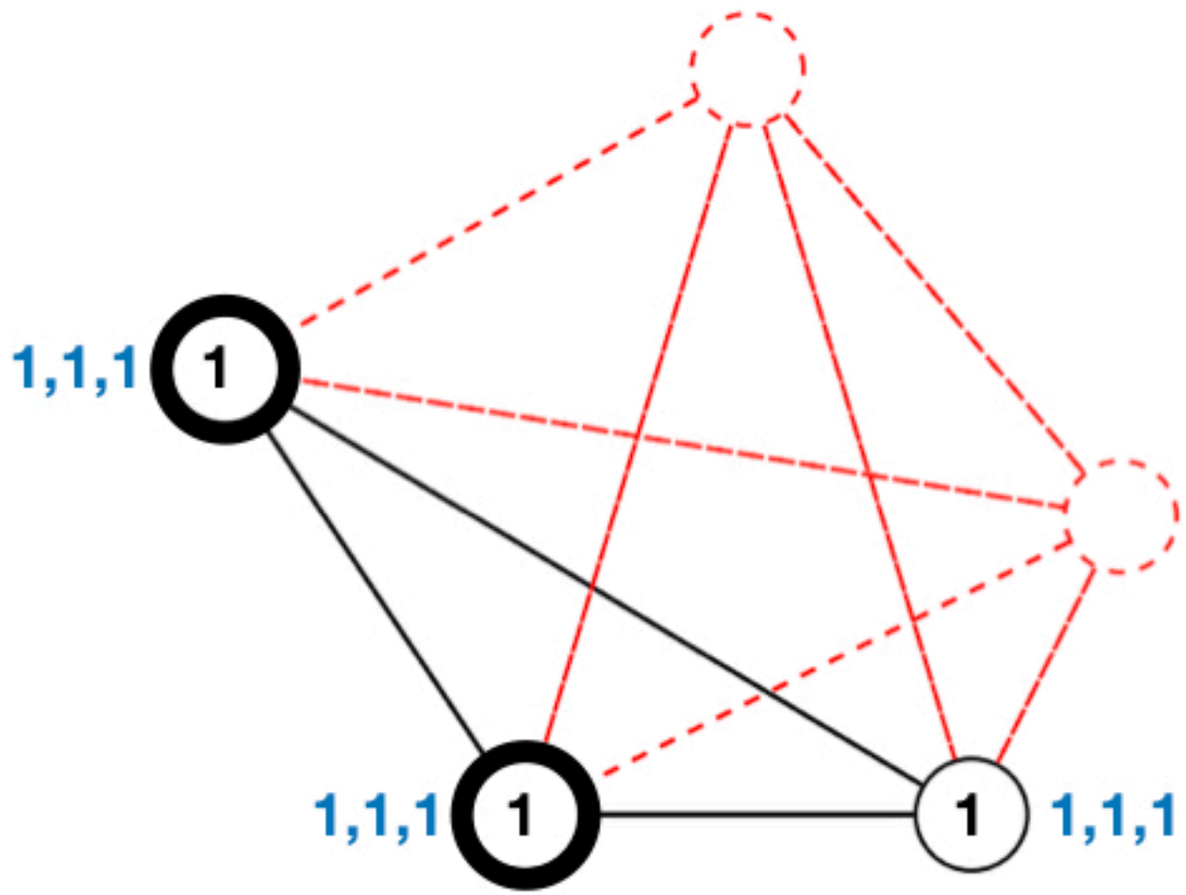
Step 2: send DECIDE based on vote messages



r=3

Consensus Protocol

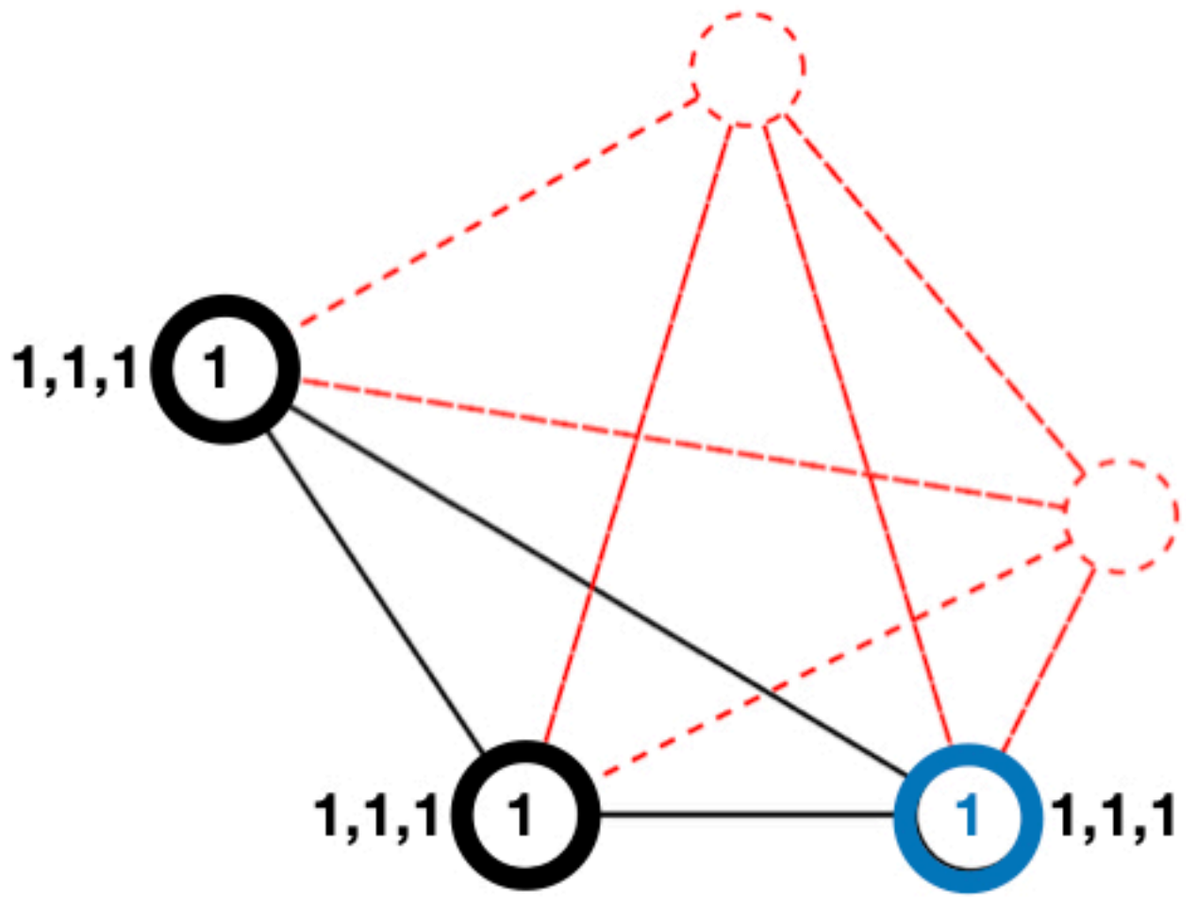
Step 3: wait for N-t DECIDE



r=3

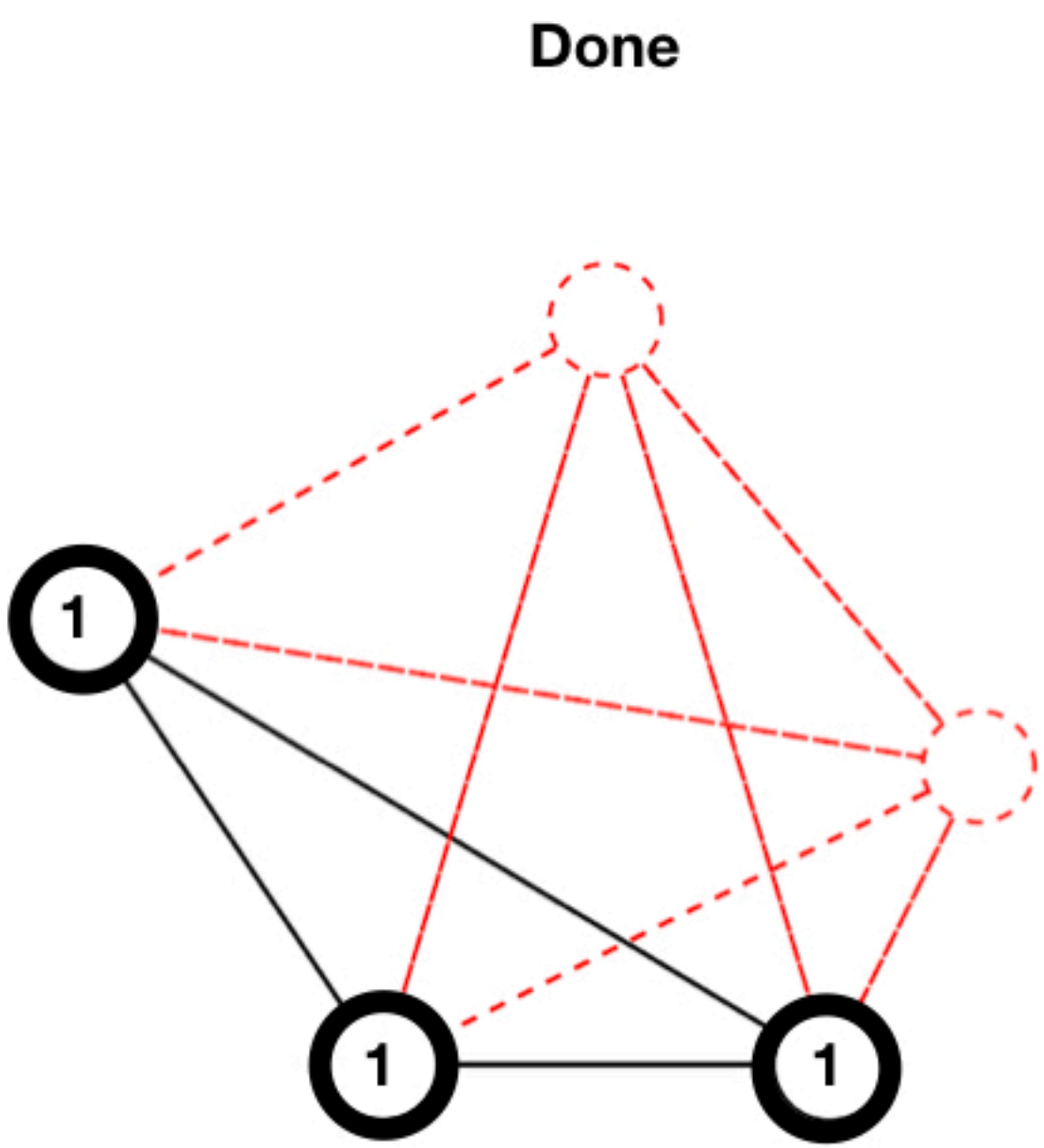
Consensus Protocol

Step 3: update node preferences



r=3

Consensus Protocol



Complexity

It is complicated

Very bad in worse case, the Ben-Or algorithm can be exponential in the number of rounds. This means even worse than exponential in terms of message complexity.

However, if $t < O(\sqrt{N})$, then the number of rounds is O(1). Thus, message complexity becomes quadratic.

If for all P, Xp = v, v will be established in the first round. Which also results in quadratic message complexity.

Termination

Two definitions

In most of the literature termination for random algorithms occurs when all processes are done deciding on the same value.

In the Ben-Or algorithm termination could be defined to occur when all nodes stop participating in the algorithm, ie, when they actually stop (not shows in this example, requires some more code)

Byzantine Protocol

Byzantine Protocol

- **Las Vegas Style Algorithm**
  - **Terminates with high-probability**
  - **Upon termination the solution is always correct**
- **Extension by Ben-Or from original publication (not in book) handling Byzantine faults.**
- **Intuitive and simple extension of regular Ben-Or.**
- **The goal is to establish binary consensus assuming at most t < N/5 nodes crash OR suffer from Byzantine faults.**
- **Additional assumption: must be capable of determining message origin.**

Byzantine Protocol

Code Example

Byzantine Protocol

```
x = initial value 0 or 1
r = 1
STEP_1:
  send VOTE(r, x) to all
STEP_2:
  wait for N - t messages of type VOTE(r, v)
  if more than N/2 messages have same v:
    send DECIDE(r, v) to all
  else:
    send DECIDE(r, ?) to all
STEP_3:
  wait for N - t messages of type DECIDE(r, *)
  if one or more DECIDE(r, v) message with same v:
    x = v
    if t+1 or more DECIDE(r, v) messages with same v:
      decide(v)
  else:
    x = 0 or 1 with p = 0.5
STEP_4:
  r = r + 1
  GOTO STEP_1
```

Byzantine Protocol

```
x = initial value 0 or 1
r = 1
STEP_1:
  send VOTE(r, x) to all
STEP_2:
  wait for N - t messages of type VOTE(r, v)
  if more than N/2 => (N+t)/2 messages have same v:
    send DECIDE(r, v) to all
  else:
    send DECIDE(r, ?) to all
STEP_3:
  wait for N - t messages of type DECIDE(r, *)
  if one or more DECIDE(r, v) message with same v:
    x = v
    if t+1 or more DECIDE(r, v) messages with same v:
      decide(v)
  else:
    x = 0 or 1 with p = 0.5
STEP_4:
  r = r + 1
  GOTO STEP_1
```

Byzantine Protocol

```
x = initial value 0 or 1
r = 1
STEP_1:
  send VOTE(r, x) to all
STEP_2:
  wait for N - t messages of type VOTE(r, v)
  if more than N/2 => (N+t)/2 messages have same v:
    send DECIDE(r, v) to all
  else:
    send DECIDE(r, ?) to all
STEP_3:
  wait for N - t messages of type DECIDE(r, *)
  if one => t+1 or more DECIDE(r, v) message with same v:
    x = v
    if t+1 or more DECIDE(r, v) messages with same v:
      decide(v)
  else:
    x = 0 or 1 with p = 0.5
STEP_4:
  r = r + 1
  GOTO STEP_1
```

Byzantine Protocol

```
x = initial value 0 or 1
r = 1
STEP_1:
  send VOTE(r, x) to all
STEP_2:
  wait for N - t messages of type VOTE(r, v)
  if more than N/2 => (N+t)/2 messages have same v:
    send DECIDE(r, v) to all
  else:
    send DECIDE(r, ?) to all
STEP_3:
  wait for N - t messages of type DECIDE(r, *)
  if one => t+1 or more DECIDE(r, v) message:
    x = v
    if t+1 => (N+t)/2 or more DECIDE(r, v) messages with same v:
      decide(v)
  else:
    x = 0 or 1 with p = 0.5
STEP_4:
  r = r + 1
  GOTO STEP_1
```

References

- Michael Ben-Or. 1983. Another advantage of free choice (Extended Abstract): Completely asynchronous agreement protocols. In Proceedings of the second annual ACM symposium on Principles of distributed computing (PODC '83). ACM, New York, NY, USA, 27-30. DOI=http://dx.doi.org/10.1145/800221.806707

- Aspnes, J. Distrib. Comput. (2003) 16: 165. https://doi.org/10.1007/s00446-002-0081-5

- Nicola Santoro. 2006. Design and Analysis of Distributed Algorithms (Wiley Series on Parallel and Distributed Computing). Wiley-Interscience, New York, NY, USA

FAULT TOLERANCE THROUGH RANDOMNESS

Erik Selin