

Distributed Computing From First Principles

Kenneth Emeka Odoh

<https://kenluck2001.github.io>

Table of Contents

- Theoretical foundations (two-general problem, impossibility proof).
- Primer on Parallel programming (exclude networking programming)
- OpenMPI tutorial
- Logical clocks
- Paxos (Single Value Paxos, Sequence Paxos)
- Failure detector
- Leadership election
- Raft
- Anti-entropy (CRDT, Merkle tree)
- Case studies
- Exercises

Recommended reading: https://kenluck2001.github.io/blog_post/distributed_computing_from_first_principles.html

Diversity & inclusion Initiative:

https://kenluck2001.github.io/blog_post/authoring_a_new_book_on_distributed_computing.html

Theoretical Foundations

- Distributed systems is a series of independent nodes connected by a network and appears as a coherent system.
 - Common examples: Internet, Edge computing, and cloud computing.
 - Challenging due to network failures, node failures.
- CAP theorem (AP, CP system).
- Most algorithms for distributed system consist of a derivative of Paxos, Raft tuned to meet specific application needs.
- **Applications** include distributed caching, distributed key-value hash e.t.c

Theoretical foundations for Distributed systems. They include:

- FLP Impossibility of Consensus
- Two general problem

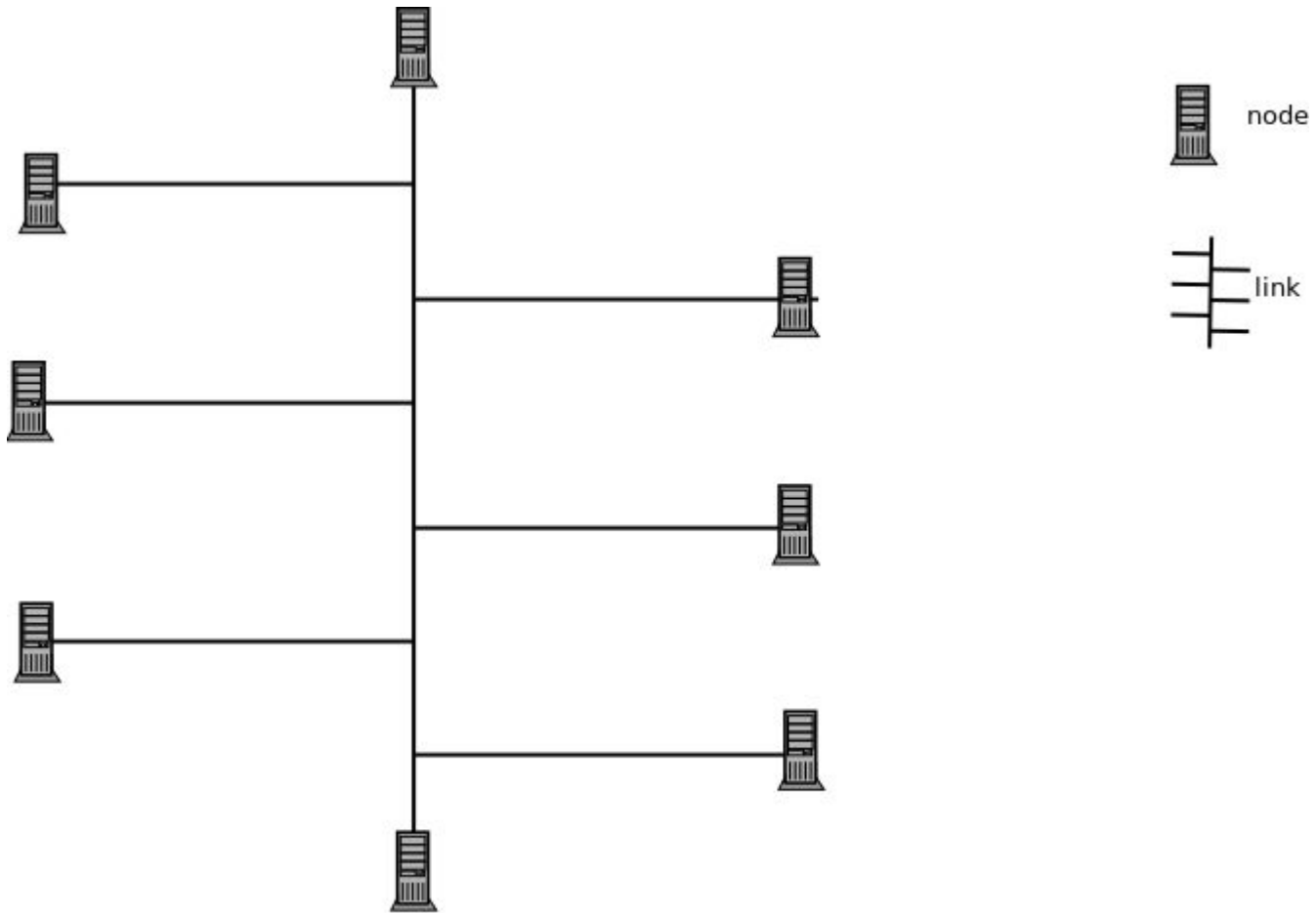
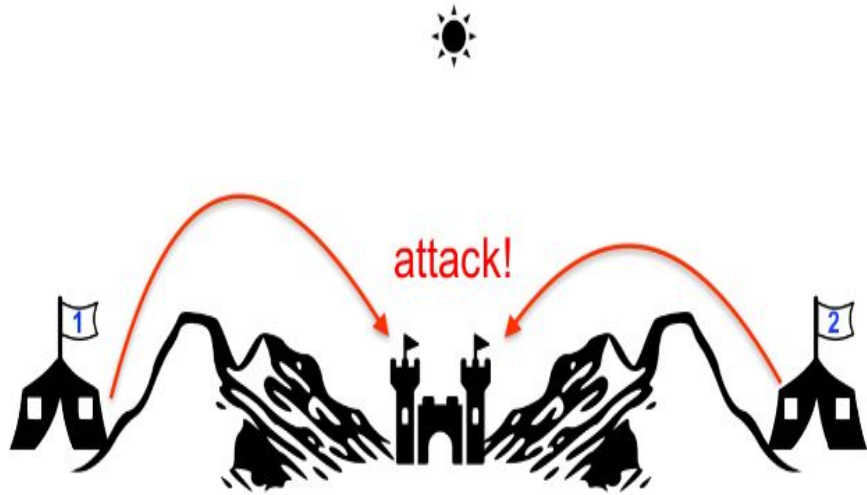


Figure 1: Distributed Systems with nodes

Two Generals' problem



S. Haridi, KTHx 1d2203.1x

Figure 2: Two general problem

FLP Impossibility of Consensus

This describes the necessary conditions for consensus to be achieved. The following properties must hold:

- Agreement (every node must have the same value)
- Validity (only decide on values that were previously proposed)
- Termination (quit the algorithm after the decision has been made)

This is shown when consensus is impossible in an asynchronous, synchronous, and partially synchronous system.

Consensus cannot be solved in an asynchronous system if there is a single failure. Hence, **failure detectors** are needed.

Consensus cannot be solved in a synchronous system if $N-1$ nodes fail.

Consensus is solvable in a partially synchronous system with up to $N/2$ crashes.

Primer on Parallel programming

Modern computers can multitask. It is desirable to exploit this property to efficiently solve our problem. Our focus will be on the following:

- Multi-Core programming
 - Handling parallelism by running on multiple processor
- Multi-Threading
 - Slicing a program using the scheduler to run several portions at intervals.
 - Context switching

When to use concurrency?

- Task can be split and independent (minimal signaling)
- Divide and Conquer

Amdahl's law ?

Synchronization

Synchronization is to make two things to happen simultaneously at a given time

Synchronization constraints:

- **Serialization**
 - Events are ordered
- **Mutual exclusion**
 - Two events does not happen at the same time

It is easy to impose synchronization constraint by using a global clock.

Synchronization has mutexes, semaphores, monitors, conditional variable

The focus is on using **semaphores** and mutexes

Thread A (You)

```
1 Eat breakfast
2 Work
3 Eat lunch
4 Call Bob
```

Thread B (Bob)

```
1 Eat breakfast
2 Wait for a call
3 Eat lunch
```

Figure 3: Example of two people seeking to meet to launch

Concurrency

Concurrency is when it is impossible to determine the order of execution from looking at the source code.

Non-determinism makes debugging harder.

Thread safety: manipulating shared resources in a way to prevent side effects.

Level of thread safety

- Thread safe: free from race condition when handle by multiple threads.
- Conditionally safe (partial thread safe)
- Not thread safe

How to achieve Thread Safety

- Re-entrancy: swappable threads still gives the desired computation.
- thread-local storage
- Immutable objects

https://en.wikipedia.org/wiki/Thread_safety

Local variable have minimal synchronization problems. Shared variable are the bone of contention for synchronization

Atomic variable: cannot be preempted

Semaphores

Semaphores: in real life, it is a visual method of communication between people using lights, flags among others.

Formulation of semaphores

- On initialize, it is set to a defined value.
- Impossible to read the current value of a semaphore **?WHY**
- Thread can increment or decrement the semaphores.
- On decrement, if the value is negative, the thread blocks until another thread wakes it up.
- On increment, after waking up, there is no guarantee on the order of the thread to be executed by the scheduler.

Semaphores (Continue)

- Impossible to know if a semaphore would block or not.
- Impossible to know if there are threads waiting when if there a new thread signals the semaphores, so it may not be 0
- After an increment operation by a thread on a semaphore and a thread is woken up, both threads will run concurrently.

Thread states

- Block: notify the scheduler not to run it
- Unblock: notify the scheduler that it can run

Semaphores (Continue)

`sem = Semaphore (1)`

`sem.increment () / sem.signal()`

`sem.decrement () / sem.wait()`

Advantages of using semaphores

- Solutions with semaphore are clean and less risk of errors
- It is efficient in many systems

Signal vs Wait

- Signaling is a use case for semaphores. This is when a thread sends a signal to other thread to show that an event has occurred.

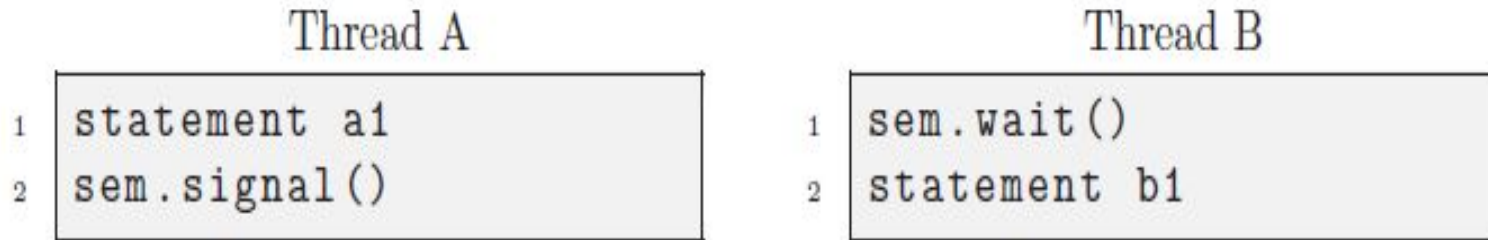


Figure 4: Show diagram of how to $a1 < b1$ (initial sem = 0)

Thread A

```
1 statement a1  
2 statement a2
```

Thread B

```
1 statement b1  
2 statement b2
```

Figure 5: Show examples of two thread

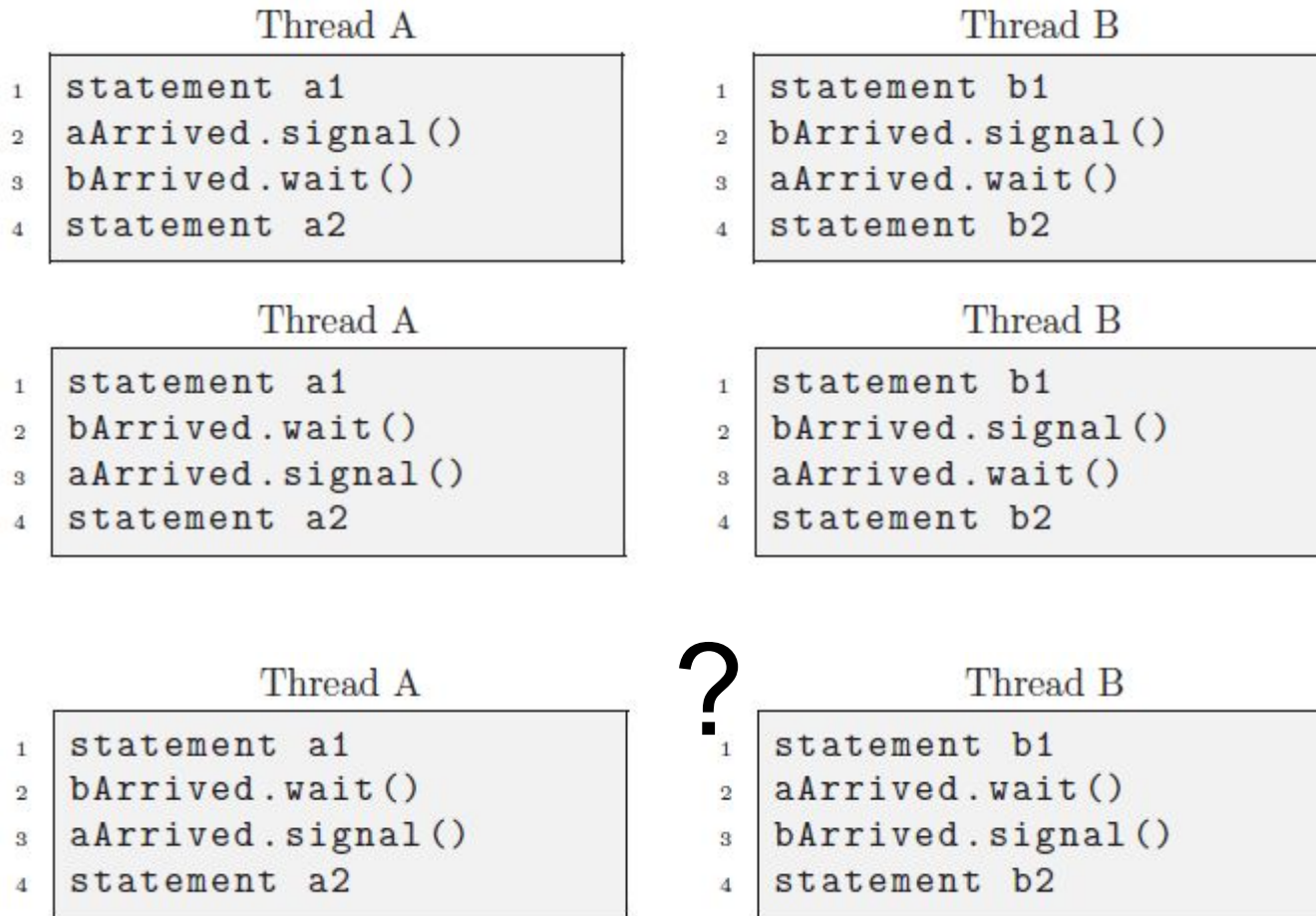


Figure 6: Synchronizing two threads

Mutex

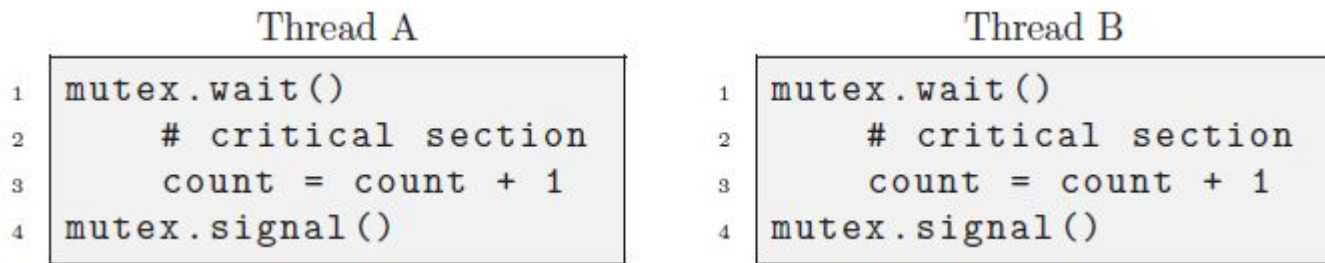


Figure 7: Show a counter as an example

- Symmetric solution (all threads run the same code)
- Asymmetric solution (multiple threads run the multiple code)

Barrier

```
1 rendezvous  
2 critical point
```

- The goal is that all the threads will not enter the critical section until the rendezvous is complete
- Barrier is locked until every thread has arrived.

Turnstile

```
9 | barrier.wait()  
10 | barrier.signal()
```

- Barrier requires a turnstile
- Turnstile is a rapid wait and signal in succession with initial value set to 0 is locked

Downey, The Little Book of Semaphores

- This allows one thread to proceed at a time, and stop all other threads
- If locked, the nth thread unlocks it, then all threads go through

```
1 rendezvous
2
3 mutex.wait()
4     count = count + 1
5 mutex.signal()
6
7 if count == n: barrier.signal()
8
9 barrier.wait()
10 barrier.signal()
11
12 critical point
```

Downey, The Little Book of Semaphores

Better Version of Barrier

```
1 # rendezvous
2
3 mutex.wait()
4     count += 1
5     if count == n:
6         turnstile2.wait()           # lock the second
7         turnstile.signal()         # unlock the first
8 mutex.signal()
9
10 turnstile.wait()                   # first turnstile
11 turnstile.signal()
12
13 # critical point
14
15 mutex.wait()
16     count -= 1
17     if count == 0:
18         turnstile.wait()           # lock the first
19         turnstile2.signal()        # unlock the second
20 mutex.signal()
21
22 turnstile2.wait()                  # second turnstile
23 turnstile2.signal()
```

2-phase barrier

It force thread to block twice

- for every thread arriving the critical section
- for every thread departing the critical section

Disadvantage may lead to more context switching

Read more about **preloaded turnstile**

Downey, The Little Book of Semaphores

OpenMPI

Two-sided communication

- MPI_Irecv
- MPI_Isend
- MPI_Waitall
- MPI_Recv
- MPI_Send

One-sided communication

- MPI_WIN_create()
- MPI_WIN_allocate()
- MPI_GET()MPI_PUT()
- MPI_Accumulate()
- MPI_Win_free()

Best Practices for One-sided Communication

- No user-defined operation in MPI_Accumulate.
- Ensure local completion before accessing the buffer in an epoch.
- It is impossible to mix MPI_GET, MPI_PUT, MPI_Accumulate in a single epoch

Logical Clocks

Lamport Clocks

Source code: <https://github.com/kenluck2001/DistributedSystemResearch/blob/master/blog/lamport1.c>

- Each process has a local **logical clock**, kept in variable t_p , initially $t_p = 0$
 - A process p piggybacks (t_p, p) on every message sent
- On internal event a :
 - $t_p := t_p + 1$; perform internal event a
- On send event message m :
 - $t_p := t_p + 1$; send($m, (t_p, p)$)
- On delivery event a of m with timestamp (t_q, q) from q :
 - $t_p := \max(t_p, t_q) + 1$; perform delivery event a

Vector Clocks

Source code: <https://github.com/kenluck2001/DistributedSystemResearch/blob/master/blog/vector2.c>

- The happen-before relation is a partial order
- In contrast logical clocks are total
 - Information about non-causality is lost
 - We cannot tell by looking to the timestamps of event a and b whether there is a causal relation between the events, or they are **concurrent**
- Vector clocks guarantee that:
 - if $\mathbf{v}(a) < \mathbf{v}(b)$ then $a \rightarrow_{\beta} b$, in addition to
 - if $a \rightarrow_{\beta} b$ then $\mathbf{v}(a) < \mathbf{v}(b)$
 - where $\mathbf{v}(a)$ is a vector clock of event a

S. Haridi, KTHx 1d2203.1x

Vector Clocks (Continue)

- Processes p_1, \dots, p_n
- Each process p_i has local vector \mathbf{v} of size n (number of processes)
 - $\mathbf{v}[i] = 0$ for all i in $1 \dots n$
 - Piggyback \mathbf{v} on every sent message
- For each transition (on each event) update local \mathbf{v} at p_i :
 - $\mathbf{v}[i] := \mathbf{v}[i] + 1$ (internal, send or deliver)
 - $\mathbf{v}[j] := \max(\mathbf{v}[j], \mathbf{v}_q[j])$, for all $j \neq i$ (deliver)
 - where \mathbf{v}_q is clock in message received from process q

S. Haridi, KTHx 1d2203.1x

Vector Clocks (Continue)

- $v_p \leq v_q$ iff
 - $v_p[i] \leq v_q[i]$ for all i
- $v_p < v_q$ iff
 - $v_p \leq v_q$ and for some i , $v_p[i] < v_q[i]$
- v_p and v_q are concurrent ($v_p \parallel v_q$) iff
 - not $v_p < v_q$, and not $v_q < v_p$
- Vector clocks guarantee
 - If $v(a) < v(b)$ then $a \rightarrow b$, and
 - If $a \rightarrow b$, then $v(a) < v(b)$
 - where $v(a)$ is the vector clock of event a

$$(3,0,0) \leq (3,1,0)$$

$$[3,0,0] < [3,1,0]$$

$$[3,1,0] \langle \rangle [4,0,0]$$

S. Haridi, KTHx 1d2203.1x

Consensus

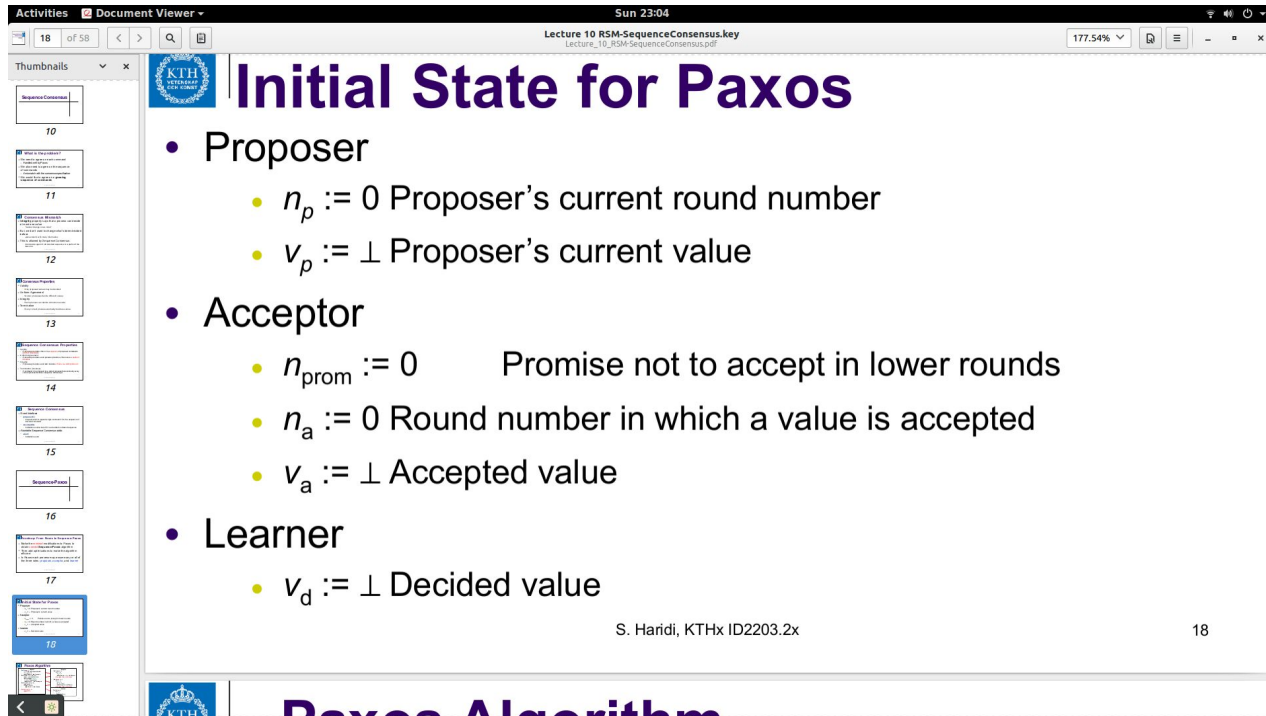
- All correct nodes propose a value.
- Every node decides on the same value.
- Only decide on proposed values.
- Proposed values are either committed or aborted.

Atomic broadcast (All correct node deliver same message).

Our use case will favour linearizability / atomic consistency (multiple nodes) vs sequential consistency (single node).

Paxos (Single Value)

Source code: <https://github.com/kenluck2001/DistributedSystemResearch/blob/master/blog/single-paxos3.C>

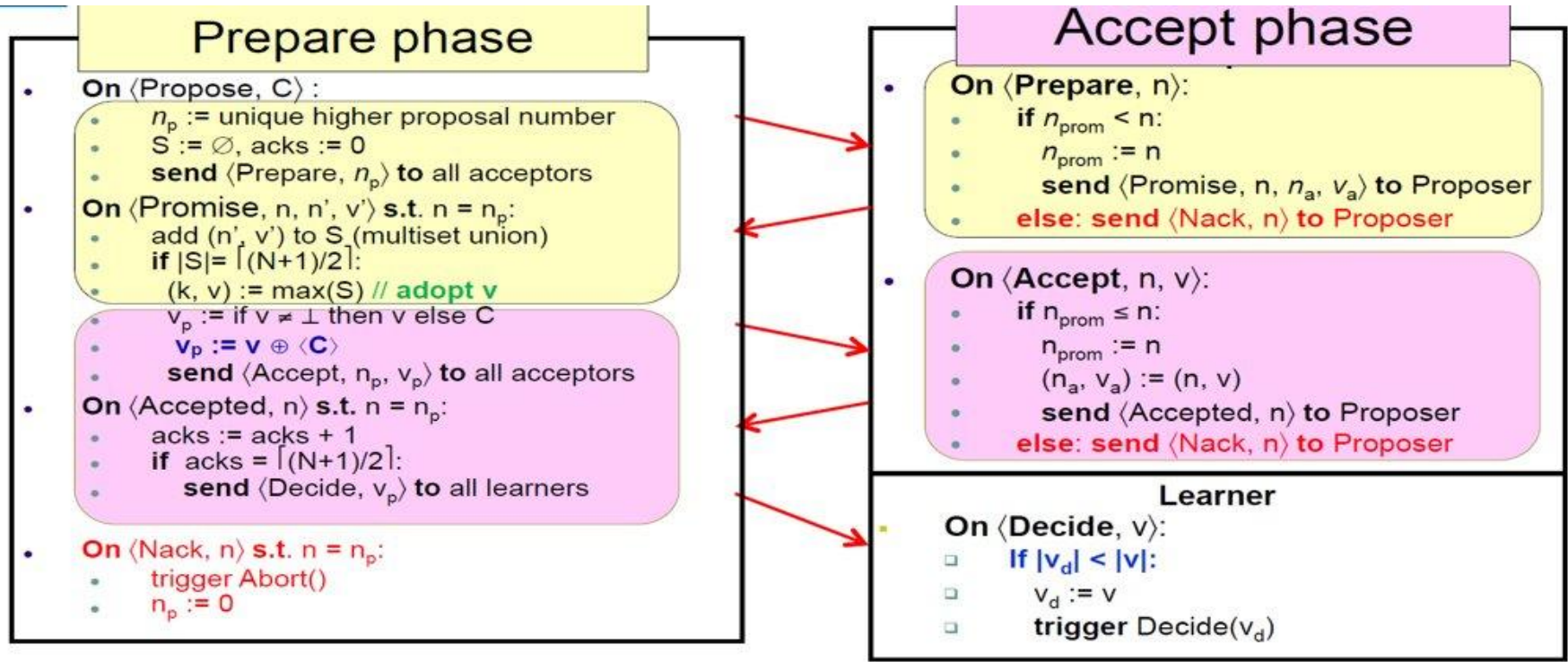


The image shows a presentation slide titled "Initial State for Paxos" from a document viewer. The slide is part of a presentation on "Lecture 10 RSM-SequenceConsensus.key". The slide content is as follows:

- Proposer
 - $n_p := 0$ Proposer's current round number
 - $v_p := \perp$ Proposer's current value
- Acceptor
 - $n_{\text{prom}} := 0$ Promise not to accept in lower rounds
 - $n_a := 0$ Round number in which a value is accepted
 - $v_a := \perp$ Accepted value
- Learner
 - $v_d := \perp$ Decided value

At the bottom of the slide, it says "S. Haridi, KTHx ID2203.2x" and "18".

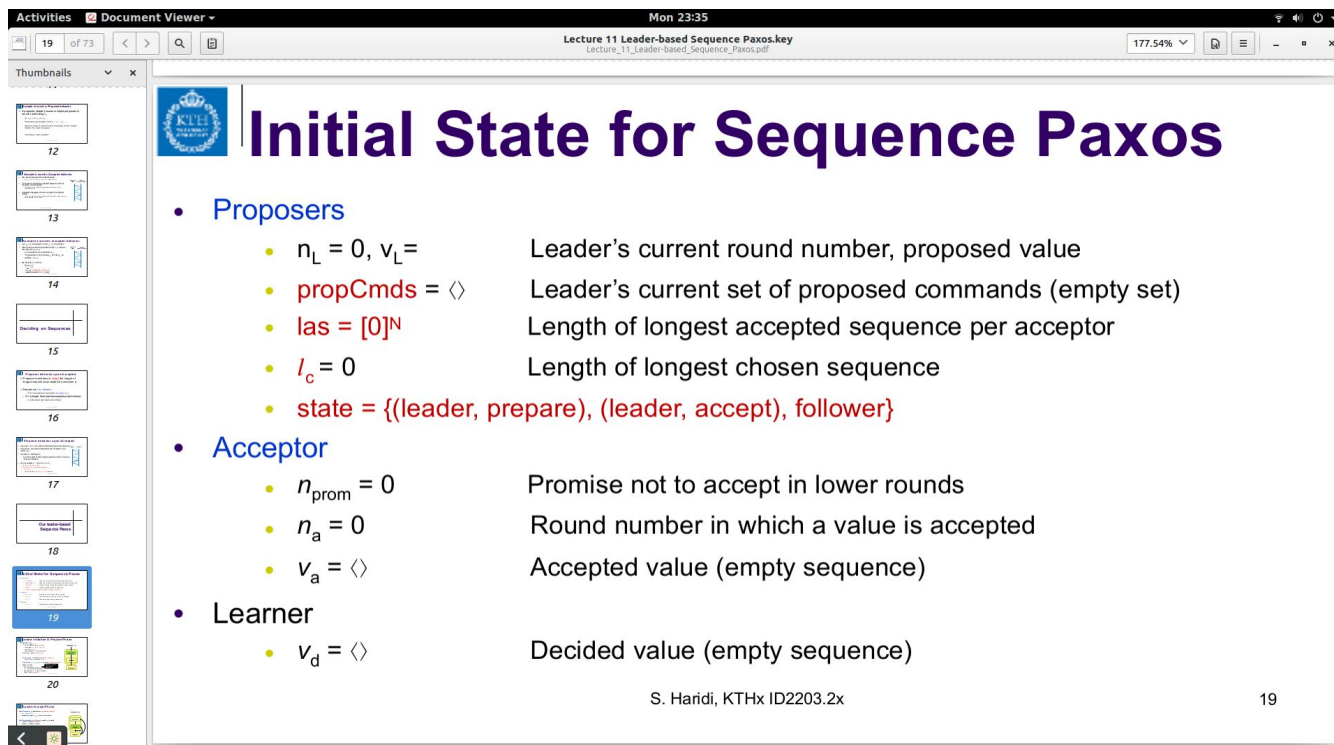
Paxos (Single Value)



Max(S) is any element (k, v) of S s.t k is highest proposal number

Paxos (Sequence)

Source code: <https://github.com/kenluck2001/DistributedSystemResearch/blob/master/blog/sequence-paxos4.c>



The screenshot shows a document viewer window titled "Lecture 11 Leader-based Sequence Paxos.key". The slide content is as follows:

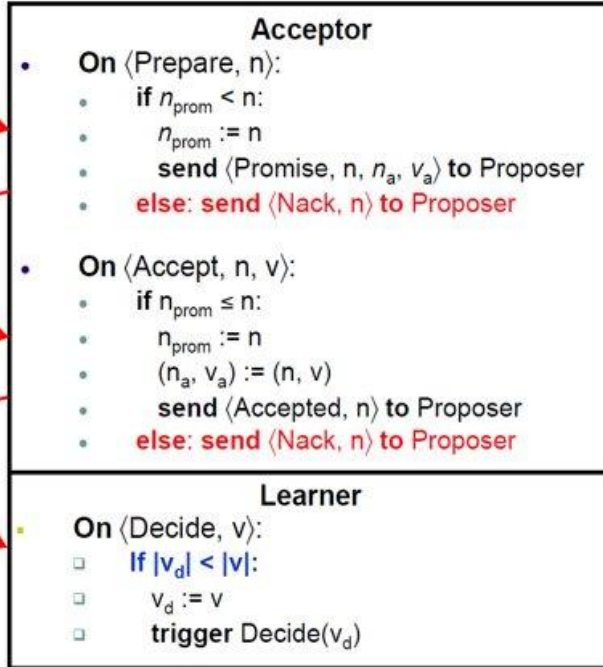
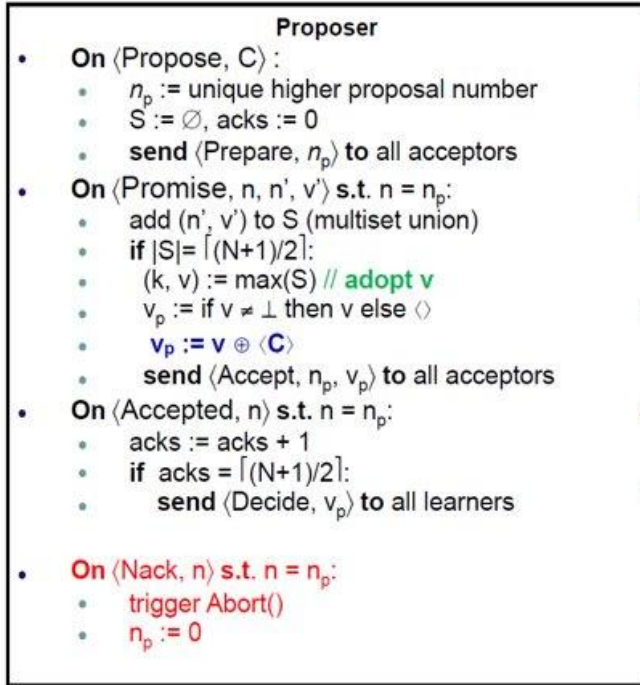
Initial State for Sequence Paxos

- **Proposers**
 - $n_L = 0, v_L =$ Leader's current round number, proposed value
 - $\text{propCmds} = \langle \rangle$ Leader's current set of proposed commands (empty set)
 - $\text{las} = [0]^N$ Length of longest accepted sequence per acceptor
 - $l_c = 0$ Length of longest chosen sequence
 - $\text{state} = \{(\text{leader, prepare}), (\text{leader, accept}), \text{follower}\}$
- **Acceptor**
 - $n_{\text{prom}} = 0$ Promise not to accept in lower rounds
 - $n_a = 0$ Round number in which a value is accepted
 - $v_a = \langle \rangle$ Accepted value (empty sequence)
- **Learner**
 - $v_d = \langle \rangle$ Decided value (empty sequence)

S. Haridi, KTHx ID2203.2x

19

Paxos (Sequence)



$S = \{(n_1, v_1), \dots, (n_k, v_k)\}$
 fun $\text{max}(S)$:
 $(n, v) =: (0, \langle \rangle)$
 for (n', v') in S :
 if $n < n'$ or $(n = n'$ and $|v| < |v'|)$:
 $(n, v) := (n', v')$
 return (n, v)

Failure Detector

Source code: <https://github.com/kenluck2001/DistributedSystemResearch/blob/master/blog/failure-detector2.c>

Working principles:

- A way to identify failures among nodes.
- Periodically exchange heartbeat message.
- Mark delayed processes as suspected, modify time deltas and if exceeded again and repeat if set epoch is exceeded, marks the process as dead.
- Hence, trade off between completeness and accuracy.

Contrast the difference between Failure detection and Leader election

- Failure detector identifies failed processes
- Leader election detects correct process
- Leader election is a failure detector. Hence, always suspect all processes except leader

Failure Detector (Continue)

The algorithm does the following:

- Each node has a failure detector
- initially wrong, but eventually correct
- periodically exchange heartbeat messages with every supposedly alive process
- if timeout, then suspect process
- if a message is received from a suspected node, revise suspicion, and increase the timeout.
- Otherwise, detects a crash

For a failure detector to be useful, it must meet the requirements with varying certainty.

- **Completeness:** (when do crash nodes get detected?)

Every crashed process is eventually detected by every correct process (liveness).

- **Accuracy:** (when do alive nodes get suspected?)

No correct process is ever suspected (safety).

Leader Election

Source code: <https://github.com/kenluck2001/DistributedSystemResearch/blob/master/blog/leader-election2.c>

- There are problems with multiple proposers. Hence, we can designate a single proposer as the leader.
- A leader can transition to a follower and vice versa.

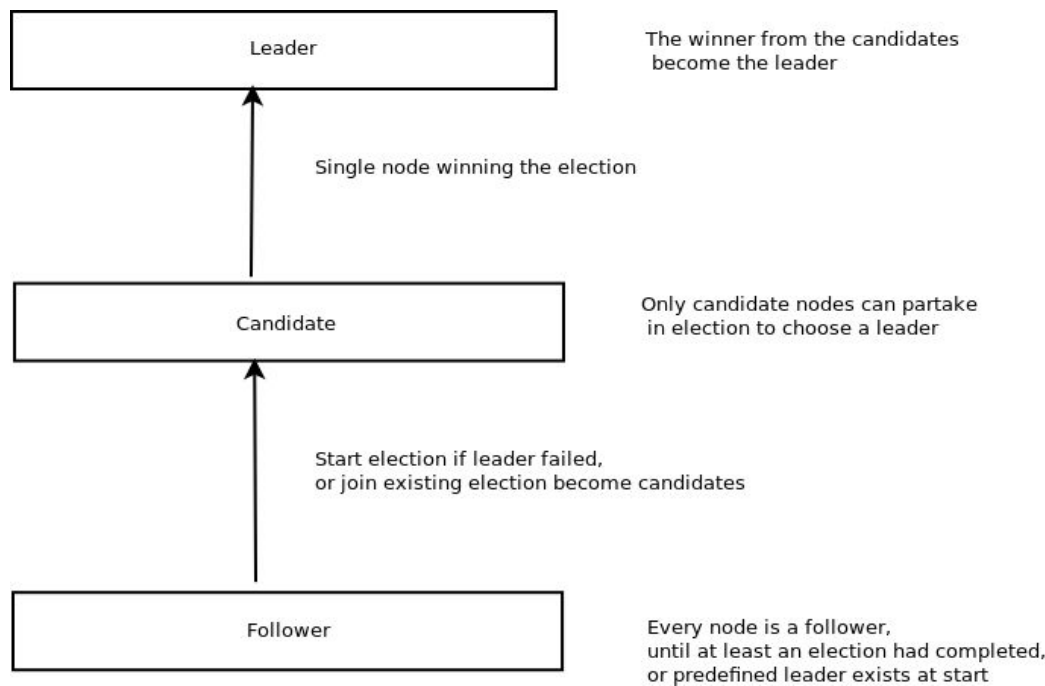


Figure 8: Description of Leader election

Leader Election (Continue)

The rule of thumb on deciding duration

- if the value is set too low, then the second candidate begins election before the end of the first election triggered by the first candidate.
- if too high, then it will take too long for the election to start after the old leader has died. The new candidate starts an election.

Discuss possibilities for **optimization**?

https://kenluck2001.github.io/blog_post/distributed_computing_from_first_principles.html#leader-election

Raft

Raft is a leader election-based sequence Paxos. It consists of Paxos, log, and leader.

Using a combination of [sequence paxos](#) and [leader election](#).

Leader election is called if the leader is dead.

There are roles in the Raft algorithm

- Candidate: node aiming to be a leader.
- Leader: it is the candidate that is chosen as a leader.
- Follower: a participant that is not engaged in the election.

Possible problems:

- Multiple leaders
- No leaders
- Missing log entries
- Divergent log

Can an election choose multiple leaders? yes

Can an election fail to choose a leader? yes

Raft (Continue)

```
use leader election to get the leader and use as proposer
if (rank == leader)
{
    // use as proposer
}
else
{
    isacceptor = rank % 2 //or other forms of grouping
    if (isacceptor)
    {
        // use as acceptor
    }
    else
    {
        // use as learner
    }
}
```

use leader election to get the leader and use as proposer

```
if (rank == leader)
{
    // use as proposer
}
else
{
    islearner = (leader + 1) % n //or other forms of grouping where n:
    total number of processes
    if (islearner)
    {
        // use as learner
    }
    else
    {
        // use as acceptor
    }
}
```

Implementation details of possible Raft patterns

Anti-entropy mechanisms

- CRDT
 - Conflict-free replicated data type
- Ancillary Structures

CRDT

This is a data structure that maintains a consistent state irrespectively of the order of operations executed on it.

Remote syncing (replication) across multiple devices can be challenging to achieve. Mathematically, it is a partially ordered monoid forming a **lattice**. It has to be commutative and idempotent.

- Eventual consistency
- Preserve ordering of the data
- Local-first application

Ancillary Structures

- Merkle trees
- Error control codes
 - Turbo codes, Reed–Solomon codes

Case Studies

- Distributed shared primitives

- **Source code / implementation:**

- https://kenluck2001.github.io/blog_post/distributed_computing_from_first_principles.html#distributed-shared-primitive

- Distributed hashmap

- **Source code:**

- <https://github.com/kenluck2001/DistributedSystemResearch/blob/master/blog/lamport1-majority-voting8.c>

Code Philosophy / Lessons

- Our philosophy has been to think locally and act globally. We do computations on the node (update internal state) and communicate with other nodes by messaging.
- Retrieving messages and probing to check the tag of messages to identify a specific event. Busy waiting is used to retrieving messages on an irecv. Otherwise, only the last sent is received on polling. This can be a bug where you retrieve the same message multiple times.
- It is good to take steps to avoid both deadlocks and livelock. Deadlock can happen in mismatch message order between send and receive, especially in blocking mode. It is possible in non-blocking mode to consider how request objects are owned between the receiving and sending nodes.

- Rather than communicating by sharing memory, it is better to share memory by communicating.
- For the Paxos algorithm when using Unix timestamps as the round number or ballots for their monotonically increasing properties, then a necessary prerequisite is to synchronize the time settings on at least the set of proposers.
- Organizing the processes into groups with custom communicators. This allows for targeted synchronization for grouped processes without impacting the whole processes in the application.
- When trying to create an array of atomic counters. It is desirable to utilize an array of shared pointers, rather than an array of shared values.
- Livelock is possible too in a non-blocking case when we pull in a busy wait manner. As we exit from the end of the loop when we have received the expected number of messages. It can be sensible to keep track of the number of exchanged messages to force an exit from the endless loop.

- We can cancel pending requests and tune the criteria for quorum based on business needs. This would impact **resilience** on the Distributed Systems.
- We use pooling on receiving the message and checking each tag, rather than waiting on specific tags to make code modular.
- Always pool on waiting reads in a busy-wait style.
- Make use of simple structure. Even our log for sequence Paxos is not a log, but an abused linked list with some atomic primitives.
- Our sequence Paxos uses single Paxos on each item that the proposer will send. Unfortunately, our logic is restricting to only the possibility of having one proposer.
- There are problems with passing **pointers** across nodes. This is because we don't have a universal shared memory. Always keep pointers **local** as a lack of Distributed memory makes indirection on a pointer useless.

Exercises

- Write tests for the Distributed algorithm discussed in the blog. We will give the user the task of implementing tests as an exercise.
- Implement telemetry for experimentation on characteristics of any algorithm
- Set up a test bed with a [simulated LAN with vagrant VM](#) for running MPI cluster.
- implement network shaping using VM to test out different performances in varying network bandwidth.

Conclusions

Various implementation of distributed systems can vary along:

- Who partakes in the leadership election? Is it a client with read or write privileges?
- Value of time delta in the failure detector?
- Optional server to help late clients sync to get decided information from past rounds?
- Set fixed threshold on rounds that are considered in quorum?
- Number of clients that can participate in a quorum?

Writing a **textbook on Distributed Systems** (

https://kenluck2001.github.io/blog_post/authoring_a_new_book_on_distributed_computing.html)

References

1. <http://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf> by Allen Downey
2. Database Internals: A Deep-dive into how distributed data systems works by Alex Petrov.
3. <https://e-science.se/2020/05/course-on-reliable-distributed-systems-part-i/>
4. <http://wgropp.cs.illinois.edu/courses/cs598-s15/lectures/lecture34.pdf>