FIT3143

Parallel Computing

Week 7: Faults, Distributed Consensus

Vishnu Monn and ABM Russel

Unit Topics

Week	Lecture Topic	Tutorial	Laboratory	Remarks
1	Introduction to Parallel Computing and Distributed Systems	None	Lab Week 01 - Introduction to Linux & Setting up VM (Not assessed)	Group formation for lab activities and assignments (Two students per group)
2	Parallel computing on shared memory (POSIX and OpenMP)	Tutorial Week 02 - Introduction to Parallel Computing	Lab Week 02 - C Primer (Not assessed)	
3	Inter Process Communications; Remote Procedure Calls	Tutorial Week 03 - Shared memory parallelism	Lab Week 03 - Threads (POSIX)	Assignment 1 specifications released
4	Parallel computing on distributed memory (MPI)	Tutorial Week 04 - Inter process communication	Lab Week 04 - Threads (OpenMP)	
5	Synchronization, MUTEX, Deadlocks	Tutorial Week 05 - Distributed memory parallelism	Lab Week 05 - Message passing interface	Assignment 2 specifications released
6	Election Algorithms, Distributed Transactions, Concurrency Control	Tutorial Week 06 - Synchronization	Lab Week 06 - Communication patterns	
7	Faults, Distributed Consensus, Security, Parallel Computing	Tutorial Week 07 - Transactions and concurrency	Lab Week 07 - Parallel data structure (I)	Assignment 1 due
8	Instruction Level Parallelism	Tutorial Week 08 - Consensus	1st assignment interview	
9	Vector Architecture	Tutorial Week 09 - Instruction level parallelism	Lab Week 09 - Parallel data structure (II)	
10	Data Parallel Architectures, SIMD Architectures	Tutorial Week 10 - Vector architecture	Lab Week 10 - Master slave	
11	Introduction to MIMD, Distributed Memory MIMD Architectures	Tutorial Week 11 - Data parallelism	Lab Week 11 - Performance tuning	Assignment 2 due
12	Recent development in Parallel Computing - Software, Hardware, Applications etc. (GPGPU etc.), Exam Revision	Tutorial Week 12 - Revision	2nd assignment code demo and interview	

Assessment	Weight
Assignment 1	15%
Assignment 2	25%
Lab-work and Quizzes	10%
Final exam	50%

Feedback

- Assignment - more consultations and release Assignment 2 early during mid semester break.
- Lecture - focus on key points (due to being no prior knowledge unit) with more real life examples
- Tutorial - more guidance from tutors and group interaction continues
- Lab - pre lab activities continues
- Consultations - more ad-hoc sessions as required

Background

- Concurrency vs Parallellism vs Distributed

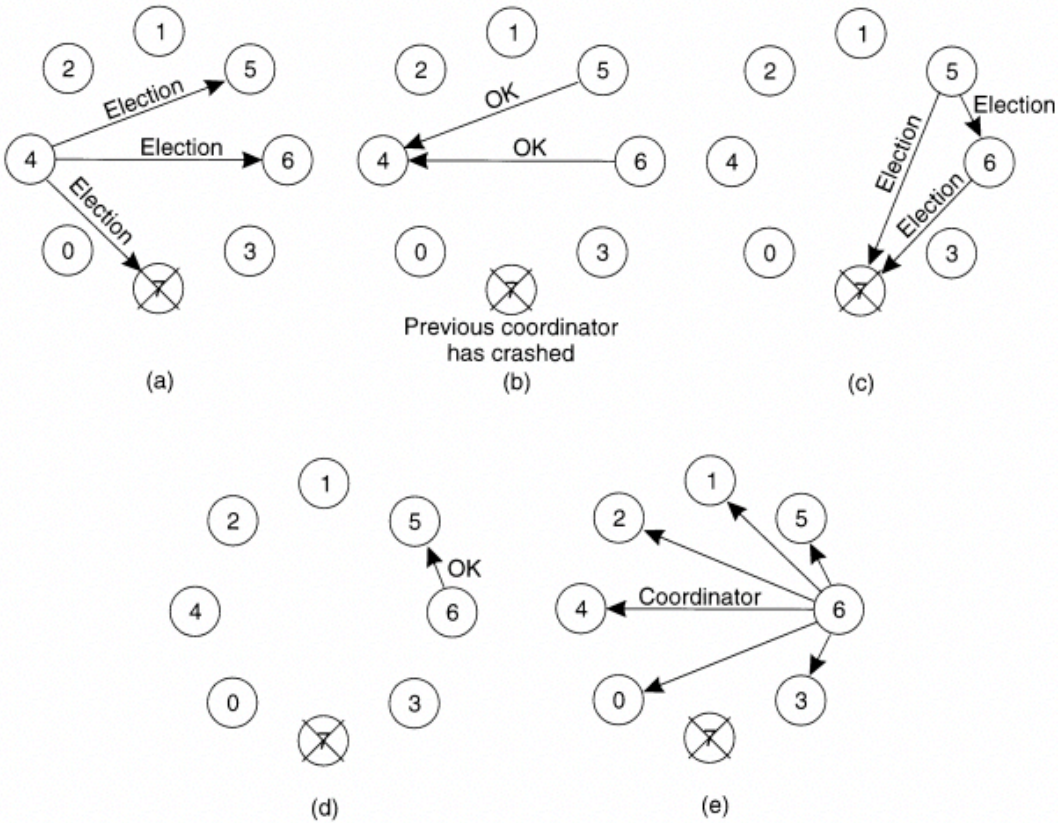
Today

- Recap: Election Algorithms, Transactions, Concurrency
- Faults and Fault-Tolerant Systems
- Distributed Consensus

Recap

FLUX: Election

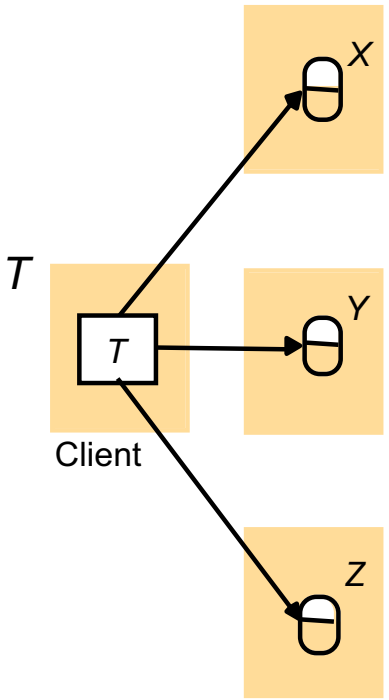
Election

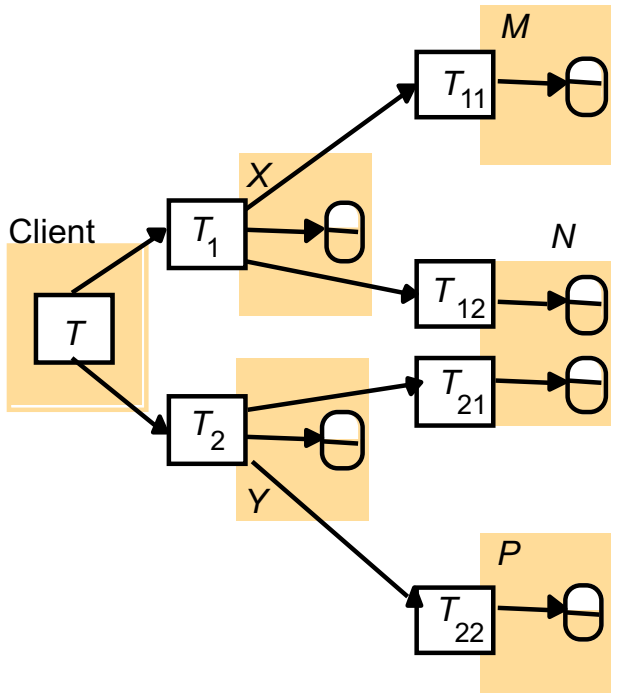


JYGAFY

FLUX: Transactions

Transactions





JYGAFY

FLUX: Concurrency

Concurrency

JYGAFY

Fault-Tolerant Systems

Dependability

Building a dependable system comes down to controlling failure and faults.

- Availability: System is ready to be used immediately
- Reliability: System can run continuously without failure
- Safety: When a system (temporarily) fails to operate correctly, nothing catastrophic happens
- Maintainability: How easily a failed system can be repaired

Total vs Partial Failure

- Total Failure:
  - All components in a system fail
  - Typical in non-distributed system
- Partial Failure:
  - One or more (but not all) components in a distributed system fail
  - Some components affected
  - Other components completely unaffected
  - Considered as fault for the whole system

Failure

- Failure: a system fails when it fails to meet its promises or cannot provide its services in the specified manner
- Error: part of the system state that leads to failure (i.e., it differs from its intended value)
- Fault: the cause of an error (results from design errors, manufacturing faults, deterioration, or external disturbance)
- Recursive:
  - Failure may be initiated by a mechanical fault
  - Manufacturing fault leads to disk failure
  - Disk failure is a fault that leads to database failure
  - Database failure is a fault that leads to email service failure

Classification of failures

- Our view of a distributed system is a process-level view, so we begin with the description of certain types of failures that are visible at the process level.
- Major classes of failures
  - Crash Failure
  - Omission Failure
  - Transient Failure
  - Byzantine Failure
  - Software Failure
  - Temporal Failure
  - Security Failure

Crash Failure

- A process undergoes crash failure, when it permanently ceases to execute its actions. This is an irreversible change.
    - In an asynchronous model, crash failures cannot be detected with total certainty, since there is no lower bound of the speed at which a process can execute its actions.
    - In a synchronous system where processor speed and channel delays are bounded, crash failures can be detected using timeouts.

Omission Failure

- If the receiver does not receive one or more of the messages sent by the transmitter, an omission failure occurs. For wireless networks, collision occurs in MAC layer or receiving node moves out of range.

Transient Failure

- A transient failure can disturb the state of processes in an arbitrary way. The agent inducing this problem may be momentarily active but it can make a lasting effect on the global state. E.g., a power surge, or a mechanical shock, or a lightening.

Byzantine Failure

- Byzantine Failure
  - Byzantine failures represent the weakest of all failure model that allows every conceivable form of erroneous behaviour. The term alludes to uncertainty and was first proposed by Pease et al.
  - Assume that process i forwards the value x of a local variable to each of its neighbours. The followings inconsistencies may occur:
    - two distinct neighbours j and k receive values x and y, where $x \neq y$
    - one or more neighbours do not receive any data from I
    - every neighbour receives a value z where $z \neq x$

Causes of the Byzantine failures

- Some possible causes of the Byzantine failures are:
  - total or partial breakdown of a link joining i with one of its neighbours
  - software problems in process i
  - hardware synchronization problems – assume that every neighbour is connected to the same bus, and reading the same copy sent out by i, but since the clocks are not perfectly synchronized, they may not read the value of x at the same time. If value of x varies with time, then different neighbours of i may read different values of x from process i.

Software Failure

- Primary causes of software failure:
- **Coding error or human errors**: program fails to use the appropriate physical parameters. September 23, 1999 NASA lost $125 million Mars Orbiter spacecraft because one engineering team used metric units while another used English units, leading to a navigation fiasco, causing it to burn in the atmosphere.
- **Software design error** – Mars pathfinder mission landed flawlessly on the Martial surface on July 4, 1997. However, later its communication failed due to a design flaw in the real-time embedded software kernel VxWorks. The problem was later diagnosed to be caused due to **priority inversion**
    - Priority inversion: Low priority task **LP** locks file **F**
    - High priority task **HP** is scheduled next, it also needs to lock file **F**
    - A medium priority **MP** task (with high CPU requirement) becomes ready to run
    - **MP** is the highest priority unblocked task, its allowed to run, consumes all CPU
    - **LP** has no CPU, it stops. **HP 's** priority < **MP's** prioirity (priority inversion)

Memory Leaks

- Memory Leaks
  - Processes fail to fully free up the physical memory that has been allocated to them. This effectively reduces the size of available physical memory over time. When the available memory falls below the minimum requirement by the system, a crash becomes inevitable.
- Problem with inadequacy of specification e.g. Y2K bug
  - **Note:** that many of the **failures** like crash, omission, transient and Byzantine can be caused by **software bugs**. For example, a poorly designed loop that does not terminate can mimic a **crash failure** in the sender process. An inadequate policy in the router software can cause packets to drop and trigger **omission failure**.

Temporal Failure

- Real-time systems require actions to be completed within a specific time frame. When this time limit is not met, a temporal failure occurs.

Security

- Virus and other malicious software may lead to unexpected behaviour that manifests itself as a system fault.

Human Errors

- Human errors play can play a role in system failure.
  - In November 1988, much of the long distance service along the East Coast of USA was disrupted when a construction crew accidentally detached a major fibre optic cable in New Jersey; as a result 3,500,000 call attempts were blocked.
  - On September 17, 1991 AT&T technicians in NY attending a seminar on warning systems failed to respond to an activated alarm for six hours. The resulting power failure blocked nearly 5 million domestic and international calls and paralysed air travel throughout the Northeast, causing nearly 1,170 flights to be cancelled or delayed.

History of Fault Tolerant Systems

- The first known fault-tolerant computer was SAPO, built in 1951 in Czechoslovakia by Antonin Svoboda.

- Most of the development in the so called LLNM (Long Life, No Maintenance) computing was done by NASA during the 1960's, in preparation for Project Apollo and other research aspects. NASA's first machine went into a space observatory, and their second attempt, the JSTAR computer, was used in Voyager. This computer had a backup of memory arrays to use memory recovery methods and thus it was called the JPL Self-Testing-And-Repairing computer. It could detect its own errors and fix them or bring up redundant modules as needed.

- Smart Sensor Network in the Ageless Space Vehicle Project [Don Price et al]

Fault-Tolerant System

- We designate a system that does not tolerate failures as a fault-tolerant system. In such systems, the occurrence of a fault violates ***liveness*** and ***safety*** properties.
- The are four major types of fault-tolerance
    - Masking tolerance
    - Non-masking tolerance
    - Fail-safe tolerance
    - Graceful degradation

Note:

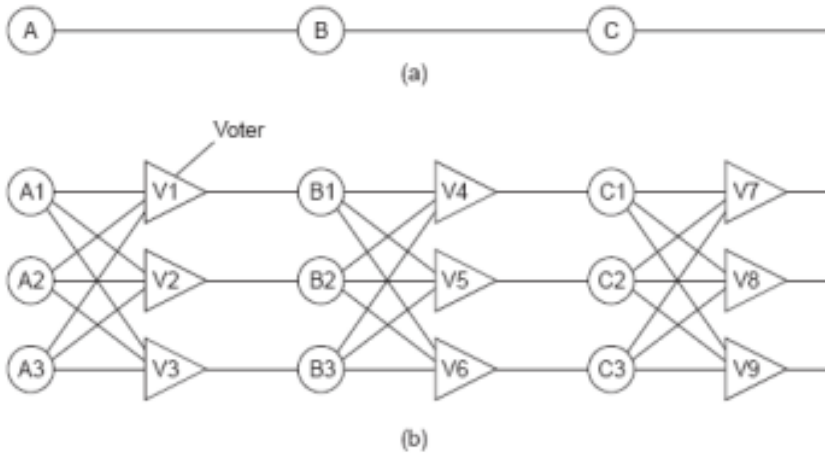
- ***Safety*** properties specify that “something bad never happens
    - Doing nothing easily fulfils a safety property as this will never lead to a “bad” situation
- ***Safety*** properties are complemented by ***liveness*** properties
- ***Liveness*** properties assert that: “something good” will eventually happen [Lamport]

Masking Tolerance

- Let P be the set of configurations for the fault-tolerance system.
- Given a set of fault actions F, the fault span Q corresponds to the largest set of configurations that the system can support.
- In Masking tolerance system, when a fault F is masked its occurrence has no impact on the application, that is $P = Q$.
- Masking tolerance is important in many safety-critical applications where the failure can endanger human life or cause massive loss of properties.
- An aircraft must be able to fly even if one of its engines malfunctions.
- Masking tolerance preserve both ***safety*** and ***liveness*** properties of the original system.

Implementing Failure Masking

- Introduce Redundancy
  - Information redundancy
  - Time redundancy
  - Physical redundancy



Non-Masking Tolerance

- In non-masking fault tolerance, faults may temporarily affect and violate the ***safety*** property, that is P is subset of Q
- However, ***liveness*** is not compromised, and eventually normal behaviour is restored.
- Consider that while watching a movie, the server crashed, but the system automatically restored the service by switching to a standby proxy server.
- Stabilization and Checkpointing represent two opposing scenario in non-masking tolerance.
    - Checkpointing relies on history and recovery is achieved by retrieving the lost computation.
    - Stabilization is history-insensitive and does not care about lost computation as long as eventual recovery is guaranteed.

Fail-Safe Tolerance

- Certain faulty configurations do not affect the application in an adverse way and therefore considered harmless.
- A fail-safe system relaxes the tolerance requirement by avoiding only those faulty configurations that will have catastrophic consequences (not withstanding the failure).
- As an example,
  - At a two-way traffic crossing, if the lights are green in both directions then a collision is possible. However, if the lights are red, at best traffic will stall but will not have any catastrophic side effect.

Graceful Degradation

- There are systems that neither mask, nor fully recover from the effect of failures, but exhibit a degraded behaviour that falls short of normal behaviour, but is still considered acceptable.
- The notion of acceptability is highly subjective and entirely dependent on the user running the application.
- Some examples
  - While routing a message between two points in a network, a program computes the shortest path. In the presence of a failure, if this program returns another path but not the shortest, then this may be acceptable.
  - An operating system may switch to a safe mode where users cannot create or modify files, but can read the files that already exist.

Distributed Consensus

Bivalent

Univalent

Byzantine General Problem

Why Distributed Consensus

- Example 1: The leader election problem in a network of processes. Each process begins with an initial proposal for leadership. At the end one of it a candidate is elected as a leader, it reflects the final decision of every process. [Week 6]

- Example 2: Fund transfer

- Example 3: Synchronizing clocks [Week 5]

Consensus is easier to achieve in the absence of failures. We will study distributed consensus in the presence of failures.

Problem Definition

The Consensus may be formulated as follows:

- A distributed system contains n processes {0, 1, 2, …, n-1}.
- Every process has an initial value in a mutually agreed domain.
- The challenge is to devise an algorithm, which in spite of the occurrence of failures, allows processes to reach an irrevocable decision that fulfils the following conditions:
  - **Termination.** Every (non-faulty) process must eventually come to a decision.
  - **Agreement.** The final decision of every (non-faulty) process must be identical.
  - **Validity.** If every (non-faulty) process begins with the same initial value v, their final decision must be v.

Consensus in Asynchronous System

- If there is no failure, then reaching an agreement is trivial.
- Reaching consensus, however, becomes surprisingly difficult when one or more members fail to execute actions.
- Assume that at most k members (k>0) can fail.
  - An important finding by Fischer et al. is that in a fully asynchronous system, it is impossible to reach consensus even if k=1.

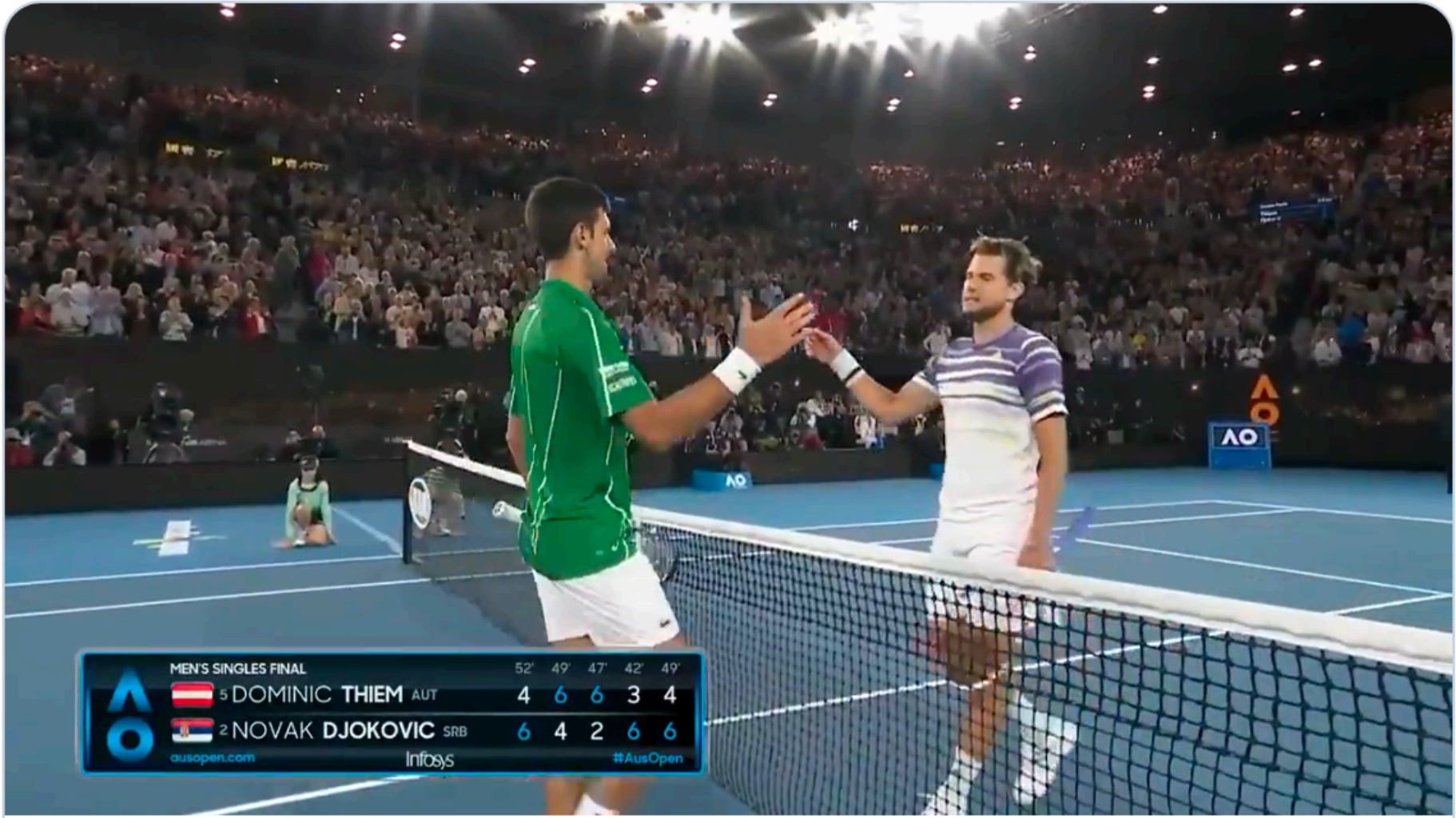
Bivalent and Univalent States

- A decision state is bivalent, if starting from a state, there exist at least two distinct executions leading to two distinct decision values e.g. 0 or 1.

- On the other hand, a state from which only one decision value can be reached is called a univalent state. Univalent state states can be either 0-valent or 1-valent.

- Consider a best-of-five-sets tennis match between A and B. If the score is 6-3, 6-4 in favour of A, the decision state is bivalent, since anyone can win at this point. However, if the score becomes 6-3, 6-4, 7-6 in favour of A, then the state becomes univalent.

AO 2020



Djokovic wins eighth Australian Open title | AO 2020

Image source: AO twitter

The Byzantine General Problem

Leslie Lamport [2002]

- Lamport showed (by proof):
  - For a system of **n+1** nodes, there cannot be more than **n/3** faulty nodes. (if we want to establish distributed consensus)
  - Alternatively: There must be more than **3m** troops in an army with up to **m** traitors to launch a concerted attack.

Byzantine General Problem

Lab Week 7 Overview

- MPI continues

Tutorial Week 7 Overview

- Election Algorithm.
- Distributed Transactions
- Concurrency and parallelism

Next week: ILP

- Instruction Level Parallelism
- Have a safe mid semester break