182.703 Distributed Algorithms (Verteilte Algorithmen)

SS 2015

[v2.11, March 7, 2015]

Prof. Ulrich Schmid

http://ti.tuwien.ac.at/ecs/teaching/courses/valg

Technische Universität Wien
Institut für Technische Informatik
Embedded Computing Systems Group (E182/2)
A View of Distributed Computing
Lamport’s definition of a distributed system:

“You know you have one when the crash of a computer you’ve never heard of stops you from getting any work done.”
Facts (II)

Spatially distributed computing systems are ubiquitous nowadays:

- The Internet
- PCs connected via a LAN
- Networked embedded systems
- Shared-memory multiprocessor machines
- Systems-on-Chip

Increasing dependence of our society on correct operation of such systems

Reasoning about distributed systems is important
Characteristics of DS

- Multiple processes, on multiple processors, characterized by
  - asynchronous concurrent computations
  - local state
  - work on common goal \(\Rightarrow\) need to access (part of) global state
- Processes can only communicate with each other, via
  - message passing
  - shared memory
- Processes may fail without immediate recognition by the rest of the system
Distributed Systems Dilemma

In theory, distributed systems offer

- increased reliability/availability
- increased performance
- scalability
In theory, distributed systems offer
- increased reliability/availability
- increased performance
- scalability

In practice, building up distributed systems is notoriously difficult:
- Heterogeneity of HW & SW
- Lacking adherence to standards
- System size and complexity
- Fundamental problems!
Fundamental Problems

Building distributed systems is difficult due to the processes’ uncertainty about the global system state, as caused by:
- different/unknown processor speeds
- varying/unknown communication delays
- partial failures
- local interaction with the environment
Fundamental Problems

Building distributed systems is difficult due to the processes’ uncertainty about the global system state, as caused by:

- different/unknown processor speeds
- varying/unknown communication delays
- partial failures
- local interaction with the environment

Need distributed algorithms for pivotal services like leader election, mutual exclusion and consensus that

- can live with this uncertainty
- can be proved to work correctly
Course Overview
Paradigm (I)

Attack distributed algorithms from a theoretical perspective:
- Identify and abstract fundamental problems
- State problems carefully
- State system model and failure model carefully
- Design algorithms to solve those problems
- Prove correctness of those algorithms under the system and failure model
- Analyze time/space/message complexity
- Prove impossibility results and lower bounds
**Paradigm (II)**

**Granted:** Theoretical reasoning cannot replace (but only complement) engineering:
- Theory often deals with high-level specifications, rather than fully implemented algorithms
- Real-world requirements often difficult/impossible to model

**But:**
- Careful specification clarify intent
- Mathematical proofs increase confidence in correctness of implemented algorithms
- Good abstractions can be re-used in multiple contexts
- Inherent limitations are revealed
Course Content (I)

What you will NOT hear about:

- CSP, CCS and other logic-based and algebraic specifications
- Formal verification
- Complex distributed algorithms
- Distributed programming

Some of those topics are covered by other basic courses, like

- Formale Methoden der Informatik
- Computer-Aided Verification
Course Content (II)

What you will hear about:

- Communicating state machines
- Computational models
- Failure models
- Correctness proofs and performance analysis of simple distributed algorithms
- Impossibility results and lower bound proofs
Some Background Info

- Actual course content defined by my slides
  [Link to slides](http://ti.tuwien.ac.at/ecs/teaching/courses/valg/misc/valg.pdf)
  - Slides essentially text-only
  - Lectures primarily use blackboard drawings
  - **Recommended:** Print slides and add info during lectures

- Prerequisites: Analysis of algorithms and basic discrete mathematics — will be checked in first quiz!
What to Do?

- **5 Homework assignments (45%)**, to be carefully, rigorously and completely worked out *by yourself* (using \texttt{\LaTeX})
  - first version, also presented on blackboard in class
  - doubly-blind review of your colleagues’ first versions
  - final version, incorporating the feedback
  - doubly-blind shephering review of final versions

- **5 Quizzes (40%)**, 20–25 min. each, covering both advance reading and past material (including prerequisites) of current chapter

- **Final exam (15%)**, 40-50 min., a quiz covering the whole content of the course

- Participation in discussions in class
Course Admission

Participation in VALG needs admission:

- Limited class size
- Students with insufficient skills in devising basic mathematical proofs almost always fail to pass the course, possibly despite large efforts

Admission based on the following criterions:

- Performance in the first and second quiz: You need to be positive in at least one of those
- Master students [VALG mandatory usually preferred]
- Bachelor students also eligible if class size allows [if needed, I will defer issuing the certificate upon your request]
General Rules

- Passing requires $\geq 60\%$ of the achievable maximum
- Presence in class is mandatory
- Advance reading of textbook required — will be checked in quizzes!
- Graduate courses like VALG adhere to “pull-based” learning — you have to obtain all the information you need for doing your work
- All work must be done on your own and written up in your own words; all sources of information must be properly referenced (except textbook and slides)
- Enroll via myTI only after you satisfy the admission criterions
Expected Achievements

Having passed this course, you should

- have improved formal/mathematical skills in general (major rationale of most Master TI basic courses)
- have seen another example of “computer science ⊆ programming”
- have a first basis for own work in this area

Regarding this course . . .

- Some success stories of former VALG participants:
  [http://ti.tuwien.ac.at/ecs/teaching/courses/valg/misc/success_st](http://ti.tuwien.ac.at/ecs/teaching/courses/valg/misc/success_st)
- Against rumors about “too much effort”:
  [http://ti.tuwien.ac.at/ecs/teaching/courses/valg/misc/Benchmark_VALG.pdf](http://ti.tuwien.ac.at/ecs/teaching/courses/valg/misc/Benchmark_VALG.pdf)
Follow-up Courses (I)

Problems in Distributed Computing 182.703
- Overview lectures of advanced topics in distributed algorithms
- Common reading of papers and book chapters
- Student’s lectures

Building Reliable Distributed Systems 182.704
- Lectures on basic (communication) services
- Simple correctness proofs and performance analyses
- Implementation of distributed algorithms
Follow-up Courses (II)

Embedded Systems Engineering Scientific Project 182.706

- First steps in own scientific work in a (self-)assigned distributed algorithms project
- Guided writing of a small scientific paper + presentation
- Learning about the scientific community in the field

Master thesis, Dissertation

- Typically funded positions (Diplomarbeitsstipendium, PhD research position)
- Learn about top-level international research
- Try out your (first) own steps in real scientific research
Questions ?
Formal Model (Message Passing)
Network Model

Communications graph, made up of

- $n$ processors $p_0, \ldots, p_{n-1}$
- processors communicate by sending messages $m \in \mathcal{M}$
- up to $n(n-1)/2$ bidirectional point-to-point links

Restrictions in this course:

- Reliable links
- Usually fully-connected network
State Machines Modeling Processors (I)

Processor $p_i$ modeled as state machine $P_i = (Q_i, \Phi_i, I_i, T_i)$

- state set $Q_i$ (possibly infinite)
- non-empty set of initial states $I_i \subseteq Q_i$
- non-empty set of terminal states $T_i \subseteq Q_i$ (possibly $T_i = Q_i$)
- transition function $\Phi_i \subseteq Q_i \times Q_i$ (successor relation)
Processor $p_i$ modeled as state machine $P_i = (Q_i, \Phi_i, I_i, T_i)$

- state set $Q_i$ (possibly infinite)
- non-empty set of initial states $I_i \subseteq Q_i$
- non-empty set of terminal states $T_i \subseteq Q_i$ (possibly $T_i = Q_i$)
- transition function $\Phi_i \subseteq Q_i \times Q_i$ (successor relation)

Transition $(q_i, q'_i) \in \Phi_i$, also termed step, denoted $(q_i, \phi_i, q'_i)$,

- happens upon occurrence of event $\phi_i$ (we will use $\phi_i = i$, denoting “$p_i$ makes a step”)
- when in state $q_i$ (enabling condition), step moves $p_i$ to state $q'_i$
State transitions executed
- atomically (at once, i.e., non-interruptable)
- in zero time, but:
- model non-zero execution times via time between successive steps
State transitions executed

atomically (at once, i.e., non-interruptable)

in zero time, but:

model non-zero execution times via time between successive steps

Depending on transition relation:

- **Deterministic** state machines (this course): If \((q_i, \phi, q_i')\) and \((q_i, \phi, q''_i)\) are valid state-transitions, then \(q_i' = q''_i\) (⇒ event and step essentially equivalent)

- Non-deterministic [randomized] state machines: Multiple \(q'_i\) [according to some probability distribution]
Our Processor States

State $Q_i$ partitioned in $Q_i = L_i \times inbuf_i[*] \times outbuf_i[*]$

- “Ordinary” internal state $L_i$ (local memory, registers)
- Received messages: $inbuf_i[*] = \bigcup_{\ell=0}^{n-1} inbuf_i[\ell]$
- Messages in transit: $outbuf_i[*] = \bigcup_{\ell=0}^{n-1} outbuf_i[\ell]$
- Transition enabling only depends on accessible state $S_i = L_i \cup inbuf_i[*]$, i.e., $p_i$ knows only $S_i$

Transition $(q_i, \phi, q'_i)$ at $p_i$ involves

- removing messages from $inbuf_i[*]$ and/or
- changing local state and/or
- moving messages to $outbuf_i[*]$
Distributed State Machine (I)

Build global state machine \( S = (C, \Phi, I, T) \), by composing all \( p_i \)'s state machines

- Global states: Configurations \( C, I, T \)
- Global transitions: \( \Phi \subseteq C \times C \)

Configurations \( C = (q_0, q_1, \ldots, q_{n-1}) \in Q_0 \times \cdots \times Q_{n-1} \)

- vector of all \( p_i \)'s local states [including \( inbuf_i[\ast] \) and \( outbuf_i[\ast] \)]
- only known to omiscent obverver
- initial and terminal configurations composed from \( I_i \) and \( T_i \), respectively
Distributed State Machine (II)

Message delivery relation $\Delta \subseteq C \times C$

- move a non-empty subset of messages in $outbuf^*[\ast]$ to $inbuf^*[\ast]$
- message delivery usually non-deterministic
Distributed State Machine (II)

Message delivery relation $\Delta \subseteq C \times C$

- move a non-empty subset of messages in $outbuf_*[*]$ to $inbuf_*[*]$
- message delivery usually non-deterministic

Transition $(q, q') \in \Phi = \Delta \cup \bigcup_{i=0}^{n-1} \Phi_i$ of global state machine:

- $\Phi$ is union of all $p_i$’s transition relations $\Phi_i$ and message delivery relation $\Delta$
- global state transition = either some local state transition or delivery of messages
Distributed Algorithm vs. Adversary

View execution of global state machine $S$ as interplay between algorithm and adversary

- Algorithm (via $\Phi_i$) determines what to do in a step
- Adversary determines
  - Message scheduling: order and times of deliveries
  - Processor scheduling: order and times of events
  - Failures: type and times of failures

Power of adversary constrained by

- System model (synchronous, asynchronous)
- Fairness conditions (message & processor sched.)
- Failure model
Transition Function of Textbook (I)

Only two simple types of events:

- **Deliver event** $\phi_j = \text{del}(i, j, m)$ at $p_j$: For some single message $m \in \text{outbuf}_i[j]$, move $m$ from $\text{outbuf}_i[j]$ to $\text{inbuf}_j[i]$

- **Computation event** $\phi_i = \text{comp}(i)$ at $p_i$: Move processor $p_i$ from $q_i$ to $q'_i$ [with $\text{inbuf}_i[*] = \emptyset$], and add zero or more messages $m \in M_\ell$ to $\text{outbuf}_i[\ell]$, for every $\ell$

Note:

- Any $\text{comp}(i)$ must always be **applicable**, i.e., there must always be an enabled transition at any $p_i$

- Message delivery need not be FIFO

- Can also model hardware broadcast communication
Executions (I)

Execution segment \( C^0, \phi^1, C^1, \ldots, \phi^m, C^m \) of system \( S \):

- Finite sequence of configurations alternating with events, ending in a configuration

- Event \( \phi^k \) is either:
  - \( \phi^k = \text{comp}(i) \), identifying the processor \( p_i \) that performs the step \( (C^{k-1}, \phi_i^k, C^k) \), or
  - \( \phi^k = \text{del}(j, i, m) \), identifying the delivery of message \( m \) at processor \( p_i \)

- at most processor \( p_i \) (and \( \text{outbuf}_j[i] \) of \( p_j \), in case of a delivery event) change state when \( S \) moves from \( C^{k-1} \) to \( C^k \)
Executions (II)

Executions are infinite execution segments \( C^0, \phi^1, C^1, \ldots \)

- starting with an initial configuration
  \[
  C^0 = (q^0_0, \ldots, q^0_{n-1}) \in I = I_1 \times \cdots \times I_n
  \]

- eventually reaching (and remaining within) terminal configurations, i.e., \( \exists K \geq 0 \) such that, for all \( k \geq K \)
  \[
  C^k = (q^k_0, \ldots, q^k_{n-1}) \in T = T_1 \times \cdots \times T_n
  \]

- a configuration \( C \) occurring in some valid execution is called reachable configuration

For an execution segment \( C^0, \phi^1, C^1, \ldots, \phi^m, C^m \), the schedule \( \sigma = \phi^1 \phi^2 \ldots \phi^m \) is

- the totally ordered sequence of events

- successive events possibly occur at the same time
Deterministic processors: Schedule $\sigma$ + initial config. $C^0$ uniquely determine execution $\text{exec}(C^0, \sigma) = C^0, \phi^1, C^1, \ldots$

- For $\phi^k = \text{comp}(i)$, this holds only since $\text{inbuf}_i[\ast]' = \emptyset$ [otherwise, we would not know which message(s) from multiple ones are actually processed in the step]

- Given some event $\phi$ applicable in configuration $C$, we write $C' = \phi(C)$ if $(C, \phi, C')$ is a valid step,

- Given $\sigma = \phi^1 \phi^2 \ldots \phi^m$, the corresponding execution segment is $\text{exec}(C^0, \sigma) = C^0, \phi^1, C^1, \phi^2, C^2, \ldots \phi^m, C^m$, where $C^k = \phi^k(C^{k-1})$ for $1 \leq k \leq m$
Transition Function of Textbook

Summary of implications of $\text{inbuf}_i[*] = \emptyset$ after $\text{comp}(i)$:

- Is key for 1:1 correspondence of executions and schedules,
- despite local computation events consisting of processor id only
- dropping it would require incorporating the processed message(s) in an event [e.g., FLP model]

Processing order of messages may sometimes differ from delivery order (determined by algorithm in case of multiple delivered messages)

Makes definition of end-to-end message delay independent of “receptiveness” [i.e. readiness for processing a message] of algorithm
Uniqueness of Executions?

Starting from same $C^0$, there are usually different possible schedules/executions:

- In a given configuration $C$, transitions of several $p_i$’s could be enabled
- Multiple events $\phi$, $\phi'$ applicable in $C$

$\Rightarrow$ successor configuration $C'$ could be either $\phi(C)$ or $\phi'(C)$, depending on which event comes first [depends on scheduling by adversary]
Uniqueness of Executions?

Starting from same $C^0$, there are usually different possible schedules/executions:

- In a given configuration $C$, transitions of several $p_i$’s could be enabled
- Multiple events $\phi$, $\phi'$ applicable in $C$
  $\Rightarrow$ successor configuration $C'$ could be either $\phi(C)$ or $\phi'(C)$, depending on which event comes first [depends on scheduling by adversary]

Question: When can events be reordered in a schedule?
Independence of Events

**Theorem 35.** Let $\phi_i$ and $\phi_j$ be two events at different processors $p_i \neq p_j$ that are both applicable to configuration $C$. Then,

- $\phi_i$ is applicable to $\phi_j(C')$
- $\phi_j$ is applicable to $\phi_i(C')$
- and the events commute $\phi_i(\phi_j(C')) = \phi_j(\phi_i(C'))$
Independence of Events

**Theorem 35.** Let $\phi_i$ and $\phi_j$ be two events at different processors $p_i \neq p_j$ that are both applicable to configuration $C$. Then,

- $\phi_i$ is applicable to $\phi_j(C')$
- $\phi_j$ is applicable to $\phi_i(C')$
- and the events commute $\phi_i(\phi_j(C')) = \phi_j(\phi_i(C'))$

**Proof.** Case analysis:

- $\phi_i = \text{comp}(i)$ and $\phi_j = \text{comp}(j)$: Affects states of $p_i \neq p_j$ independently $\Rightarrow$ events independent
- $\phi_i = \text{comp}(i)$ and $\phi_j = \text{del}(x, j, m)$: Since $\phi_j$ applicable in $C'$, either $x \neq i$ or $m' \neq m$ for $m'$ sent in $\phi_i$ $\Rightarrow$ events independent
- $\phi_i = \text{del}(x, i, m)$ and $\phi_j = \text{del}(y, j, m')$ for any $x, y$: Affects $\text{inbuf}_i[x]$ and $\text{inbuf}_j[y]$ (and $\text{outbuf}_x[i]$ and $\text{outbuf}_y[j]$) only $\Rightarrow$ events independent
Internal Causality

Events $\phi, \phi'$ only dependent, that is, $\phi(\phi'(C)) \neq \phi'(\phi(C))$, if either

- they occur at the same processor $p_i$ and
- $\phi = \text{comp}(i)$ and $\phi' = \text{del}(j, i, m)$, since $\text{comp}(i)$ must process $m$ in $\phi(\phi'(C))$ but cannot in $\phi'(\phi(C))$
- $[\phi = \text{comp}(i)$ and $\phi' = \text{comp}(i)$ (due to our simple events, they are the same, hence commute . . . )]

the step corresponding to $\phi = \text{comp}(i)$ puts $m$ into $\text{outbuf}_{i}[j]$ and $\phi' = \text{del}(i, j, m)$
Internal Causality

Events $\phi, \phi'$ only dependent, that is, $\phi(\phi'(C)) \neq \phi'(\phi(C))$, if either

- they occur at the same processor $p_i$ and
- $\phi = \text{comp}(i)$ and $\phi' = \text{del}(j, i, m)$, since $\text{comp}(i)$ must process $m$ in $\phi(\phi'(C))$ but cannot in $\phi'(\phi(C))$
- $[\phi = \text{comp}(i)$ and $\phi' = \text{comp}(i)$ (due to our simple events, they are the same, hence commute ...)]

the step corresponding to $\phi = \text{comp}(i)$ puts $m$ into $\text{outbuf}_i[j]$ and $\phi' = \text{del}(i, j, m)$

This induces the system’s internal causality relation (Lamport’s happened before relation)

- Depicted via an execution’s space-time diagram
- Dealt with in detail in Chapter “Causality and Time”
System Models
Asynchronous Systems

Consider distributed systems with

1. unbounded (or unknown) but finite transmission delays
2. no real-time clocks
3. no execution speed bounds [but fair processor scheduling]
Asynchronous Systems

Consider distributed systems with

1. unbounded (or unknown) but finite transmission delays
2. no real-time clocks
3. no execution speed bounds [but fair processor scheduling]

Features:

+ Strongest adversary, covering also unanticipated processor workloads, network congestion, etc.

− Simple semantics (“time-free algorithms”), easy to port
  − Difficult to analyze and prove correct
  − **Impossibilities**: Not all distributed computing problems have asynchronous solutions
Admissible Asynchronous Executions

Executions that also satisfy admissibility conditions:

- Every (correct) processor takes infinitely many steps
- Every message in transit is eventually delivered

Admissibility usually ensured by fairness conditions:

- Restricts adversary w.r.t. processor and message scheduling
- **Weak fairness**: Every continuously applicable event eventually occurs (⇒ infinitely many steps of every $p_i$)
- [**Strong fairness**: Every infinitely often applicable event eventually occurs (only relevant for non-deterministic processors, multiple processes etc.)]
Constrain execution of $S$ to lock-step rounds:
- Execution proceeds in a sequence of rounds $k \geq 1$
- All processors take computing steps simultaneously
Synchronous Systems (I)

Constrain execution of $S$ to **lock-step rounds**:  
- Execution proceeds in a sequence of rounds $k \geq 1$  
- All processors take computing steps simultaneously

In every round $k \geq 1$:

1. At the beginning, every $p_i$ simultaneously sends its round-$k$ message(s) to (a subset of) the processors
2. All round-$k$ messages in transit are delivered
3. At the end, every $p_i$ simultaneously performs a single $\text{comp}(i)$ [and sends the messages for round $k + 1$]

Initially: $\text{outbuf}_i[*]$ hold $p_i$’s round-1 messages
Synchronous Executions

Execution segment $C^0, \phi^1, C^1, \phi^2, C^2, \ldots, \phi^m, C^m$

- 
  Finite sequence of configurations alternating with round events, ending in a configuration

- 
  Round event $\phi^k$ represents all round $k$ deliver + $\text{comp}(0), \ldots, \text{comp}(n-1)$ at all processors

- 
  $C^0 \in \mathcal{I}$ is initial configuration, with $\text{outbuf}_i[\ast]$ holding $p_i$'s round-1 messages [often assume “virtual” round 0 ending in $C^0$ for convenience]

- 
  $C^k, 0(1) \leq k \leq m$, is configuration at the end of round $k$
Synchronous Executions

Execution segment $C^0, \phi^1, C^1, \phi^2, C^2, \ldots, \phi^m, C^m$

- Finite sequence of configurations alternating with round events, ending in a configuration
- Round event $\phi^k$ represents all round $k$ deliver + $\text{comp}(0), \ldots, \text{comp}(n-1)$ at all processors
- $C^0 \in \mathcal{I}$ is initial configuration, with $\text{outbuf}_i[\ast]$ holding $p_i$'s round-1 messages [often assume “virtual” round 0 ending in $C^0$ for convenience]
- $C^k, 0(1) \leq k \leq m$, is configuration at the end of round $k$

Admissible synchronous executions:

- Infinite execution $C^0, \phi^1, C^1, \phi^2, C^2, \ldots$

$\Rightarrow$ Every $p_i$ takes infinitely many rounds (hence steps)
Synchronous Systems (II)

Lock-step round model
- very convenient for analysis
- too far away from reality to be useful in practice?
Synchronous Systems (II)

Lock-step round model

- very convenient for analysis
- too far away from reality to be useful in practice?

No: Lockstep rounds can be simulated in synchronous systems:

1. Known upper bound $\delta$ on message transmission delays

2. Availability of real-time clock $C_i$ with bounded drift $\rho'$ at every processor $p_i$:

\[(t_1 - t_0)(1 - \rho') \leq C_i(t_1) - C_i(t_0) \leq (t_1 - t_0)(1 + \rho')\]

3. Known lower and upper bound on execution times (time between successive steps)
Synchronous systems allow clocks $C_i$ (and hence inverse clocks $c_i = C_i^{-1}$) to be kept approximately synchronized:

- $|c_p(T) - c_q(T)| \leq \pi$
- $(T_1 - T_0)(1 - \rho) \leq c_p(T_1) - c_p(T_0) \leq (T_1 - T_0)(1 + \rho)$
How to Simulate Lockstep Rounds? (I)

Synchronous systems allow clocks $C_i$ (and hence inverse clocks $c_i = C_i^{-1}$) to be kept approximately synchronized:

- $|c_p(T) - c_q(T)| \leq \pi$
- $(T_1 - T_0)(1 - \rho) \leq c_p(T_1) - c_p(T_0) \leq (T_1 - T_0)(1 + \rho)$

Use local clocks to almost simultaneously start round $k$ at every processor:

- Start round-$k$ at $p_i$ when local clock $C_i$ reads $kR$
- Choose $R \geq (\pi + \delta)/(1 - \rho)$
How to Simulate Lockstep Rounds? (II)

Claim: Every round-$k$ message is received before the first processor starts round $k + 1$: 

\[ t_k^{p_1} \leq \pi \leq \delta \leq t_{k+1}^{p_3} \]
How to Simulate Lockstep Rounds? (III)

Proof:

- Let $p$ be processor that is the first to start round $k + 1$, and $q$ be the last to start round $k$

- Need to show $t^{k+1}_p \geq t^k_q + \delta$

- Follows from adding $t^k_p$ on both sides of

\[
t^{k+1}_p - t^k_p \geq R(1 - \rho) \geq \pi + \delta \geq t^k_q - t^k_p + \delta
\]

$\Rightarrow$ In synchronous model, this provides

- lockstep rounds w.r.t. clock time

- approximately lockstep rounds w.r.t. real-time
Analysis of Distributed Algorithms
Safety and Liveness Properties

Safety properties: “Nothing bad happened yet”

- Violation shows up in a finite prefix of an execution
- Example mutual exclusion: Violated if, in any reachable configuration, two processes are in the critical section
- Proofs typically use induction
Safety and Liveness Properties

Safety properties: “Nothing bad happened yet”
- Violation shows up in a finite prefix of an execution
- Example mutual exclusion: Violated if, in any reachable configuration, two processes are in the critical section
- Proofs typically use induction

Liveness properties: “Something good eventually happens”
- Violation (often) shows up in infinite executions only
- Example leader election: The system eventually elects a leader
- Proofs typically use norm functions on well-founded sets
Assertion-based Safety and Liveness

Focusses on properties fulfilled in reachable configurations of admissible executions of an algorithm

Assertions:

- Unary relation on configurations
- Predicate $P(C)$ that delivers true or false when applied to $C$

Consider sequence of configurations reached in any execution of $S$:

- **Safety property:** Assertion that holds in every reachable configuration ($\Rightarrow$ correctness)
- **Liveness property:** Assertion that holds (perpetually) after reaching some configuration ($\Rightarrow$ progress)
Invariants

For assertions $A, B$, we write $\{A\} \rightarrow \{B\}$ if, for each configuration $C$ and each step $(C, \phi, C')$,

\[ A(C') \Rightarrow B(C'), \] i.e.,

\[ \text{if } A \text{ holds before a transition, then } B \text{ holds afterwards} \]

Assertion $A$ is an invariant if

\[ A(C') \text{ for all } C \in \mathcal{I}, \] and

\[ \{A\} \rightarrow \{A\} \]
Invariants

For assertions $A$, $B$, we write $\{A\} \rightarrow \{B\}$ if, for each configuration $C$ and each step $(C, \phi, C')$,

- $A(C') \Rightarrow B(C')$, i.e.,
- if $A$ holds before a transition, then $B$ holds afterwards.

Assertion $A$ is an invariant if

- $A(C)$ for all $C \in \mathcal{I}$, and
- $\{A\} \rightarrow \{A\}$

**Theorem 49.** If $A$ is an invariant of system $S$, then $A$ holds for each configuration of each execution of $S$. [Proof by simple induction]

**Corollary 49.** Let $B$ be an invariant of $S$ and assume $B(C') \Rightarrow A(C')$ (for each reachable $C'$). Then $A$ holds in each configuration of each execution of $S$. 
Partial order ["strikte (= irreflexive) Halbordnung"]

- Set $W$
- Partial order $<$ of elements of $W$:
  - Irreflexivity $w \not< w$
  - Transitivity $(x < y) \land (y < z) \Rightarrow x < z$
  - Asymmetry: $x < y \rightarrow y \not< x$

A partial order $(W, <)$ is well-founded if

- every non-empty subset $X \subseteq W$ has a minimal element $m \in X$, i.e., $\nexists x \in X$ with $x < m$
- Example: Tuples of natural numbers $(n_k, n_{k-1}, \ldots, n_1)$, $k \geq 1$, $n_i \geq 0$, with lexical order
Norm Functions

Equivalent definition of a well-founded partial order \((W, <)\):

- There is no infinite decreasing sequence \(w_1 > w_2 > \cdots\), \(w_i \in W\)

Let a system \(S\) and assertion \(A\) be given. A function \(f : C \rightarrow W\) is a norm function if,

- \(f(C') > f(C')\) or \(A(C)\), for each transition \((C, \phi, C')\)
Norm Functions

Equivalent definition of a well-founded partial order \((W, <)\):

- There is no infinite decreasing sequence \(w_1 > w_2 > \cdots\), \(w_i \in W\)

Let a system \(S\) and assertion \(A\) be given. A function \(f : C \rightarrow W\) is a norm function if,

- \(f(C') > f(C')\) or \(A(C)\), for each transition \((C, \phi, C')\)
- \(f(C_i) > f(C_{i+1})\) or \(A(C_i)\), for every \(i \geq 1\), for some infinite sequence of “interesting” configurations \(C_1, C_2, \ldots\) occurring in every execution
- Definition of “interesting” typically depends on \(A\)
Proving Liveness

**Theorem 52.** If system $S$ without terminal states ($\mathcal{T} = \emptyset$) has a norm function $f$, then $A$ is true in some configuration in each execution of $S$.

**Proof.** Let $E$ be longest execution prefix where $A$ never holds. The existence of $f$ implies that $E$ is finite, so $A$ must hold in the configuration following $E$. \qed
Proving Liveness

**Theorem 52.** If system $S$ without terminal states ($\mathcal{T} = \emptyset$) has a norm function $f$, then $A$ is true in some configuration in each execution of $S$.

**Proof.** Let $E$ be longest execution prefix where $A$ never holds. The existence of $f$ implies that $E$ is finite, so $A$ must hold in the configuration following $E$. □

For systems $S$ with terminal states $\mathcal{T}$,
- define assertion $T(C) = \text{true}$ iff $C \in \mathcal{T}$
- $S$ terminates properly if $T \Rightarrow A$

**Theorem 52.** If system $S$ terminates properly and a norm function $f$ exists, then $A$ is true in some configuration in each execution of $S$.

**Proof.** If some admissible execution of $S$ is finite, $A$ holds by proper termination. In an infinite admissible execution, the previous theorem applies. □
Performance Analysis (I)

Consider terminating algorithms

- every processor reaches terminal configuration
- no messages in transit eventually

Message complexity:

- Maximum number of messages sent in any execution
- Maximum number of bits sent in any execution

Space complexity:

- Maximum number of bits in any processor’s accessible state in any execution
Performance Analysis (II)

Consider timed executions

- every event associated with occurrence real-time
- timestamps of every $p_i$’s events $\phi^k_i$ strictly monotonically increasing (without bound)

End-to-end delay $\tau$ of a message $m$ sent by $p_i$ to $p_j$

- time from $\text{comp}(i)$ sending $m$ to $\text{comp}(j)$ processing $m$
  [recall that processing happens in first comp after del]
- incorporates both computation and communication

Time complexity:

- Sync: # rounds until last processor in terminal state
- Async: Max. termination time of last processor for $\tau \leq 1$
Mathematical definition $\Omega(.)$

- $f(n) = \Omega(g(n))$ if there are constants $C, n_0$ such that $|f(n)| \geq C|g(n)|$ for $n \geq n_0$

- Application to performance measures of distributed algorithms?
A Note on Lower Bounds (I)

Mathematical definition $\Omega(.)$

- $f(n) = \Omega(g(n))$ if there are constants $C, n_0$ such that $|f(n)| \geq C|g(n)|$ for $n \geq n_0$

Application to performance measures of distributed algorithms?

Two possibilities for lower bounds on complexities:

- **Worst case**: For algorithm $\mathcal{A}$, there is some execution $E$ where $\mathcal{A}$ has complexity $C^{wc}(\mathcal{A}) = \Omega(f(n, \ldots))$

- **Best case**: For algorithm $\mathcal{A}$, the complexity $C^{bc}(\mathcal{A})$ of $\mathcal{A}$ for any execution $E$ is $\Omega(f(n, \ldots))$
A Note on Lower Bounds (II)

We will focus on the worst case lower bound for problem $P$:

- **Lower bound**: $\inf_{A} C^{wc}(A) = \Omega(f(n, \ldots))$
- **Tightness**: $\exists A$ that solves $P$ with $C^{wc}(A) = O(f(n, \ldots))$
Basic Broadcasting Algorithms
Broadcast on a Spanning Tree

Consider distinguished processor \( p_r \) that

- has some message \( M \) it wants to broadcast
- is root of a given spanning tree \( T \) (i.e., every \( p_i \) knows its parent and children)

Simple algorithm

- \( p_r \) sends \( M \) to all its children in \( T \)
- every \( p_i \) that receives \( M \) for the first time from its parent sends \( M \) to all its children in \( T \)
- processors terminate after having sent \( M \)
Pseudo-Code Algorithm 1 (Asynchronous)

1. Initially $M$ is in transit from $p_r$ to all its children

2. **Code for $p_r$:**
   3. on receiving no message: // first comp($p_r$) event
      4. terminate

5. **Code for $p_i$, $0 \leq i \leq n - 1$, $i \neq r$:**
   6. on receiving $M$ from parent:
      7. send $M$ to all children
      8. terminate
**State Machine Description Algorithm 1 (I)**

Variables $\in L_i$ of processor $p_i$:

- $parent_i$ holds processor index (or nil in case of $p_r$)
- $children_i$ holds set of processor indices
- $term_i$ indicates whether $p_i$ has terminated

Initial state:

- $\forall i : parent_i$ and $children_i$ form spanning tree rooted at $p_r$
- $\forall i : term_i = false$
- $\forall j \in children_r : outbuf_r[j] = M$ and otherwise
  - $\forall i \neq r : outbuf_i[*] = \emptyset$
- $\forall i : inbuf_i[*] = \emptyset$
State Machine Description Algorithm 1 (II)

Processor $p_r$ (root):

- $\forall q_r \in Q_r$ with $\text{term}_r = \text{false}$: $(q_r, \phi_r, q'_r) \in \Phi_r$, where $q'_r = q_r$, except
- $\text{term'}_r := \text{true}, \text{inbuf}_r[*]' := \emptyset$

Processor $p_i, i \neq r$:

- $\forall q_i \in Q_i$ with $\text{term}_i = \text{false}$ and $\text{inbuf}_i[\text{parent}_i] = X \neq \emptyset$: $(q_i, \phi_i, q'_i) \in \Phi_i$, where $q'_i = q_i$, except
- $\forall j \in \text{children}_i$: $\text{outbuf}_i[j]' := \text{outbuf}_i[j] \cup X$
- $\text{term'}_i := \text{true}, \text{inbuf}_i[*]' := \emptyset$

- For all other $q_i \in Q_i$ (idle transition): $(q_i, \phi_i, q'_i) \in \Phi_i$, where $q'_i = q_i$ except $\text{inbuf}_i[*]' := \emptyset$
General State Machine Descriptions (I)

Complex algorithms involve multiple state transitions
\((q_i, \phi_i, q'_i) \in \Phi_i\)

- Deterministic transitions \(\Rightarrow\) Specification of \(q_i\) ("guard") of different transitions must be disjoint!

- It is **not** allowed to use conditional statements ("if ... then") when describing how \(q'_i\) looks like!

- **But:** One may introduce new variables (like \(X\)) in the description of \(q_i\) that can be used in the description of \(q'_i\):
  - Shorthand for multiple transitions, one for each possible value of \(X\)
  - Type of such variables usually clear from context (Algorithm 1: \(X = \{M\}, \text{fixed}\)
General State Machine Descriptions (II)

Static vs. dynamic behavior of algorithms:

- Being conservative by adding “dead” transitions (never executed in any run) does not harm

- Safe removal of dead transitions requires dynamic analysis
General State Machine Descriptions (II)

Static vs. dynamic behavior of algorithms:

- Being conservative by adding “dead” transitions (never executed in any run) does not harm.
- Safe removal of dead transitions requires dynamic analysis.

Complication due to requirement of $inbuf_i[\ast]' := \emptyset$:

- Occurs if multiple messages $X = \{m_1, m_2, \ldots\}$ may be present in $inbuf_i[\ast]$. 
  - All to be processed within every single transition.
  - Implication: need one dedicated transition for every possible $X$.
- Sometimes cumbersome to write . . .
Possible alternative: Define, for every possible $m_{\ell} \in X$, an elementary transition $(q_i, \phi_{i\ell}, q'_i)$:

- involves guard $m_{\ell} \in X$ in the description of $q_i$
- applies only those changes to $q'_i$ that result from processing $m_{\ell}$
- removes only $m_{\ell}$ from $inbuf_i[*]$

Define the actual transitions in $\Phi_i$ as compound transitions:

- CT is element of the transitive closure of the elementary transitions $\Rightarrow$ automatically implies $inbuf_i[*] = \emptyset$
- Ensure deterministic $\Phi_i$ by incorporating only one CT for a given $X$ $\Rightarrow$ fixed order of processing multiple messages
Pseudo-Code Algorithm 1 (Synchronous)

Code for processor $p_i$, $0 \leq i \leq n - 1$, including $p_r$:

1. $\text{term} := \text{false} /*$ Initialization */
2. $\text{msg} := \emptyset /*$ No message */
3. if $p_i = p_r$ then
4. \hspace{1em} $\text{msg} := M /*$ Root sends $M$ in round 1 */
5. \hspace{1em} $\text{term} := \text{true} /*$ can terminate loop 1 round later */
6. do forever /* Loop over rounds */
7. \hspace{1em} send $\text{msg}$ to all children /* $\text{msg} = \emptyset \Rightarrow$ do nothing */
8. \hspace{1em} /* receive all messages in the round */
9. \hspace{1em} if received $M$ from parent then
10. \hspace{2em} $\text{msg} := M$
11. \hspace{1em} $\text{term} := \text{true} /*$ can terminate loop 1 round later */
12. \hspace{1em} else $\text{msg} := \emptyset /*$ send $M$ at most once */
Analysis of Algorithm 1

A trivial induction on the level in the tree reveals that the algorithm works correctly in both
- synchronous systems
- asynchronous systems

Complexity:
- Message complexity is $n - 1$, since exactly one $M$ is sent over every edge in the spanning tree $T$.
- Show: Time complexity for spanning tree with depth $d$
  $[d = 1 \text{ for } n = 2 \text{ processors, for example}]$:
  - Synchronous: $d$ rounds [if leafs don’t send $M$]
  - Asynchronous: $\leq d$ termination time ($\tau = 1$)
**Proof Synchronous Case**

**Lemma 67.** *Every processor at distance* \( t \geq 1 \) *from* \( p_r \) *in* \( T \) *receives* \( M \) *in round* \( t \).

**Proof.** By induction:

- **Basis** \( t = 1 \): Since \( M \) is initially in transit, every child of \( p_r \) receives \( M \) in round 1.

- **Induction step:**
  - Assume that every \( p_j \) at distance \( t - 1 \geq 1 \) receives \( M \) in round \( t - 1 \).
  - Show that every \( p_i \) at distance \( t \) receives \( M \) in round \( t \):
    Applying induction hypothesis to parent \( p_j \) of \( p_i \) reveals that \( p_j \) receives \( M \) in round \( t - 1 \), and hence, by the code, sends \( M \) in round \( t \) \( \Rightarrow \) \( p_i \) receives \( M \) in round \( t \).
Lemma 68. Every processor at distance $t \geq 1$ from $p_r$ in $T$ receives $M$ by time $t$.

Proof. By induction:

- **Basis $t = 1$:** Since $M$ is initially in transit, every child of $p_r$ receives $M$ by time 1 since $\tau = 1$

- **Induction step:**
  - Assume that every $p_j$ at distance $t - 1 \geq 1$ receives $M$ by time $t - 1$
  - Show that every $p_i$ at distance $t$ receives $M$ by time $t$:
    Applying induction hypothesis to parent $p_j$ of $p_i$ reveals that $p_j$ sends $M$ by time $t - 1 \Rightarrow p_i$ receives and processes $M$ by time $t - 1 + \tau \leq t$
Broadcast via Flooding

Some processor $p_r$

- wants to broadcast message $M$
- without a given spanning tree rooted at $p_r$

Flooding algorithm:

- Processor $p_r$ sends $M$ to all its (direct) neighbors
- Every processor $p_i$ that receives $M$ from some $p_j$ for the very first time sends $M$ to all its neighbors $p_l \neq p_j$

Can be adapted to construct a spanning tree rooted at $p_r$
### Pseudo-Code Algorithm 2

1. **Code for processor** $p_i$, $0 \leq i \leq n - 1$, with neighbors $Nb_i$

2. **VAR**
   - $parent := \emptyset$
   - $children := \emptyset$
   - $term := false$

3. **Root** $p_r$ only (initial state):
   - $parent := NULL$

4. $M$ is initially in transit from $p_r$ to all its neighbors $Nb_r$

5. **on receiving** $M$ from neighbor $p_j$:
   - **if** $parent = \emptyset$ then
     - $parent := p_j$; send $\langle parent \rangle$ to $p_j$
     - **if** DONE then $term := true$ else send $M$ to $Nb \setminus \{p_j\}$
   - **else** send $\langle already \rangle$ to $p_j$

6. **on receiving** $m \in \{\langle parent \rangle, \langle already \rangle\}$ from neighbor $p_j$:
   - **if** $m = \langle parent \rangle$ then add $p_j$ to $children$
   - **else** add $p_j$ to $other$

7. **if** DONE then $term := true$ // must still answer $M$ msgs!

8. **Macro** $DONE := children \cup other \cup parent = Nb$
Correctness of Algorithm 2

We show that:

- The algorithm builds a parent/child relation $T$ (hopefully a spanning tree) that is “locally eventually consistent” (next lemma)
- Every processor eventually terminates, such that the parent/child relation $T$ is eventually
  - **locally consistent:**
    \[ \forall j, i \neq r : \text{parent}_i = p_j \iff p_i \in \text{children}_j \]
  - **static:** does not change any more
- The finally constructed graph $T$ is a spanning tree:
  - There is no cycle in $T$
  - Every $p_i$ is reachable from the root $p_r$ in $T$
Lemma 72. In every reachable configuration of an admissible execution $C^0, \phi^1, C^1, \ldots$, the parent/child relation is locally eventually consistent:

$$\forall j, i \neq r : \text{parent}_i = p_j \Leftrightarrow (p_i \in \text{children}_j) \lor (\langle \text{parent} \rangle \in \text{outbuf}_i[j] \cup \text{inbuf}_j[i])$$

Proof. Invariant induction on subsequent configurations. Let $LEC(C)$ be the assertion that configuration $C$ is locally eventually consistent.

- **Induction basis** $k = 0$: Initially, $LEC(C^0)$ holds trivially.

- **Induction step** $k - 1 \rightarrow k$: Assume $LEC(C^{k-1})$ holds for $k \geq 1$. We have the following exhaustive cases for the (augmented) event $\phi^k$ in step $(C^{k-1}, \phi^k, C^k)$ [we abbreviate e.g. $\text{parent}_i = \text{parent}_i(C^{k-1})$ and $\text{parent}_i' = \text{parent}_i(C^k)$]:
  - For all transitions that do not affect $LEC$, we obviously have $LEC(C^{k-1}) \Rightarrow LEC(C^k)$. 
Safety Proof of Algorithm 2 (II)

Proof. (cont.)

If \( \phi^k = \text{del}(i, j, m) \), for some message \( m \):
- If \( m = \langle \text{parent} \rangle \), it is just moved from \( \text{outbuf}_i[j] \) to \( \text{inbuf}_j[i]' \).
  Hence \( LEC(C^k) \) holds since \( LEC(C^{k-1}) \) held.
- For any other message \( m \), the statement is not affected at all.
Safety Proof of Algorithm 2 (II)

Proof. (cont.)

- If $\phi^k = \text{del}(i, j, m)$, for some message $m$:
  - If $m = \langle \text{parent} \rangle$, it is just moved from $\text{outbuf}_i[j]$ to $\text{inbuf}_j[i]'$. Hence $\text{LEC}(C^k)$ holds since $\text{LEC}(C^{k-1})$ held.
  - For any other message $m$, the statement is not affected at all.

- If $\phi^k = \text{comp}(i)$, $i \neq r$:
  - If $\text{parent}_i = \emptyset$ (Line 8), $\text{parent}_i'$ is set to $p_j$ and $\langle \text{parent} \rangle$ is put into $\text{outbuf}_i[j]'$. Hence, $\text{LEC}(C^k)$ holds.
  - If $\text{parent}_i \neq \emptyset$ (Line 10), $\text{LEC}(C^k)$ continues to hold.
Safety Proof of Algorithm 2 (II)

Proof. (cont.)

- If $\phi^k = \text{del}(i, j, m)$, for some message $m$:
  - If $m = \langle \text{parent} \rangle$, it is just moved from $\text{outbuf}_i[j]$ to $\text{inbuf}_j[i]'$. Hence $LEC(C^k)$ holds since $LEC(C^{k-1})$ held.
  - For any other message $m$, the statement is not affected at all.

- If $\phi^k = \text{comp}(i)$, $i \neq r$:
  - If $\text{parent}_i = \emptyset$ (Line 8), $\text{parent}_i'$ is set to $p_j$ and $\langle \text{parent} \rangle$ is put into $\text{outbuf}_i[j]'$. Hence, $LEC(C^k)$ holds.
  - If $\text{parent}_i \neq \emptyset$ (Line 10), $LEC(C^k)$ continues to hold.

- If $\phi^k = \text{comp}(j)$ puts $p_i$ into $\text{children}_j'$ (Line 12):
  - Happens only upon reception of $\langle \text{parent} \rangle$ from $p_i$
  \[ \Rightarrow \] Since $LEC(C^{k-1})$ holds, $\text{parent}_i = p_j$, hence $LEC(C^k)$ continues to hold.
Safety Proof of Algorithm 2 (II)

- Standard procedure for invariant proofs
- No need to explicitly deal with steps that process multiple delivered messages at once (above proof allows arbitrary composition)
Safety Proof of Algorithm 2 (II)

- Standard procedure for invariant proofs
- No need to explicitly deal with steps that process multiple delivered messages at once (above proof allows arbitrary composition)

Inspecting the code reveals additional simple properties:

(a) Every processor $p_i$ sets $parent_i$ and sends $M$ to all neighbors $\neq parent_i$ at most once [trivial induction with trivial indirect proof in step]

(b) Every processor $p_i$ that receives $M$ from $p_j$ replies with either $\langle parent \rangle$ or $\langle already \rangle$ [immediate from code]

(c) In every execution, the sets $children_i$ and $other_i$ can only increase [proof technique (a)]
Liveness Proof of Algorithm 2 (I)

Lemma 75. Every process $p_j$ eventually sets $parent_j \neq \emptyset$ and sends $M$ to all neighbors $\neq parent_i$ exactly once.

Proof. Induction on distance $k$ from $p_r$ in communications graph:

- Induction basis $k = 0$: The root $p_r$ evidently sets $parent_r = \text{NULL}$ in line 4 and broadcasts $M$.

- Induction step $k - 1 \rightarrow k$:
  - Assume that all $p_i$ at distance $k - 1 \geq 0$ set $parent_i \neq \emptyset$ in line 4 or 8, where $M$ is sent to all other neighbors.
  - Since the execution is admissible, a process $p_j$ at distance $k$ eventually receives $M$ from some $p_i$ and sets $parent_j := p_i$ in line 8, if it has not already done so.

- Since every process sets $parent_j \neq \emptyset$ and sends $M$ at most once by simple property (a), exactly once follows.
Liveness Proof of Algorithm 2 (II)

**Theorem 76.** Every processor $p_i$ eventually terminates and constructs a spanning tree $T$ rooted at $p_r$.

**Proof.** By previous lemma, every $p_i$ sets $parent_i$ and sends $M$ exactly once to all neighbors $\neq parent_i$. Hence:

- By simple property (b), they respond with $\langle parent \rangle$ or $\langle already \rangle$.
- Since the execution is admissible, every $\langle parent \rangle$ resp. $\langle already \rangle$ is eventually delivered and processed, which adds the sender processor to $children_i$ resp. $other_i$. By the simple property (c), $p_i$ will eventually execute line 14 or 9 and terminate.
- Recalling local eventually consistency, local consistency follows.
Liveness Proof of Algorithm 2 (II)

**Theorem 76.** Every processor $p_i$ eventually terminates and constructs a spanning tree $T$ rooted at $p_r$.

**Proof.** By previous lemma, every $p_i$ sets $parent_i$ and sends $M$ exactly once to all neighbors $\neq parent_i$. Hence:

- By simple property (b), they respond with $\langle parent \rangle$ or $\langle already \rangle$.
- Since the execution is admissible, every $\langle parent \rangle$ resp. $\langle already \rangle$ is eventually delivered and processed, which adds the sender processor to $children_i$ resp. $other_i$. By the simple property (c), $p_i$ will eventually execute line 14 or 9 and terminate.
- Recalling local eventually consistency, local consistency follows.

Must still show that the constructed graph $T$ is a spanning tree:

- There is no cycle in $T$
- Every $p_i$ is reachable from the root $p_r$ in $T$
Proof. (con’t)

Suppose there is a cycle \( p_{i_1}, p_{i_2}, \ldots, p_{i_{k+1}} \) with \( i_{k+1} = i_1 \):

- Let \( \phi_l = \text{comp}(i_l) \) where \( \text{parent}_{i_l} \) is set and \( M \) is sent by \( p_{i_l} \)
- Parent/child relation obviously requires \( \phi_l \rightarrow \phi_{l+1} \)
- \( \phi_1 \rightarrow \phi_2, \phi_2 \rightarrow \phi_3 \) and \( \phi_k \rightarrow \phi_{k+1} = \phi_1 \) reveals a cycle in the causality relation \( \Rightarrow \) Contradiction

Suppose \( p_i \) is not reachable from \( p_r \) in \( T \):

- Parent/child relation was shown to be locally consistent
- Up-stream path starting from \( p_i \) could hence either
  - lead to cycle \( \Rightarrow \) already shown to be impossible
  - lead to root \( p_r \) (which has \( \text{parent}_r = \text{NULL} \) \( \Rightarrow \) contradicts assumption that \( p_i \) is not reachable from \( p_r \)
Liveness Proofs via Induction? (I)

We said earlier that liveness proofs cannot be done via induction:

- Violations show up in infinite executions only
- Liveness proofs usually use norm functions
Liveness Proofs via Induction? (I)

We said earlier that liveness proofs cannot be done via induction:

- Violations show up in infinite executions only.
- Liveness proofs usually use norm functions.

So why does it work for Algorithm 2?

- Because we are dealing with bounded liveness here:
  - We know maximum end-to-end delay $\tau = 1$.
  - We know $n$, and hence the maximum diameter (= path length) of the communication graph.

- Liveness property termination becomes a safety property termination within time $X$!
Liveness Proofs via Induction? (II)

We could even define a suitable norm function: For any configuration $C$, let

- $w_i(C) = |parent_i| + |children_i| + |other_i|$ at processor $p_i$
- Note: $parent_r = NULL$ is treated like $\emptyset$
- $w_i(C) \leq n - 1$, since obviously [by the code]:
  - $p_j \neq p_i$ can appear in at most one of $parent_i$, $children_i$ or $other_i$
  - $p_j$ is added upon processing of (first) $M$, $\langle parent \rangle$ or $\langle already \rangle$
  - $p_i$ (and NULL in case of $p_i = p_r$) never occurs in any of those sets
Consider trivial well-founded partial order \((\mathbb{N}, <)\):

- Start from vectors \(w = (w_{n-1}, \ldots, w_0)\) with \(0 \leq w_i \leq n - 1\)
- Interprete \(w\) as \(n\)-digit base-\(n\) number \(w = \sum_{i=0}^{n-1} w_i n^i\), with usual meaning of \(<\)

Define \(f(C) = n^n - w(C) = n^n - \sum_{i=0}^{n-1} w_i n^i\)

- Any comp event of Algorithm 2 either
  - increases some \(w_i\) (upon processing of \(M, \langle parent\rangle\) or \(\langle already\rangle\)), or
  - does not change any \(w_i\) (and hence the parent/child/other relation)
Liveness Proofs via Induction? (IV)

\[ f(C') = n^n - w(C') = n^n - \sum_{i=0}^{n-1} w_i n^i \]

is a norm function:

- \( f(.) \) has a minimum \( \Rightarrow \) infinitely many "decreasing" comp events impossible
- all messages eventually delivered in admissible execution \( \Rightarrow \) infinitely many successive "non-increasing" comp events impossible

Once \( f(C') \) attains its minimum \( \Rightarrow T \) locally consistent.
Performance Analysis of Algorithm 2

The algorithm constructs a spanning tree $T$ both in
- synchronous systems
- asynchronous systems

Assume communications graph $G$ with
- $n \geq 2$ processors
- $n - 1 \leq m \leq \frac{n(n-1)}{2}$ links
- diameter $D = \max_{x,y \in G} d(x, y)$ where $d(x, y) = \min_{W(x,y) \in G} |W(x,y)|$
Theorem 83. In both synchronous and asynchronous systems, Algorithm 2 has message complexity $O(m)$

Proof. Algorithm 2 sends $M$

- twice (once in every direction) over every link $\notin T$
- once over every link $\in T$

$\Rightarrow$ sends total of $2m - (n - 1) \leq (n - 1)^2$ messages $M$

Every $M$ message is answered by either $\langle parent \rangle$ or $\langle already \rangle$

$\Rightarrow$ total of $4m - (2n - 2) \leq 2(n - 1)^2$ messages
Synchronous Systems (I)

**Theorem 84.** In every admissible execution in the synchronous model, Algorithm 2 constructs a BFS tree [where nodes at distance $d$ in $G$ are at depth $d$ in $T$] in $O(D)$ rounds.

**Proof.** We will show that, at the end of round $t \geq 1$,

- the parent$_i$ variable of every $p_i$ at distance $d_i \leq t$ from $p_r$ in $G$ points to a process at distance $d_i - 1$.
- only $M$ messages sent by processes at distance $t$ are in transit.

This implies:

- $T$ is BFS tree.
- The execution terminates within $D + 2$ rounds (one additional round for receiving last $M$ and for receiving $\langle already \rangle$ reply).
Synchronous Systems (II)

Proof. (con’t)

Induction:

- Basis $t = 1$:
  - In the initial configuration, $\forall i \neq r : \text{parent}_i = \emptyset$ and $M$ is in transit to all neighbors of $p_r$.
  - $\Rightarrow$ All $p_j$ at distance 1 from $p_r$ get $M$ in round 1, set $\text{parent}_j = p_r$, and forward $M$ to their neighbors $\neq p_r$.

- Induction step: Assume that the hypothesis holds for $t - 1 \geq 1$
  - every $p_i$ that receives $M$ in round $t$ is at distance $d \leq t$:
    - If $d < t$, $p_i$ has already set $\text{parent}_i$ $\Rightarrow$ does not forward $M$
    - If $d = t$, $p_i$ did not see $M$ before, hence sets $\text{parent}_i$ and forwards $M$ on the very first $M$ as required.
  - Processors at distance $> t$ cannot receive $M$ $\Rightarrow$ do nothing
Stop: The round invariant used in the proof of Theorem 84 is too weak for the proof to actually go through!

- The $parent_i$ variable of every $p_i$ at distance $d_i \leq t$ from $p_r$ in $G$ points to a process at distance $d_i - 1$
- Only $M$ messages sent by processes at distance $t$ are in transit
Synchronous Systems (III)

Stop: The round invariant used in the proof of Theorem 84 is too weak for the proof to actually go through!

- The parent variable of every \( p_i \) at distance \( d_i \leq t \) from \( p_r \) in \( G \) points to a process at distance \( d_i - 1 \)
- Only \( M \) messages sent by processes at distance \( t \) are in transit

In the proof, we (silently) used the fact that processes \( p_i \) at distance \( > t \) have \( parent_i = \emptyset \)!

- This does not follow from the original invariant!
- Need to add this requirement explicitly to the invariant!
Stop: The round invariant used in the proof of Theorem 84 is too weak for the proof to actually go through!

- The parent variable of every $p_i$ at distance $d_i \leq t$ from $p_r$ in $G$ points to a process at distance $d_i - 1$
- Only $M$ messages sent by processes at distance $t$ are in transit

In the proof, we (silently) used the fact that processes $p_i$ at distance $> t$ have parent $p_i = \emptyset$!

- This does not follow from the original invariant!
- Need to add this requirement explicitly to the invariant!

Make sure to state invariants that are strong enough!
Asynchronous Systems

Theorem 87. *In every admissible execution of an asynchronous system, Algorithm 2 constructs a spanning tree within time* $O(D)$

*Proof.* A simple induction proof—left as an exercise—shows that, by time $t$, the message $M$ reaches all $p_i$ at distance at most $t$ from $p_r$. □

In asynchronous systems, the spanning tree $T$

- need not be BFS
- can even be a chain (depth $n - 1$), although $D < n - 1$ (why is there no contradiction here?)
The BFS tree construction (Algorithm 2) ensures
- maximum concurrency and hence speed, but
- may connect direct neighbors in $G$ via long paths (via root) in $T$

Alternative: Depth-first search Algorithm 2.3
- Direct neighbors in $G$ are on path from the root in $T$
- Recursive pre-order traversal
- Concurrent—but in fact serialized—implementation of recursive DFS algorithm

**Theorem 88.** Algorithm 3 has time complexity $O(m)$ and message complexity $O(m)$

**Proof.** See textbook.
Multiple Processes per Processor
Multiple Processes

- In this course, we will primarily focus on a single process (= thread of control) per processor.
- In practice, multiple processes can be executed concurrently (multi-tasking or even multi-core) on a single processor.

Multiple processes per processor:
- allow multiple distributed algorithms to be executed in the system.
- facilitate modular algorithms and proofs (simulations).

**BUT:** Requires extensions of our formal distributed computing model.
Layered Process Model

Every processor (node) \( p_i \in \{p_0, \ldots, p_{n-1}\} \) executes \( k \geq 2 \) processes \( p_{i,1}, \ldots, p_{i,k} \) arranged in a stack.

- Layer-\( j \) process \( p_{i,j} \) communicates with \( p_{i,j-1} \) (its top process) and \( p_{i,j+1} \) (its bottom process).
- Top process of \( p_{i,1} \) is the environment/user of the DS.
- Bottom process of \( p_{i,k} \) is the (inter-processor) communication subsystem.

Local inter-process communication via events (that may carry additional data):

- \( p_{i,k} \) input events: Triggered by top resp. bottom process [may occur at any time, i.e., cannot be blocked by \( p_{i,k} \)].
- \( p_{i,k} \) output events: Triggered by \( p_{i,k} \) itself.
Example: Mutual Exclusion (I)

General setting:
- Distributed application, consisting of multiple client processes
- Client processes
  - alternate between remainder sections (RS) and critical sections (CS) in their code
  - invoke an underlying distributed mutual exclusion algorithm (ME) to ensure at most one client process in the CS at any time
- Modular design: Client implementation independent of ME implementation
Example: Mutual Exclusion (II)

ME top interface:

- **ME input events:** Signals client process at $p_i$ wants to
  - enter CS ($\phi_i^{\text{enterCS}}$)
  - exit CS ($\phi_i^{\text{exitCS}}$)

- **ME output events:** Grants client process $i$ to
  - enter CS (= exits RS) ($\phi_i^{\text{exitRS}}$)
  - exits CS (= enters RS) ($\phi_i^{\text{enterRS}}$)

Client processes only know semantics of ME top interface
Example: Mutual Exclusion (III)

Specification ME problem:

- Define set of feasible traces $\mathcal{E}_ME$ of the events at the top interfaces of all ME processes

- **Note:** Depends on client requests
Example: Mutual Exclusion (III)

**Specification** ME problem:

- Define set of feasible traces \( \mathcal{E}_{ME} \) of the events at the top interfaces of all ME processes
- **Note:** Depends on client requests

Every trace \( \beta \in \mathcal{E}_{ME} \) must satisfy:

- Events at processor \( p_i \), denoted \( p_i \)'s view \( \beta|i \), must occur cyclic: 
  \[
  \beta|i = \left\{ \phi_i^{\text{enterCS}} \phi_i^{\text{exitRS}} \phi_i^{\text{exitCS}} \phi_i^{\text{enterRS}} \right\}^* 
  \]

- \( \phi_i^{\text{exitRS}}, \phi_j^{\text{exitRS}} \) must have \( \phi_i^{\text{enterRS}} \) in between

- Additional liveness requirements, like no deadlock or no lockout
Example: Mutual Exclusion (IV)

ME bottom interface:

- Provides the ME process at $p_i$ with some means of communication with the ME processes at other processors $p_j$:
  - Bottom events $\text{send}_i(M, j)$ and $\text{recv}_i(M, j)$ (emulate standard message-passing communication via $\text{outbuf}_i[j]$ and $\text{inbuf}_i[j]$)
  - Bottom events $\text{bc-send}_i(M)$ and $\text{bc-recv}_i(M, j)$ (reliable broadcast simulation, implemented by ME bottom process)

Implementation complexity of the ME algorithm depends on bottom interface
General Trace-based Analysis

Initial configuration $C^0 + \text{schedule } \sigma = \phi^1 \phi^2 \ldots$ defines

- unique execution $\text{exec}(C^0, \sigma) = C^0, \phi^1, C^1, \phi^2, \ldots$
- unique sequence of steps $(C^0, \phi^1, C^1), (C^1, \phi^2, C^2), \ldots$
General Trace-based Analysis

Initial configuration $C^0 +$ schedule $\sigma = \phi^1 \phi^2 \ldots$ defines

- unique execution $\text{exec}(C^0, \sigma) = C^0, \phi^1, C^1, \phi^2, \ldots$
- unique sequence of steps $(C^0, \phi^1, C^1), (C^1, \phi^2, C^2), \ldots$

Abstract away irrelevant information:

- Augment event $\phi^k$ by additional data from $(C^{k-1}, \phi^k, C^k)$
  $\Rightarrow$ Sequence of augmented events $\phi^1 \phi^2 \ldots$ of $\text{exec}(C^0, \sigma)$
- Take subsequence of relevant augmented events (problem-dependent) $\Rightarrow$ trace of $\text{exec}(C^0, \sigma)$

Problem $\mathcal{P}$ specified by set $\mathcal{E}_\mathcal{P}$ of feasible traces
Simple Example

Relevant augmented events for Leader Election Problem:

- $p_i$ enters terminal state as leader ($\phi_i^{leader}$) or non-leader ($\phi_i^{nonleader}$)

- $E_{LE} = \text{all traces where } \phi_i^{leader} \text{ occurs once for exactly one } i \text{ and } \phi_j^{nonleader} \text{ for the remaining indices } j \neq i$
Simple Example

Relevant augmented events for Leader Election Problem:

- $p_i$ enters terminal state as leader ($\phi_i^{leader}$) or non-leader ($\phi_i^{nonleader}$)
- $\mathcal{E}_{LE} =$ all traces where $\phi_i^{leader}$ occurs once for exactly one $i$ and $\phi_j^{nonleader}$ for the remaining indices $j \neq i$

To prove that a distributed algorithm $A$ solves problem $P$:

- Study the set of traces $\mathcal{E}_A$ generated by all admissible executions of a distributed algorithm $A$
- **Correctness**: $\mathcal{E}_A \subseteq \mathcal{E}_P$
- **Non-triviality**: $\mathcal{E}_A$ contains non-trivial traces in $\mathcal{E}_P$
Trace-Inclusion-based Safety and Liveness

A trace property $\mathcal{E}$ is a safety property if it is

- non-empty: $\mathcal{E} \neq \emptyset$ (contains at least empty trace $\varepsilon$)
- prefix-closed: Every finite prefix $\beta_i$ of every trace $\beta \in \mathcal{E}$ is also in $\mathcal{E}$
- limit-closed: For every infinite sequence $\beta_1, \beta_2, \ldots$ of finite traces $\beta_i \in \mathcal{E}$, with $\beta_i$ being a prefix of $\beta_{i+1}$, the unique limit $\beta = \lim_{i \to \infty} \beta_i$ is also in $\mathcal{E}$
Trace-Inclusion-based Safety and Liveness

A trace property $\mathcal{E}$ is a safety property if it is

- **non-empty**: $\mathcal{E} \neq \emptyset$ (contains at least empty trace $\varepsilon$)
- **prefix-closed**: Every finite prefix $\beta_i$ of every trace $\beta \in \mathcal{E}$ is also in $\mathcal{E}$
- **limit-closed**: For every infinite sequence $\beta_1, \beta_2, \ldots$ of finite traces $\beta_i \in \mathcal{E}$, with $\beta_i$ being a prefix of $\beta_{i+1}$, the unique limit $\beta = \lim_{i \to \infty} \beta_i$ is also in $\mathcal{E}$

A trace property $\mathcal{E}$ is a liveness property if

- Every finite trace has some extension that is in $\mathcal{E}$
Caveat: More Complex Formal Model

Implications of multiple processes per processor:

- Replace processor id by tuple (processor id, process id)

⇒ Several events can be applicable on a single processor at any point in time [But: deterministic processes ⇒ at most one per process]

- Drop $\text{inbuf}_i[\ast]$, $\text{outbuf}_i[\ast]$ and deliver events altogether

- Drop assumption $\text{inbuf}_i[\ast] = \emptyset$

⇒ Simple $\text{comp}(i)$ events no longer sufficient: Need to also incorporate data from input events (e.g., actually received messages) controlled by the adversary
Leader Election in Rings
Motivation

The ability to elect a leader is often useful:

- Programming distributed applications typically easier in master/slave settings:
  - Broadcasting site (using e.g. a spanning tree)
  - Coordinator in distributed transactions
- Handling exceptional situations often requires a leader:
  - Breaking deadlocks
  - Token loss recovery in token rings/buses

Using dynamic rather than static leader election advantageous:

- Allows varying set of processors to choose from
- Allows to re-elect leader in case of leader failure
Definition Leader Election Problem

Every process $p_i$ irrevocably decides whether it considers itself leader or not upon termination $\Rightarrow$ Terminal states $T_i$ partitioned in (closed) sets of

- elected states
- non-elected states

Safety properties:
- At most one processor is in the elected state

Liveness properties:
- Eventually, every processor terminates
- Eventually, some processor enters an elected state
Rings

We consider processors arranged in a ring network. Why?
- Simple to analyze
- Abstraction of token ring
- Lower bounds for rings apply to arbitrary topologies

Oriented rings:
- Processors consistently distinguish left and right:
  - If $p_i$ sends msg to its right neighbor $p_j$, then $p_j$ gets msg from left neighbor
  - Put under the rug: Index $j$ in $outbuf_i[j], inbuf_i[j]$ is actually not processor id $p_j$, but local link number
- Sending message to left neighbor $\iff$ clockwise direction
Classiﬁcation of Algorithms for Rings

Direction:
- Unidirectional: Messages sent in one direction only
- Bidirectional: Messages sent in both directions

Availability of unique IDs:
- Non-anonymous: Every processor $p_i$ has UID $id_i$
- Anonymous: Processors are indistinguishable

Knowledge of ring size:
- Non-uniform algorithms: Processors know $n$
- Uniform algorithms: Same algorithm for every $n$
Overview of Upcoming Results

Our first impossibility result:

- There is no anonymous leader election algorithm

Some simple leader election algorithms:

- Asynchronous: $O(n^2)$ resp. $O(n \log n)$ messages
- Synchronous: $O(n)$ resp. $O(n \log n)$ messages

First lower bound results:

- Every asynchronous LE algorithm needs $\Omega(n \log n)$ msg’s
- Every synchronous LE algorithm needs
  - $\Omega(n)$ messages if certain tricks are allowed
  - $\Omega(n \log n)$ messages otherwise
- Lower bounds asymptotically tight: $\Omega \rightarrow \Theta$
Asynchronous Leader Election
Leader Election in Anonymous Rings

Recall: The $n$-processor leader election algorithm $A^n_i$ (including UID) at processor $p_i$ is called

- uniform if $A^n_i = A^m_i$ for every ring size $n, m$
- anonymous if $A^n_i = A^n_j$ for every pair of processors $i, j$
Leader Election in Anonymous Rings

Recall: The $n$-processor leader election algorithm $A^n_i$ (including UID) at processor $p_i$ is called

- uniform if $A^n_i = A^m_i$ for every ring size $n, m$
- anonymous if $A^n_i = A^n_j$ for every pair of processors $i, j$

Theorem 107. There is no anonymous deterministic leader election algorithm
Leader Election in Anonymous Rings

Recall: The $n$-processor leader election algorithm $A_i^n$ (including UID) at processor $p_i$ is called

- uniform if $A_i^n = A_i^m$ for every ring size $n, m$
- anonymous if $A_i^n = A_j^n$ for every pair of processors $i, j$

**Theorem 107.** There is no anonymous deterministic leader election algorithm

**Proof.** By contradiction: We show that there is not even a synchronous non-uniform anonymous anonymous LE algorithm:

- Induction on number of rounds reveals that every $p_i$ sends and receives the same messages
- Electing exactly one leader requires breaking this symmetry ⇒ impossible
A Simple Asynchronous LE Algorithm

Every processor $p_i$, $0 \leq i \leq n - 1$:

- sends its $id_i$ to left neighbor
- on receiving a message with $mid$ from the right neighbor:
  - if $mid > id_i$, forward it to the left ($p_i$ will never be leader)
  - if $mid = id_i$ (got back own message), enter elected state and send termination message
  - if $mid < id_i$, then swallow message
- on receiving termination message from right
  - if $p_i$ not yet in elected state $\Rightarrow$ forward termination message to the left and enter non-elected state
  - if $p_i$ in elected state $\Rightarrow$ swallow termination message
Correctness of Simple LE Algorithm

Safety and liveness proofs based on:

- Only message from $p_i$ with $id_i = \max$ never swallowed
- Only $p_i$ ever receives a message with $mid = id_i$ from the right and thus enters elected state
- All $p_j \neq p_i$ enter non-elected state via $p_i$’s termination msg

Detailed proofs left as a simple exercise.
Complexity Analysis of Simple LE Algorithm

Theorem 110. *The simple leader election algorithm has termination time* $2n$ *and sends* $\Theta(n^2)$ *messages*
Complexity Analysis of Simple LE Algorithm

Theorem 110. *The simple leader election algorithm has termination time* $2n$ *and sends* $\Theta(n^2)$ *messages*

*Proof.* The algorithm terminates after a full circulation of both $p_i$’s message and the termination message, taking time $n$ each.
Theorem 110. The simple leader election algorithm has termination time $2n$ and sends $\Theta(n^2)$ messages

Proof. The algorithm terminates after a full circulation of both $p_i$’s message and the termination message, taking time $n$ each.

Upper bound on message complexity:
- Every of the $n$ processors sends/forwards at most $n + 1$ messages
- $\Rightarrow$ need at most $O(n^2)$ total messages
**Theorem 110.** *The simple leader election algorithm has termination time* $2n$ *and sends* $\Theta(n^2)$ *messages*

*Proof.* The algorithm terminates after a full circulation of both $p_i$’s message and the termination message, taking time $n$ each.

Upper bound on message complexity:
- Every of the $n$ processors sends/forwards at most $n + 1$ messages
  $\Rightarrow$ need at most $O(n^2)$ total messages

Lower bound on message complexity:
- Consider ring $0, n - 1, n - 2, \ldots, 2, 1$
- The message from $p_i$ is sent/forwarded exactly $i + 1$ times
  $\Rightarrow$ need $n + \sum_{i=0}^{n-1} (i + 1) = \frac{n^2 + 3n}{2} = \Omega(n^2)$ messages
LE Algorithm by Le Lann / Chang & Roberts

Improve the message complexity of our simple algorithm by a more clever ("divide and conquer") forwarding:

- For $\ell \geq 0$, consider $2^\ell$-neighborhood of any $p_i$
  - $p_i$ itself
  - $2^\ell + 2^\ell$ consecutive processors to the left and right

Algorithm proceeds in consecutive phases $\ell \geq 0$ (not synchronized) at every processor

- In phase $\ell$, $p_i$ checks whether it is leader in its $2^\ell$-neighborhood:
  - If $p_i$ is leader $\Rightarrow$ proceed to next phase
  - Otherwise $\Rightarrow$ get stuck

$\Rightarrow$ Fewer and fewer processors proceed to higher phases
How to Explore $2^\ell$-Neighborhood?

Processor $p_i$ sends $\langle \text{probe}, id, \ell, hop \rangle$ messages to both left and right neighbor

- if $p_j$ receives $\langle \text{probe} \rangle$ with $id > id_j$, it either
  - forwards it in the same direction, with increased hop count (if $hop < 2^\ell$)
  - sends $\langle \text{reply}, id, \ell \rangle$ back in the opposite direction (if $hop \geq 2^\ell$, i.e., end of neighborhood reached)

- if $p_j$ receives $\langle \text{probe} \rangle$ with $id \leq id_j$, it swallows the msg

- if $p_j$ receives $\langle \text{reply} \rangle$ with $id \neq id_j$, it forwards the message in the same direction

$\Rightarrow$ $p_i$ gets back $\langle \text{reply}, id_i, \ell \rangle$ from left and right only if

$id_i = \max \text{ in } p_i$’s $2^\ell$-neighborhood (and gets stuck otherwise)
Complete LE Algorithm 6

Complete code:

- The above forwarding/swallowing rules +
- Leader termination: A processor that becomes leader in its $2^L$-neighborhood with $L = \lceil \log_2(n - 1) \rceil - 1$ (that is, $2^{L+1} + 1 \geq n$)
  - terminates in the elected state [in phase $L + 1$, where it gets own $\langle \text{probe} \rangle$ in exploration]
  - sends termination message to the left
- Non-leader termination: A processor not in the elected state that receives a termination message from the right
  - terminates in the not-elected state
  - forwards termination message to the left
Analysis of Algorithm 6

Correctness proof uses same argument as simple LE algorithm

Message and time complexity determined by exploration of $2^\ell$-neighborhood of any $p_i$:

- $2 \cdot 2^\ell \langle \text{probe} \rangle$ and $2 \cdot 2^\ell \langle \text{reply} \rangle$ messages
- totally $4 \cdot 2^\ell$ messages
- takes at most $2 \cdot 2^\ell$ time since left and right neighborhood explored concurrently

Last fully explored phase is $\ell = L = \lceil \log_2(n - 1) \rceil - 1$
Time Complexity of Algorithm 6

Theorem 115. Algorithm 6 has time complexity $O(n)$
Time Complexity of Algorithm 6

Theorem 115. Algorithm 6 has time complexity $O(n)$

Proof. Time complexity determined by the eventual leader $p_i$

- Explorations of $p_i$’s $2^\ell$-neighborhoods, $0 \leq \ell \leq L$ yields

$$\sum_{\ell=0}^{L} 2 \cdot 2^\ell = 2(2^{L+1} - 1) = 2(2^{\lceil \log_2(n-1) \rceil} - 1)$$

$$\leq 2(2^{\log_2(n-1)+1} - 1) = O(n)$$

- Additional time $2n$ for termination detection (incomplete exploration by leader) and termination message circulation in phase $L + 1$ already covered by $O(n)$

\qed
Lemma 116. The number of processors that are still leaders of their $2^\ell$-neighborhood at the end of phase $\ell \geq 0$ [and thus enter phase $\ell + 1$] is at most $\frac{n}{2^\ell + 1}$.

Note: Number of leaders surviving last phase $L = \lceil \log_2(n - 1) \rceil - 1$ is 1 as required, since $n/(2^L + 1) < 2$. 
Lemma 116. The number of processors that are still leaders of their $2^\ell$-neighborhood at the end of phase $\ell \geq 0$ [and thus enter phase $\ell + 1$] is at most $\frac{n}{2^\ell + 1}$

Note: Number of leaders surviving last phase $L = \lceil \log_2(n - 1) \rceil - 1$ is 1 as required, since $n/(2^L + 1) < 2$

Proof. Two leaders $p_i, p_j$ of their $2^\ell$-neighborhoods can at most share the same left resp. right neighborhood

- at least $2^\ell$ processors $\neq p_i, p_j$ in between
- dense packing over the entire ring $\Rightarrow$ at most $n/(2^\ell + 1)$ leaders
Theorem 117. Algorithm 6 sends $O(n \log n)$ messages
Theorem 117. Algorithm 6 sends $O(n \log n)$ messages

Proof. We know:

- Total number of any active $p_i$’s exploration messages for its $2^\ell$-neighborhood is $4 \cdot 2^\ell$
- Total number of active $p_i$’s in phase $\ell > 0$ is at most $n/(2^{\ell-1} + 1)$
- Total number of active $p_i$’s in phase $\ell = 0$ is $n$
- Termination detection and termination message circulation adds at most $O(n)$ additional messages

Hence, the total message complexity (including termination) is

$$O(n) + 4n + \sum_{\ell=1}^{L} 4 \cdot 2^\ell \frac{n}{2^{\ell-1} + 1} \leq O(n) + 8nL = O(n \log n)$$
We will show that ANY leader election algorithm $A$ that
(a) works in asynchronous rings
(b) is uniform
(c) elects processor with maximum $id$
(d) guarantees that every processor learns the $id$ of the leader
has message complexity $\Omega(n \log n)$
Asynchronous Lower Bound on Messages

We will show that ANY leader election algorithm $A$ that
(a) works in asynchronous rings
(b) is uniform
(c) elects processor with maximum $id$
(d) guarantees that every processor learns the $id$ of the leader

has message complexity $\Omega(n \log n)$

Conditions:
- (a) necessary for lower bound to hold, (b) for our proof to work
- (c) and (d) simplify the proof
Reduction

The important principle of reduction can be used to get rid of conditions (c) and (d)

Assume that

- we are given some uniform asynchronous LE algorithm $B$ that does not satisfy (c) and (d)
- with less than $\Omega(n \log n)$ additional messages, we can derive an algorithm $A$ that satisfies (c) and (d) from $B$

$\Rightarrow B$ must also send $\Omega(n \log n)$ messages, since $A$ derived from $B$ would need less than $\Omega(n \log n)$ otherwise

Note: Converting any $B$ to $A$ needs only $O(n)$ additional messages
Definitions Lower Bound Proof

We consider open schedules $\sigma$, defined as

- schedule $\sigma$ of some execution prefix of algorithm $A$
- there is an edge $e$ of the ring such that no message over $e$ is delivered (but maybe sent) in $\sigma$
- open schedule can be finite and need not be admissible

Additional assumptions for our proof:

- $n$ is a power of 2 (can be removed by reduction)
- the set $S$ of identifiers is an arbitrary subset of the natural numbers
Lemma 121. For $n = 2$, any algorithm $A$ has an open schedule $\sigma$ where at least $M(2) = 1$ messages are delivered.
Lemma 121. For $n = 2$, any algorithm $A$ has an open schedule $\sigma$ where at least $M(2) = 1$ messages are delivered.

Proof. Consider $p_0$ and $p_1$ where w.l.o.g. $id_0 > id_1$.

In any admissible execution $\alpha$,

- $p_0$ must send a message with $id_0$ to $p_1$ such that it can learn $id_0$, as required by condition (d)

- $\sigma$ is prefix of $\alpha$ up to and including the first del-event, w.l.o.g. over edge $(p_0, p_1)$

$\Rightarrow$ other edge $(p_1, p_0)$ is open and exactly $M(2) = 1$ messages are delivered as required.

$\square$
Lemma 122. For $n > 2$, any algorithm $A$ has an open schedule $\sigma$ where at least $M(n) = 2M(n/2) + \frac{1}{2}(n/2 - 1)$ messages are delivered.
**Lemma 122.** For \( n > 2 \), any algorithm \( A \) has an open schedule \( \sigma \) where at least \( M(n) = 2M(n/2) + \frac{1}{2}(n/2 - 1) \) messages are delivered.

**Proof.** By induction. Basis \( n = 2 \) is provided by previous lemma. Induction step:

- Split identifier set \( S \) into two halves \( S_1 \) and \( S_2 \), assigned to two rings \( R_1 \) and \( R_2 \) of \( n/2 \) processors each.

- Inductive hypothesis:
  - \( R_1 \) has open schedule \( \sigma_1 \) with at least \( M(n/2) \) messages and \( e_1 = (p_1, q_1) \) is open edge
  - \( R_2 \) has open schedule \( \sigma_2 \) with at least \( M(n/2) \) messages and \( e_2 = (p_2, q_2) \) is open edge

- Paste \( R_1 \) and \( R_2 \) together in a big ring \( R \), by replacing \( e_1, e_2 \) by \( e_p = (p_1, p_2) \) and \( e_q = (q_1, q_2) \)
Proof. (cont.)

Because of uniformity of $A$:

- Processors in $R_1$ cannot tell difference to left half of $R \Rightarrow$ same schedule $\sigma_1$ also in $R$.
- Processors in $R_2$ cannot tell difference to right half of $R \Rightarrow$ same schedule $\sigma_2$ also in $R$.

Distinguish 2 cases: Without unblocking $e_p$ and $e_q$,

1. The catenated schedule $\sigma_1 \sigma_2$ can be extended by some schedule $\sigma_3$ where additional $\frac{1}{2}(n/2 - 1)$ messages are received $\Rightarrow$ sought open schedule $\sigma = \sigma_1 \sigma_2 \sigma_3$ found and we are done.
2. Every extension of $\sigma_1 \sigma_2$ leads to a quiescent state.

\(\square\)
**Lower Bound Proof (IV)**

*Proof. (cont.)*

Let

- $\sigma_3$ be an extension of $\sigma_1 \sigma_2$ that leads to a quiescent state (without unblocking $e_p, e_q$)

- $\sigma'_4$ be an extension of $\sigma_1 \sigma_2 \sigma_3$ to an admissible schedule:
  - all processors terminate
  - all messages are delivered on $e_p$ and $e_q$

Claim: At least $n/2$ messages are sent in $\sigma'_4$, since

- every processor in the half of $R$ that does not contain the leader must get the leader’s id

- until the beginning of $\sigma'_4$, there has been no communication between the two halves
Lower Bound Proof (V)

Proof. (cont.)

Unfortunately, $\sigma_1\sigma_2\sigma_3\sigma'_4$ is not open.

Let $\sigma'_4$ be prefix of $\sigma''_4$ when $n/2 - 1$ additional messages have been delivered

- before $\sigma'_4$, system was quiescent
- set of processors $P, Q$ that delivered additional messages can only expand outwards around $e_p$ and $e_q$
- $P \cap Q = \emptyset$ since less than $n/2$ messages have been delivered in $\sigma'_4$ and every half consists of at least $n/2$ processors
- Assume that majority of the $n/2 - 1$ messages have been delivered in $P \Rightarrow$ at least $\frac{1}{2}(n/2 - 1)$ messages
Lower Bound Proof (VI)

Proof. (cont.)

Let $\sigma_4$ be the sequence of events of $\sigma'_4$ that involve processors in $P$ only.

Claim: In $\sigma = \sigma_1 \sigma_2 \sigma_3 \sigma_4$, processors in $P$ behave as in $\sigma_1 \sigma_2 \sigma_3 \sigma''_4$ and hence deliver at least $\frac{1}{2}(n/2 - 1)$ messages, since

- $P \cap Q = \emptyset$ imply that processors in $P$ cannot have heard anything from processors in $Q$.

Note: The ability to use $\sigma_4$ instead of $\sigma'_4$ relies heavily on asynchrony assumption (a).

Hence, in $\sigma$,

- $e_q$ can be left open

- still, at least $2M(n/2) + \frac{1}{2}(n/2 - 1)$ messages are delivered
Final Lower Bound Proof

We know now that any LE algorithm $A$ delivers at least

$$M(n) = 2M(n/2) + \frac{1}{2}(n/2 - 1)$$

$$M(2) = 1$$

messages in some admissible execution, since it does so in an open schedule.
Final Lower Bound Proof

We know now that any LE algorithm $A$ delivers at least

$$M(n) = 2M(n/2) + \frac{1}{2}(n/2 - 1)$$

$$M(2) = 1$$

messages in some admissible execution, since it does so in an open schedule.

Expanding the definition of $M(n)$ reveals

$$\log_2 n$$ terms of order $\Omega(n)$ each

$$M(n) = \Omega(n \log n)$$
Final Lower Bound Proof

We know now that any LE algorithm $A$ delivers at least

- $M(n) = 2M(n/2) + \frac{1}{2}(n/2 - 1)$
- $M(2) = 1$

messages in some admissible execution, since it does so in an open schedule.

Expanding the definition of $M(n)$ reveals

- $\log_2 n$ terms of order $\Omega(n)$ each
- $M(n) = \Omega(n \log n)$

This finally confirms our lower bound $\Omega(n \log n)$. 
Synchronous Leader Election
Leader Election in Synchronous Rings

In asynchronous systems,

- messages can be arbitrarily delayed
- information can only be disseminated by sending a message
Leader Election in Synchronous Rings

In asynchronous systems,
- messages can be arbitrarily delayed
- information can only be disseminated by sending a message

In synchronous systems,
- messages must arrive by the beginning of the next round
- information can also be disseminated by *not* sending a message ("communication by time")
Leader Election in Synchronous Rings

In asynchronous systems,
- messages can be arbitrarily delayed
- information can only be disseminated by sending a message

In synchronous systems,
- messages must arrive by the beginning of the next round
- information can also be disseminated by **not** sending a message ("communication by time")

Question: Does this help for solving Leader Election?
Leader Election in Synchronous Rings

In asynchronous systems,

- messages can be arbitrarily delayed
- information can only be disseminated by sending a message

In synchronous systems,

- messages must arrive by the beginning of the next round
- information can also be disseminated by not sending a message ("communication by time")

Question: Does this help for solving Leader Election? YES!
A Synchronous LE Algorithm

We consider
- a unidirectional ring
- a non-uniform algorithm (knows ring size $n$)

The algorithm:
- Proceeds in a finite (but unbounded) number of consecutive phases $x \geq 0$
- Every phase $x$ consists of $n$ rounds where
  - process $p_i$ with $id_i = x$ sends message containing $id_i$
  - every $p_j$ that gets message with $id \neq id_j$ forwards message to the left and terminates as a non-leader
  - $p_i$ terminates as the leader when it gets msg $id = id_i$
The algorithm:
- Elects the processor $p_i$ with minimal $id_i$ as leader
- Requires synchronous rounds
- Non-uniformity is not vital: More advanced synchronous Algorithm 3.2 in textbook is uniform
Properties Synchronous LE Algorithm

The algorithm:

- Elects the processor $p_i$ with minimal $id_i$ as leader
- Requires synchronous rounds
- Non-uniformity is not vital: More advanced synchronous Algorithm 3.2 in textbook is uniform

Trivial performance analysis:

- Message complexity: $n$
- Time complexity: $(id_i + 1)n$ rounds
Properties Synchronous LE Algorithm

The algorithm:
- Elects the processor $p_i$ with minimal $id_i$ as leader
- Requires synchronous rounds
- Non-uniformity is not vital: More advanced synchronous Algorithm 3.2 in textbook is uniform

Trivial performance analysis:
- Message complexity: $n$
- Time complexity: $(id_i + 1)n$ rounds

But:
- Termination time depends on particular choice of $id$’s
- The $id$’s are not just compared but used for deciding when to send a message
Comparison-Based Algorithms (I)

Some definitions:

- Two rings $R_1 = (x_1, x_2, \ldots, x_n)$ and $R_2 = (y_1, y_2, \ldots, y_n)$ are order-equivalent if $x_i < x_j ⇔ y_i < y_j$ for any $i, j$

- A ring is spaced if there are at least $n$ unused id’s between any two $x_i, x_j$

- Processors $p_i$ in $R_1$ and $p_j$ in $R_2$ are matching if they have same distance from processor with minimum id

- Local executions $\alpha_1$ and $\alpha_2$ at $p_1$ and $p_2$ in $R_1$ and $R_2$, respectively, are called similar if, for all rounds $k$,
  - $p_1$ sends a message to left (right) neighbor in round $k$ in $\alpha_1 ⇔ p_2$ does so in $\alpha_2$
  - $p_1$ terminates as a leader/non-leader in round $k$ in $\alpha_1 ⇔ p_2$ does so in $\alpha_2$
Some more definitions:

- An algorithm is called **comparison-based** if every pair of matching processors have similar behaviors in order equivalent rings $R_1$ and $R_2$.

- A round $r$ is **active** in an execution if some processor sends a message in round $r$.

- $r_k$ is the number of the $k$-th active round [with $r_0 = 0$ representing the (virtual) “initial round” ending in the initial configuration.]

In case of a comparison-based algorithm:

- Order equivalent rings have the same sequence of active rounds $r_k$, $k \geq 0$.

- We will show even more ...
Behavior in Identical Neighborhoods

Lemma 134. Let $p_1$ and $p_2$ be processors with identical $k$-neighborhoods, $k \geq 0$, in order-equivalent rings $R_1$ and $R_2$. Then, $p_1$ and $p_2$ are in the same state after rounds $0, \ldots, r_k$. 
Lemma 134. Let \( p_1 \) and \( p_2 \) be processors with identical \( k \)-neighborhoods, \( k \geq 0 \), in order-equivalent rings \( R_1 \) and \( R_2 \). Then, \( p_1 \) and \( p_2 \) are in the same state after rounds \( 0, \ldots, r_k \).

Proof. By induction: For \( k = 0 \), \( p_1 \) and \( p_2 \) have the same \( id_1 = id_2 \) and are hence in the same initial state after (virtual) round \( r_0 = 0 \).

Induction step: Since identical \( k \)-neighborhood (\( = p_i + k \) left + \( k \) right neighbors) implies identical \( k - 1 \)-neighborhood, we can assume that

- \( p_1 \) and \( p_2 \) are in the same state after \( r_{k-1} \)
- left and right neighbor \( p^l_1, p^r_1 \) of \( p_1 \) are in same state as \( p^l_2, p^r_2 \)
- in non-active rounds \( r_{k-1} + 1, \ldots, r_k - 1 \), \( p_1 \) and \( p_2 \), as well as \( p^l_1 \) and \( p^l_2 \), and also \( p^r_1 \) and \( p^r_2 \), perform the same state transitions
- in round \( r_k \), \( p_1 \) and \( p_2 \) receive the same messages \( \Rightarrow \) perform the same state transition.
Lemma 135. Let $p_1$ and $p_2$ be processors with order-equivalent $k$-neighborhoods in a single spaced ring $R$. Then, $p_1$ and $p_2$ have similar behaviors in rounds $1, \ldots, r_k$. 
Lemma 135. Let $p_1$ and $p_2$ be processors with order-equivalent $k$-neighborhoods in a single spaced ring $R$. Then, $p_1$ and $p_2$ have similar behaviors in rounds $1, \ldots, r_k$.

Proof. Since $R$ is spaced and $p_1$, $p_2$ have order-equivalent $k$-neighborhoods, we can construct another ring $R'$ that satisfies:

- $p_2$'s $k$-neighborhood in $R'$ is identical with $p_1$'s $k$-neighborhood in $R$
- $p_2$ in $R'$ matches $p_2$ in $R$
- $R$ and $R'$ are order-equivalent
- the id's in $R'$ are unique

The lemma follows since, in rounds $1, \ldots, r_k$,

- $p_1$ in $R$ has identical state as $p_2$ in $R'$ by previous lemma
- $p_2$ in $R'$ is matching to $p_2$ in $R$ $\Rightarrow$ similar behaviors since algorithm is comparison-based
We know that

- processors with order-equivalent neighborhoods have similar behaviors
- we need to find just one ring where any algorithm needs $\Omega(n \log n)$ messages
Outline Lower Bound Proof

We know that

- processors with order-equivalent neighborhoods have similar behaviors

- we need to find just one ring where any algorithm needs \( \Omega(n \log n) \) messages

We will proceed as follows:

- Construct a ring \( S_n \) where any \( p_i \)'s neighborhood is order-equivalent to many other \( p_j \)'s neighborhood

- At least one processor sends a msg per active round \( \Rightarrow \) many messages sent per active round

- Show that there is a lower bound for the number of active rounds in \( S_n \)

- Summing up over all active rounds yields \( \Omega(n \log n) \)
Consider the ring $R_n = (0, 1, 2, \ldots, n - 1)$.

- Let $\text{rev}(i)$ be the integer corresponding to the reverse binary representation of $i$

  Example: $i = 4 = 100_2 \Rightarrow \text{rev}(i) = 1 = 001_2$

- Define $R_n^{\text{rev}} = (\text{rev}(0), \text{rev}(1), \ldots, \text{rev}(n - 1))$
Consider the ring $R_n = (0, 1, 2, \ldots, n-1)$.

Let $\text{rev}(i)$ be the integer corresponding to the reverse binary representation of $i$.

Example: $i = 4 = 100_2 \Rightarrow \text{rev}(i) = 1 = 001_2$

Define $R_n^{\text{rev}} = (\text{rev}(0), \text{rev}(1), \ldots, \text{rev}(n-1))$

The ring $R_n^{\text{rev}}$ has interesting properties:

- One can show that all segments of $2^k$ consecutive processors are order equivalent.
- This property is preserved in the spaced ring $S_n$, where every $id$ in $R_n^{\text{rev}}$ is replaced by $(n + 1) \cdot id + n$.
Order Equivalent Neighborhoods in $S_n$

**Lemma 138.** For $k \leq n/8$ and all $k$-neighborhoods $N$ of $S_n$, there are more than $\frac{n}{2(2k+1)}$ $k$-neighborhoods that are order equivalent to $N$ (including $N$).
Lemma 138. For $k \leq n/8$ and all $k$-neighborhoods $N$ of $S_n$, there are more than $\frac{n}{2(2k+1)}$ $k$-neighborhoods that are order equivalent to $N$ (including $N$).

Proof. Let $j = 2^\ell$ be such that $2(2k+1) > j \geq 2k + 1$.

Partition $S_n$ in $n/j > \frac{n}{2(2k+1)}$ consecutive segments, such that

- one segment totally encompasses $N$

$\Rightarrow$ all segments are order equivalent, by the properties of $S_n$.

$\square$
Order Equivalent Neighborhoods in $S_n$

**Lemma 138.** For $k \leq n/8$ and all $k$-neighborhoods $N$ of $S_n$, there are more than $\frac{n}{2(2k+1)}$ $k$-neighborhoods that are order equivalent to $N$ (including $N$).

**Proof.** Let $j = 2^\ell$ be such that $2(2k + 1) > j \geq 2k + 1$.

Partition $S_n$ in $n/j > \frac{n}{2(2k+1)}$ consecutive segments, such that

- one segment totally encompasses $N$
- all segments are order equivalent, by the properties of $S_n$

**Corollary 138.** At least $\frac{n}{2(2k+1)}$ messages are sent in the $k$-th active round.
Lemma 139. Any leader election algorithm needs $T \geq n/8$ active rounds in $S_n$ for $n \geq 8$. 
Lemma 139. Any leader election algorithm needs $T \geq n/8$ active rounds in $S_n$ for $n \geq 8$.

Proof. Suppose $T < n/8$ and let $p_i$ be the eventual leader. By the previous lemma, there are more than

$$\frac{n}{2(2T + 1)} > \frac{n}{2(2n/8 + 1)} = \frac{2n}{n + 4} > 1$$

order equivalent $T$-neighborhoods.

Hence,

- at least one $p_j$ has order equivalent $T$-neighborhood w.r.t. leader $p_i$

$\Rightarrow$ $p_j$ is also elected by Lemma 135.

$\Rightarrow$ Contradiction.
Comparison-based Lower Bound

**Theorem 140.** For every $n = 2^\ell \geq 8$, there is a ring $S_n$ where every synchronous leader election algorithm $A$ sends $\Omega(n \log n)$ messages.
Theorem 140. For every \( n = 2^\ell \geq 8 \), there is a ring \( S_n \) where every synchronous leader election algorithm \( A \) sends \( \Omega(n \log n) \) messages.

Proof. By the previous lemmas, the number of messages is more than

\[
\sum_{k=1}^{n/8} \frac{n}{2(2k + 1)} \geq \frac{n}{6} \sum_{k=1}^{n/8} \frac{1}{k} = \Omega(n \log n)
\]

Note that \( A \) needs to be comparison-based on \( id \)'s out of \( \{0, 1, \ldots, n^2 + 2n - 1\} \) only:

- largest \( id_{\text{max}} \) in \( S_n \) is \( (n + 1) \text{rev}(n - 1) + n = n^2 + n - 1 \)
- Ring \( R' \) in Lemma 135 may need \( n \) additional \( id \)'s larger than \( id_{\text{max}} \)
Time-bounded Algorithms

Recall synchronous algorithm:

- Time complexity depends on choice of id’s
- What if we disallow such a behavior?

Consider time-bounded leader election algorithms:

- Draw any subset \( S_n \) of \( n \) distinct id’s from \( \mathbb{N} \)
- Worst-case running time must be bounded (over all \( S_n \))

We will show:

- Any time-bounded algorithm also needs \( \Omega(n \log n) \) messages
- We will use reduction to comparison-based algorithms
Some Preparations ...

A synchronous LE algorithm is \( t \)-comparison based for

- identifier set \( S \)
- order equivalent rings \( R_1 \) and \( R_2 \) with id’s from \( S \)

if every pair of matching processors have similar behaviors in rounds \( 1, \ldots, t \).

We also need:

**Theorem 142. (Ramsey’s Theorem)**

For all integers \( k, \ell \) and \( t \), there is some integer \( f(k, \ell, t) \) such that for every set \( S \) with \( |S| = f(k, \ell, t) \) and every \( t \)-coloring of the \( k \)-subsets of \( S \), some \( \ell \)-subset of \( S \) has all its \( k \)-subsets with the same color.
Lemma 143. Let $A$ be any synchronous time-bounded LE algorithm with running time bound $r(n)$ in rings with size $n$. Then, there is some set $C_n$ of $n^2 + 2n$ identifiers such that $A$ is $r(n)$-comparison-based over $C_n$. 
Lemma 143. Let $A$ be any synchronous time-bounded LE algorithm with running time bound $r(n)$ in rings with size $n$. Then, there is some set $C_n$ of $n^2 + 2n$ identifiers such that $A$ is $r(n)$-comparison-based over $C_n$.

Proof. Consider $n$-subsets $S_1, S_2 \subseteq \mathbb{N}$:

- $S_1, S_2$ are equivalent if matching processors in every pair of order-equivalent rings $R_1$ (using id’s from $S_1$) and $R_2$ (using id’s from $S_2$) have similar behavior.

- The equivalence relation partitions the $n$-subsets of $\mathbb{N}$ into finitely many equivalence classes:
  - There are only finitely many $[(n - 1)!]$ different orders $R_i$ with id’s from $S_i$.
  - There are only finitely many different possible message and termination patterns in $r(n)$ (finitely many!) rounds for any $R_i$. 


Time-bounded LE Lower Bound (II)

Proof. (cont.) Apply Ramsey’s Theorem:

- \( t \) is number of equivalence classes (colors)
- \( \ell = n^2 + 2n \)
- \( k = n \)

Since \( \mathbb{N} \) is infinite, there is

- some subset \( S \subseteq \mathbb{N} \) with size \( f(k, \ell, t) \)
- some subset \( C_n \subseteq S \) with \( |C_n| = n^2 + 2n \)
- where all \( n \)-subsets of \( C_n \) have same color (are equivalent)

Hence, algorithm \( A \) is \( r(n) \)-comparison-based over \( C_n \):

- Any two order-equivalent rings \( R_1, R_2 \) with id’s from \( S_1, S_2 \subseteq C_n \), respectively, are equivalent

\( \Rightarrow \) Matching processors have similar behaviors
Theorem 145. Every synchronous time-bounded LE algorithm $A$ sends $\Omega(n \log n)$ messages on some ring $R$ of size $n = 2^\ell \geq 8$. 
Time-bounded LE Lower Bound (III)

**Theorem 145.** *Every synchronous time-bounded LE algorithm $A$ sends $\Omega(n \log n)$ messages on some ring $R$ of size $n = 2^\ell \geq 8$.*

**Proof.** We cannot directly apply comparison-based lower bound theorem, since previous lemma holds for specific set $C_n = \{c_0, c_1, \ldots, c_{n^2+2n-1}\}$ only.

We construct modified algorithm $A'$ from $A$, which has

- id's in $S = \{0, 1, \ldots, n^2 + 2n - 1\}$
- $p_i$ with $id_i = i$ executes algorithm $A$ as if it had $id_i = c_i$

$\Rightarrow A'$ is $r(n)$-comparison-based over $S$ and terminates in $r(n)$ rounds

The $\Omega(n \log n)$ lower bounds follows from

- the comparison-based lower bound theorem, since $A'$ and $A$ send same number of messages by construction
Mutual Exclusion in Shared Memory
Shared Memory Systems

We consider asynchronous systems made up of

- \( n \) processors \( p_0, \ldots, p_{n-1} \)
- \( m \) shared memory variables (registers) \( R_0, \ldots, R_{m-1} \)

Distinguish shared memory variables by:

- **Type:** Which atomic operations supported?
  - Test-and-set
  - Read-modify-write
  - Compare-and-swap

- **Access:** Who may simultaneously access?
  - Multiple writer, multiple reader
  - [Single writer, multiple reader]
Formal Model of SHM systems (I)

Processor $p_i$ again modeled as (deterministic) state machine $P_i = (Q_i, \Phi_i, I_i, T_i)$

- state set $Q_i$ (possibly infinite)
- set of initial states $I_i \subseteq Q_i$
- set of terminal states $T_i \subseteq Q_i$
- transition function $\Phi_i \subseteq Q_i \times Q_i$ (successor relation)

Transition $(q, q') \in \Phi_i$, triggered by event $\phi_i$

- only when $p_i$ is in state $q$ (enabling condition for step $(q_i, \phi_i, q_i')$)
- modifies at most one SHM register $R_k$
- moves $p_i$ to state $q_i'$
State set $Q_i = L_i \cup S_i$ consists of

- local state $L_i$ (variables, registers)
- locally accessible portion $S_i$ of shared state
Formal Model of SHM systems (II)

State set $Q_i = L_i \cup S_i$ consists of
- local state $L_i$ (variables, registers)
- locally accessible portion $S_i$ of shared state

The only events $\phi_i$ in the SHM model are computing events $\text{comp}(i)$, which trigger a step $(q_i, \phi_i, q_i')$ of $p_i$ consisting of:

1. choosing a single shared variable $R_k \in S_i$, depending on $p_i$’s current local state $l_i$ in $q_i$
2. performing the SHM operation on $R_k$, according to its type
3. changing $p_i$’s local state to $l_i'$, depending on $l_i$ and the result of the SHM operation
Formal Model of SHM systems (III)

A configuration $C = (l_0, \ldots, l_{n-1}; r_0, \ldots, r_{m-1})$ of the global state machine consist of

- every processor $p_i$'s local state $l_i \in L_i$
- the actual content $r_k$ of every SHM register $R_k$, abbreviated as $\text{mem}(C) = (r_0, \ldots, r_{m-1})$

A configuration $C$ is similar to $C'$ w.r.t. a set $P$ of processors, denoted by $C \overset{P}{\sim} C'$, if

- every $p_i \in P$ has same local state in $C$ and $C'$
- $\text{mem}(C) = \text{mem}(C')$

No $p_i \in P$ sees any difference between $C$ and $C'$
Formal Model of SHM systems (IV)

- Execution segment $\alpha$ is finite sequence of configurations alternating with events
- Schedule $\sigma$ of $\alpha$ is corresponding sequence of events
- $P$-only schedule $\sigma$ solely from subset $P$ of processors
- Configuration $C$ and schedule $\sigma = \phi_{i_1} \ldots \phi_{i_m}$ uniquely determine execution segment, ending up in config $C' = \sigma(C) = \phi_{i_m}(\phi_{i_{m-1}}(\ldots \phi_{i_1}(C) \ldots))$
- Configuration $C'$ is reachable from $C$ if some schedule $\sigma$ exists such that $C' = \sigma(C)$

Processors in terminal states
- move to (same or other) terminal state only,
- do not modify any SHM variable
Asynchronous systems:

- Computing steps occur in zero time
- Time between successive steps of any processor is finite, but
- No upper and lower bounds on the time between local computing steps

Admissible executions:

- Every \( p_i \) executes infinitely many steps
- Note: Exception for Mutual Exclusion Problem for convenience
### Pseudo-Code Conventions (I)

1. \( \text{Want} := \text{Want} + 1 \)
2. \( \text{Priority} := \text{Priority} + \text{Want} + \text{pri} \)
3. \( \text{num} := \max\{\text{Number}[0], \ldots, \text{Number}[n-1]\} \)
4. wait until \( \text{Want} = 0 \)

In our pseudo-code descriptions,

- SHM variable names start with upper-case character
- Single statement could involve multiple computing steps [depending on SHM type]:

<table>
<thead>
<tr>
<th>SHM type</th>
<th>line 1</th>
<th>line 2</th>
<th>line 3</th>
<th>line 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W:</td>
<td>2</td>
<td>3</td>
<td>(n)</td>
<td>(K) (?)</td>
</tr>
<tr>
<td>RMW (atomic add):</td>
<td>1</td>
<td>2</td>
<td>(n)</td>
<td>(K) (?)</td>
</tr>
</tbody>
</table>
Pseudo-Code Conventions (II)

1 \hspace{1em} Want := Want + 1
2 \hspace{1em} Priority := Priority + Want + pri
3 \hspace{1em} num := \max\{Number[0], \ldots, Number[n - 1]\}
4 \hspace{1em} \text{wait until } Want = 0

The processor $p$ executing this pseudo-code is said to

- have reached line 2 $\iff p$ has already executed line 1 but not line 2
- be within lines 2–4 $\iff p$ has already executed line 2 or 3 but not line 4
Goals of Formal Analysis

Correctness of distributed SHM algorithms:
- Safety properties
- Liveness properties

Performance analysis:
- SHM space complexity, measured in
  - memory bits
  - number of variables
- Time complexity, measured as
  - execution time if maximum inter-step time at any processor is $\tau = 1$
  - somewhat simplistic due to possibly contention-dependent SHM access times
Time Complexity Analysis

For time complexity analysis,

- starting the execution of a code segment of \( k \) steps at time \( t \) completes by time \( t + (k - 1)\tau = t + (k - 1) \)
- next step \( k + 1 \) occurs by time \( t + k \)
- detailed step counting usually avoided via \( O(.) \):

1. \( \text{Want} := 0 \)
2. \( \text{Priority} := \text{Priority} + \text{pri} \)
3. \( \text{num} := \max\{\text{Number}[0], \ldots, \text{Number}[n - 1]\} \)
4. \( \text{wait until } \text{Want} = 0 \)

has time complexity \( O(1) + O(n) + O(K) = O(n + K) \),
where \( K \) is a bound on the number of iterations in line 4.
Mutual Exclusion Problem (I)

Each processor’s entire code divided in 4 sections, in endless loop:

- **Entry** section: Synchronize with others to ensure ME
- **Critical** section: Do what you have to do exclusively, but
  - do not stay indefinitely here
  - do not access synchronization-related SHM registers
- Simplifying assumption: CS is empty (no “own” step).
- **Exit** section: Clean up synchronization with others
- **Remainder** section: Non-exclusive computations. Our assumptions:
  - Remainder section is also empty, but
  - any processor may forever stop after last step in exit even in admissible executions

182.702 Distributed Algorithms (Prof. Schmid), http://ti.tuwien.ac.at/ecs/teaching/courses/valg) – p. 157/315
Mutual Exclusion Problem (II)

Mutual exclusion algorithms provide code for entry and exit sections, which guarantees:

- Mutual exclusion (safety): In every configuration of every execution, at most one processor is in the critical section

- One of the following liveness properties:
  - No deadlock: If \( p_i \) is within the entry section at some time, then later some \( p_j \) is in the critical section
  - No lockout: If \( p_i \) is within the entry section at some time, then later this \( p_i \) is in the critical section
  - \( k \)-bounded waiting: No deadlock + while \( p_i \) is within the entry section, no other processor enters the CS \( > k \) times
Mutual Exclusion Problem (III)

A note on $k$-bounded waiting:

- 0-bounded waiting = FIFO ordering
- But: The previous definition of $k$-bounded waiting refers to events, not to event occurrence times!
  - No difference if entry section is entered at different times
  - If $p_j$ enters entry section at the same time as $p_i$ ⇒ any event order possible (without further information)!

⇒ One has to conservatively assume that $p_j$ overtakes $p_i$ (and vice versa) in this case!
Overview of Upcoming Results

Powerful SHM register types:
- ME algorithm based on test-and-set
- ME algorithm based on read-modify-write
- Lower bound on required number of memory bits

Simple read/write SHM Registers:
- Lamport’s bakery ME algorithm (unbounded range)
- Peterson’s two-processor ME algorithm (bounded range)
- Lower bound on required number of registers

Restrict attention to multiple writer+reader variables
**Test-And-Set Variables**

A binary test-and-set variable $V$ can be accessed via two operations:

- $TAS(V)$ applied to address $V$ returns a binary value:

  \[
  \begin{align*}
  \text{temp} & := V \\
  V & := 1 \\
  \text{return(temp)}
  \end{align*}
  \]

  executed atomically

- $RESET(V)$ applied to address $V$ does $V := 0$

Atomicity important to

- avoid race conditions

- two processors could both get 0 from simultaneous $TAS(V)$
ME Algorithm with Test-And-Set (I)

Code TAS algorithm 7:

- **Entry:** wait until \( \text{TAS}(V) = 0 \)
- **Exit:** \( \text{RESET}(V) \)

**Theorem 162.** The TAS algorithm 7 guarantees mutual exclusion and no deadlock of \( n \) processors with a single test-and-set variable.
ME Algorithm with Test-And-Set (I)

Code TAS algorithm 7:
- Entry: wait until $TAS(V) = 0$
- Exit: $RESET(V)$

Theorem 162. *The TAS algorithm 7 guarantees mutual exclusion and no deadlock of* $n$ *processors with a single test-and-set variable*

*Proof.* Mutual exclusion is proved by induction on the number $k \geq 0$ of entries of the CS (by any process):
- Induction basis $k = 0$: Since no process entered the CS, mutual exclusion trivially holds.
- Induction step: Assume that ME held for the first $k - 1 \geq 0$ entries in CS.
  - Assume that ME is violated when the $k$-th process enters CS.
  - Derive a contradiction.
Proof. (cont.)

Let $t_j$ be the time when ME is violated upon the $k$-th entry, i.e., some processor $p_j$ enters the CS although $p_i$ is still in. Thus,

- $V$ has been set at time $t_i$ by $p_i$
- no other processor has entered and hence left the CS in $[t_i, t_j]$ since $p_j$ entry is first one where ME is violated (induction hypothesis!)
- $V$ must still be 1 at time $t_j$, so $p_j$ cannot have read $V = 0$ on entering the CS
ME Algorithm with Test-And-Set (II)

Proof. (cont.)

Let $t_j$ be the time when ME is violated upon the $k$-th entry, i.e., some processor $p_j$ enters the CS although $p_i$ is still in. Thus,

- $V$ has been set at time $t_i$ by $p_i$
- no other processor has entered and hence left the CS in $[t_i, t_j]$ since $p_j$ entry is first one where ME is violated (induction hypothesis!)
- $V$ must still be 1 at time $t_j$, so $p_j$ cannot have read $V = 0$ on entering the CS

Proceed with proof of no deadlock (by contradiction) . . .
Proof. (Cont.)

Assume $p_j$ is within entry section at $t_j$ but no processor enters CS at time $t > t_j$. However,

- no processor may remain in CS forever $\Rightarrow$ there is a time $t'$ where the processor in CS at $t_j$, if any, has left CS and exit section
- Invariant (left to reader): $V = 0 \iff$ no processor is in CS

Hence, $p_j$ must discover $V = 0$ and enter CS at some time $t \geq t'$

Hence, the TAS algorithm

- guarantees no deadlock (but NOT no lockout)
- needs just 1 bit of memory (for holding 2 states)
Read-Modify-Write Variables

Generic operation \( RMW(V, f) \):

\[
\begin{align*}
\text{temp} & := V \\
V & := f(\text{temp}) \\
\text{return(} & \text{temp}\text{)}
\end{align*}
\]

executed atomically

A read-modify-write variable \( V \) thus allows to atomically

- read \( V \)'s value \( v \)
- compute a new value \( v' \) using a type-dependent function \( v' = f(v) \), like:
  - Test-and-set: \( f(v) \equiv 1 \)
  - Compare-and-swap: Input parameters \( w, w' \)
    \[ f(v, w, w') := \text{if } v = w \text{ then } w' \text{ else } v \]
- update \( V \)'s value to \( v' \) accordingly
Basic data structure is “virtual” circular queue of length $n$:

- Processors waiting in entry section entered in queue
- Processors remember their position in the queue (“ticket”) in local variable

Shared variable $V$ keeps track of active part of queue via $V.first$ and $V.last$ pointers $\in \{0, \ldots, n - 1\}$

Entry code:

- Increment $V.last$ modulo $n$ to enqueue self
- Wait until $V.first$ equals this value

Exit code: Increment $V.first$ modulo $n$ to dequeue itself
Pseudo-Code RMW ME Algorithm 8

Code for every processor:
1. Initially $V = \langle 0, 0 \rangle$

/* Code for entry section: */
2. $pos := RMW(V, \langle V.first, V.last + 1 \rangle)$  // enqueueing at tail
3. repeat   // Per iteration: Lines 4 and 5 in a single step!
4.  $queue := RMW(V, V)$  // read head of queue
5. until $queue.first = pos.last$  // until becomes first

/* Critical section */

/* Code for exit section */
6. $RMW(V, \langle V.first + 1, V.last \rangle)$  // advance head of queue
Detailed proof of safety and liveness of Algorithm 8 is complicated by the modulo-operations involved in RMW

- We first consider a variant of Algorithm 8, where the buffer (and the size of the RMW variable) is unbounded:
  - Algorithm 8’ has same pseudo-code as Algorithm 8, but
  - RMW increments $V.first$ and $V.last$ not modulo $n$

Proof outline:

- Prove that Algorithm 8’ solves ME with 0-bounded waiting (= FIFO) and no deadlock
- Carry over these properties to Algorithm 8 using a simulation relation
We say that a processor is

- within the critical ∪ exit section, if it has passed line 5 but not executed line 6,
- within the entry ∪ critical ∪ exit section, if it has executed line 2 but not line 6

By the semantics of \textit{RMW} and the code of Algorithm 8’, we immediately obtain the following simple properties:

- Both \textit{V.first} and \textit{V.last} is advanced in strict sequence
  \{0, 1, 2, 3, \ldots \}

- Processors draw unique \textbf{tickets} \textit{pos}_i.last in line 2, in strict sequence \{0, 1, 2, 3, \ldots \}

We proceed with some invariants \ldots
Lemma 170. *In every reachable configuration of Algorithm 8’, there is at most one process $p_i$ within critical $\cup$ exit section, and its ticket satisfies $\text{pos}_i.\text{last} = V.\text{first}$.*

*Proof.* By induction; left as an exercise. $\square$
Lemma 170. *In every reachable configuration of Algorithm 8’, there is at most one process* $p_i$ *within critical ∪ exit section, and its ticket satisfies* $\text{pos}_i.\text{last} = V.\text{first}$.

*Proof.* By induction; left as an exercise. □

Theorem 170. *Algorithm 8’ guarantees mutual exclusion.*
**Lemma 170.** In every reachable configuration of Algorithm 8', there is at most one process $p_i$ within critical $\cup$ exit section, and its ticket satisfies $pos_i.last = V.first$.

*Proof.* By induction; left as an exercise. □

**Theorem 170.** Algorithm 8’ guarantees mutual exclusion.

*Proof.* Follows immediately from the above invariant. □
Lemma 171. *In every reachable configuration of Algorithm 8’, every processor $p_i$ that is within the entry $\cup$ critical $\cup$ exit section has drawn (and not returned) a unique ticket $pos_i$.last in the interval $[V.first, V.last)$, and $V.last - V.first$ equals the number $d$ of processors that have drawn a ticket.*

*Proof.* By induction; left as an exercise. $\square$
Lemma 171. In every reachable configuration of Algorithm 8’, every processor $p_i$ that is within the entry $\cup$ critical $\cup$ exit section has drawn (and not returned) a unique ticket $pos_i$.last in the interval $[V.$first, $V.$last$)$, and $V.$last $-$ $V.$first equals the number $d$ of processors that have drawn a ticket.

Proof. By induction; left as an exercise. □

Theorem 171. Algorithm 8’ guarantees 0-bounded waiting (= FIFO) and no deadlock.
Lemma 171. *In every reachable configuration of Algorithm 8’, every processor* $p_i$ *that is within the entry* $\cup$ *critical* $\cup$ *exit section has drawn (and not returned) a unique ticket* $pos_i\cdot last$ *in the interval* $[V.first, V.last)$, *and* $V.last - V.first$ *equals the number* $d$ *of processors that have drawn a ticket.*

*Proof.* By induction; left as an exercise. 

Theorem 171. *Algorithm 8’ guarantees 0-bounded waiting (= FIFO) and no deadlock.*

*Proof.* First, it follows from the above invariant that in case $p_i$ has drawn a ticket $pos_i\cdot last$

$$V.first \leq pos_i\cdot last < V.last = V.first + d \leq V.first + n$$

since $1 \leq d \leq n$ processors can have drawn a ticket. 

□
Proof. (cont.)

Assume for a contradiction that no deadlock does not hold:

1. Let $t$ be the first time when there are $d > 0$ processors that have drawn a ticket (without entering CS yet), but no processor enters the CS after $t$.

2. Let $p_i$ be the processor with the smallest ticket among the $d > 0$ ones.

3. By the previous invariant, $p_i$ must have the ticket $pos_i.last = V.first$.

However, $pos_i.last = V.first$ is exactly the condition (line 5) that causes $p_i$ to enter the CS — a contradiction.

And since processors enter the CS in the order of drawn tickets, this also implies 0-bounded waiting.

This completes the correctness proof of Algorithm 8’.
Simulation Relation (I)

Given two distributed algorithms (state-machines) $\mathcal{L}$ and $\mathcal{H}$ solving the same problem $\mathcal{P}$

- $\mathcal{H} = (C_H, \Phi_H, I_H, T_H)$, the higher abstraction level (Algorithm 8’)
- $\mathcal{L} = (C_L, \Phi_L, I_L, T_L)$, the lower abstraction level (Algorithm 8)

Generated traces (recall augmented events):

- $\mathcal{E}_\mathcal{H} = \{\beta | \beta = \mathcal{E}(\alpha) \text{ where } \alpha \text{ is execution of } \mathcal{H}\}$
- $\mathcal{E}_\mathcal{L} = \{\beta | \beta = \mathcal{E}(\alpha) \text{ where } \alpha \text{ is execution of } \mathcal{L}\}$

Map execution segments of $\mathcal{L}$ to executions segments of $\mathcal{H}$
Simulation Relation (II)

A binary relation $f \subseteq C_L \times C_H$ is a simulation relation (also called abstraction function) if

- initial states of $L$ are mapped to initial states of $H$:
  $$\forall C_0^L \in I_L : f(C_0^L) \cap I_H \neq \emptyset, \text{ with } u \in f(s) \iff (s, u) \in f$$

- events of $L$ are mapped to execution segments of $H$:
  
  For all reachable states $C_L \in C_L, C_H \in C_H$ with $C_H \in f(C_L)$, and
  
  transition $(C_L, \phi_L, C'_L) \in \Phi_L$

there is some

- execution segment $\alpha_H = C'_H, \phi_H^1, \ldots, \phi_H^k, C'_H$ with $k \geq 0$ and $C'_H \in f(C'_L)$, such that

- the traces $E(\phi_L) = E(\alpha_H)$ are the same
Simulation Relation (III)

**Theorem 175.** If there is a simulation relation $f$ from $\mathcal{L}$ to $\mathcal{H}$, then $\mathcal{E}_\mathcal{L} \subseteq \mathcal{E}_\mathcal{H}$. 
Theorem 175. If there is a simulation relation \( f \) from \( \mathcal{L} \) to \( \mathcal{H} \), then \( \mathcal{E}_\mathcal{L} \subseteq \mathcal{E}_\mathcal{H} \).

Proof. Using induction on the number of events in any (not necessarily admissible) exec. \( \alpha_\mathcal{L} \) of \( \mathcal{L} \), a (not necessarily admissible) exec. \( \alpha_\mathcal{H} \) of \( \mathcal{H} \) with \( \mathcal{E}(\alpha_\mathcal{L}) = \mathcal{E}(\alpha_\mathcal{H}) \) can be constructed. Hence, \( \mathcal{E}_\mathcal{L} \subseteq \mathcal{E}_\mathcal{H} \). \( \square \)
Simulation Relation (III)

**Theorem 175.** If there is a simulation relation $f$ from $L$ to $H$, then $E_L \subseteq E_H$.

**Proof.** Using induction on the number of events in any (not necessarily admissible) exec. $\alpha_L$ of $L$, a (not necessarily admissible) exec. $\alpha_H$ of $H$ with $E(\alpha_L) = E(\alpha_H)$ can be constructed. Hence, $E_L \subseteq E_H$. $\square$

**Theorem 175.** If there is a simulation relation $f$ from $L$ to $H$, then every safety property satisfied by $H$ is also satisfied by $L$. 
Simulation Relation (III)

Theorem 175. If there is a simulation relation $f$ from $\mathcal{L}$ to $\mathcal{H}$, then $\mathcal{E}_\mathcal{L} \subseteq \mathcal{E}_\mathcal{H}$.

Proof. Using induction on the number of events in any (not necessarily admissible) exec. $\alpha_\mathcal{L}$ of $\mathcal{L}$, a (not necessarily admissible) exec. $\alpha_\mathcal{H}$ of $\mathcal{H}$ with $\mathcal{E}(\alpha_\mathcal{L}) = \mathcal{E}(\alpha_\mathcal{H})$ can be constructed. Hence, $\mathcal{E}_\mathcal{L} \subseteq \mathcal{E}_\mathcal{H}$.  

Theorem 175. If there is a simulation relation $f$ from $\mathcal{L}$ to $\mathcal{H}$, then every safety property satisfied by $\mathcal{H}$ is also satisfied by $\mathcal{L}$.

Proof. Let $P$ denote the intersection of all safety properties satisfied by $\mathcal{H}$. Then, $\mathcal{E}_\mathcal{H} \subseteq \mathcal{E}_P$ by definition. Since $\mathcal{E}_\mathcal{L} \subseteq \mathcal{E}_\mathcal{H}$ by the above theorem, we also have $\mathcal{E}_\mathcal{L} \subseteq \mathcal{E}_P$ as required.
What about liveness properties?

- Constructing simulated execution $\alpha_H$ is entirely dictated by $\alpha_L$
- $f$ does not necessarily lead to admissible $\alpha_H$, as certain possible transitions are not taken
- $f$ sometimes (e.g., when it is 1:1 w.r.t. steps) also preserves liveness properties; the simulated execution $\alpha_H$ is then also admissible.
Theorem 177. Algorithm 8 guarantees mutual exclusion with 0-bounded waiting and no deadlock using a single RMW variable with $2^\lceil \log_2 n \rceil$ bits.
Theorem 177. Algorithm 8 guarantees mutual exclusion with 0-bounded waiting and no deadlock using a single RMW variable with $2\lceil \log_2 n \rceil$ bits.

Proof. We choose $\mathcal{L}$ as Algorithm 8 and $\mathcal{H}$ as Algorithm 8’ and consider all events as external events.

For defining $f$, we just let $(C_L, C_H) \in f$ iff

- $V(C_L).first \equiv V(C_H).first \mod n$
- $V(C_L).last \equiv V(C_H).last \mod n$
- $\forall i : \text{pos}_i(C_L).last \equiv \text{pos}_i(C_H).last \mod n$

We will show that $f$ is a simulation relation …
Proof. (cont.) $f$ is indeed a simulation relation, since

- the unique initial state $C'_L = C'_{\mathcal{H}}$ is also an initial state for $\mathcal{H}$ \Rightarrow the initial state mapping requirement is trivially fulfilled

- there is a 1-1 correspondence between the events in Algorithm 8 and Algorithm 8' \Rightarrow we only need to show $C'_{\mathcal{H}} \in f(C'_L)$.

Since $C_{\mathcal{H}} \in f(C'_L)$ holds by assumption, all state transitions involving only operations “invariant” w.r.t. $\text{mod } n$ (e.g., $V$ and $\text{pos}_i$ copied or incremented) obviously maintain $C'_{\mathcal{H}} \in f(C'_L)$.

Only the equality check in line 5 could cause a problem:

- Suppose $V(C'_L).\text{first} = \text{pos}_i(C'_L).\text{last}$, then Algorithm 8 would enter the CS after $(C'_L, \phi_L, C'_{\mathcal{L}})$.

- We must show that Algorithm 8' does the same, i.e., that also $V(C_{\mathcal{H}}).\text{first} = \text{pos}_i(C_{\mathcal{H}}).\text{last}$ in this case.
Analysis RMW ME Algorithm 8 (III)

Proof. (cont.) However, since

- $V(C_H).first \equiv pos_i(C_H).last \mod n$ (which follows from $V(C_L).first = pos_i(C_L).last$ and the definition of $f$)
- $V(C_H).first \leq pos_i(C_H).last < V(C_H).first + n$ (from the second invariant of Algorithm 8’)

$V(C_H).first = pos_i(C_H).last$ must indeed hold.

Hence, $f$ is a simulation relation. Consequently, Algorithm 8

- satisfies all safety properties of Algorithm 8’
- even satisfies all liveness properties of Algorithm 8’, since $f$ establishes a 1-1 correspondence between admissible executions

In Algorithm 8, $V.first$ and $V.last$ take on at most $n$ values, hence $V$

- needs $n^2$ different SHM states and hence $2 \lceil \log_2 n \rceil$ bits
Theorem 180. Any algorithm that solves mutual exclusion with \( k \)-bounded waiting, for some \( k \), uses at least \( n \) distinct shared memory states.
Theorem 180. Any algorithm that solves mutual exclusion with \( k \)-bounded waiting, for some \( k \), uses at least \( n \) distinct shared memory states.

Proof. Start from initial configuration \( C \) (all \( p_i \) in remainder)

- \( \exists \) infinite \( p_0 \)-only schedule \( \tau'_0 \) such that \( \text{exec}(C, \tau'_0) \) is admissible

\( \Rightarrow \) By no deadlock: \( \exists \) prefix \( \tau_0 \) such that \( p_0 \) is in the CS in \( C_0 = \tau_0(C) \)

Inductively, let

- \( \tau_i \) be \( p_i \)-only schedule that drives \( p_i \) into the entry section when starting from \( C_{i-1} \)

\( \Rightarrow \) \( p_0 \) is in CS and \( \{p_1, \ldots, p_i\} \) are within entry section in configuration \( C_i, 0 \leq i \leq n - 1 \)
Lower Bound on Memory States (II)

Proof. (cont.)

Assume, by way of contradiction, that there are less than \( n \) distinct SHM states. Then, there is some \( i < j \) with \( C_i \{ p_0, \ldots, p_i \} \sim C_j \) since

- there must be \( i < j \) such that \( \text{mem}(C_i) = \text{mem}(C_j) \)
- \( C_j = \tau(C_i) \) where \( \{ p_0, \ldots, p_i \} \) do not take steps in \( \tau = \tau_{i+1} \cdots \tau_j \) by construction

Apply an infinite \( \{ p_0, \ldots, p_i \} \)-only schedule to \( C_i \) that leads to an admissible execution. By no deadlock,

- some processor \( p_\ell \in \{ p_0, \ldots, p_i \} \) must enter CS \( \infty \) often
- there is some prefix \( \rho \) of \( \rho' \) such that \( p_\ell \) enters CS \( k + 1 \) times

\( \Box \)
Proof. (cont.)

Since $C_i \{p_0, \ldots, p_i\} \sim C_j$, it follows that:

- Applying $\rho$ to $C_j$ produces same execution for $p_\ell \Rightarrow p_\ell$ enters CS $k + 1$ times also when starting from $C_j$

- However, resulting execution not admissible since $p_{i+1}, \ldots, p_j$ do not take steps

-Appending schedule $\sigma$ where every $\{p_0, \ldots, p_j\}$ takes infinitely many steps provides admissible execution where $p_j$ waiting in entry section has been overtaken $k + 1$ times, a contradiction.
Mutual Exclusion with R/W Registers

Lamport’s Bakery Algorithm:
- Customers arriving in a bakery
- Get successively numbered tickets on entry
- Only customer with the smallest ticket is actually served

SHM variables used in pseudo code:
- $Number[i]$ holds $p_i$’s ticket (0 if none assigned)
- $Choosing[i]$ is true if $p_i$ is about to get its ticket
- Use additional processor id $i$ for creating unique tickets ($Number[i], i$)

Note: $Number[i]$ could grow without bound
Pseudo-Code Algorithm 10

Bakery algorithm: Code for processor \( p_i \), \( 0 \leq i \leq n - 1 \)

1. \( \text{VAR } \text{Choosing}[\forall j] := \text{false}; \text{Number}[\forall j] := 0 \)

   /* Code for entry section: */
   2. \( \text{Choosing}[i] := \text{true} \)
   3. \( \text{Number}[i] := \max\{\text{Number}[0], \ldots, \text{Number}[n-1]\} + 1 \)
   4. \( \text{Choosing}[i] := \text{false} \)
   5. for \( j = 0 \) to \( n - 1 \) (except \( j = i \)) do
   6. \( \quad \text{wait until } \text{Choosing}[j] = \text{false} \) // About to get ticket ?
   7. \( \quad \text{wait until } \text{Number}[j] = 0 \)
   \( \quad \text{or } (\text{Number}[j], j) > (\text{Number}[i], i) \)
   /* Critical section */

/* Code for exit section */
8. \( \text{Number}[i] := 0 \) // Throw away used ticket
Lemma 185. In every configuration $C$ of an execution $\alpha$ of Algorithm 10, if $p_i$ is in the CS and $\text{Number}[j] \neq 0$ for any $j \neq i$, then $p_j$ has a larger ticket than $p_i$, that is, $(\text{Number}[j], j) > (\text{Number}[i], i)$. 
**Correctness Bakery Algorithm (I)**

**Lemma 185.** In every configuration $C$ of an execution $\alpha$ of Algorithm 10, if $p_i$ is in the CS and $\text{Number}[j] \neq 0$ for any $j \neq i$, then $p_j$ has a larger ticket than $p_i$, that is, $(\text{Number}[j], j) > (\text{Number}[i], i)$.

**Proof.** Key argument for invariant induction: Before entering CS, $p_i$ must have finished second wait (line 7) for $j$. It either read

- $\text{Number}[j] = 0 \Rightarrow p_i$ observed $\text{Choosing}[j] = \text{false}$, and $p_j$ did not finish choosing a ticket either. Hence, $p_j$ must have started line 3 (if at all) after $p_i$ wrote $\text{Number}[i]$, such that $p_j$'s ticket is indeed larger than $p_i$'s.

- $(\text{Number}[j], j) > (\text{Number}[i], i) \Rightarrow$ remains true until $p_j$ chooses another ticket (or $p_i$ leaves CS). In the former case, the previous item reveals that $p_j$'s new ticket must also be larger than $p_i$'s.

Hence: Only process with smallest ticket can enter CS!
Theorem 186. Algorithm 10 provides mutual exclusion and no lockout.

Proof. Mutual exclusion: Assume, by way of contradiction, that both \( p_i \) and \( p_j \) are in CS. It is easy to show that \( \text{Number}[k] > 0 \) if \( p_k \) is in CS. Applying the previous lemma twice hence yields a contradiction, since 
\[
(\text{Number}[j], j) > (\text{Number}[i], i) \quad \text{and} \quad (\text{Number}[i], i) > (\text{Number}[j], j).
\]

No lockout: Assume, by way of contradiction, that ticket \( T \), for some processor \( p_i \), is the smallest starving ticket.

However, \( p_i \) must eventually pass testing loop and enter CS, since

- All processors \( p_j \) that execute entry code after \( p_i \) choose larger tickets \( \Rightarrow \) will not enter CS by previous lemma
- all \( p_j \)’s with smaller tickets are not starved by assumption and hence eventually exit CS.

\[ \square \]
Consider a ME algorithm for 2 processors $p_0$ and $p_1$ only:

- $p_i$ uses SHM variable $\text{Want}[i]$ to signal interest to enter CS
- In case of both $\text{Want}[0] = \text{true}$ and $\text{Want}[1] = \text{true}$, one processor retreats
- Additional SHM variable $\text{Priority}$ says who has (not) to retreat [simply remembers last CS exit]

Note: Textbook starts with unsymmetric algorithm where

- $p_1$ has to retreat always

$\Rightarrow$ Can only guarantee no deadlock
Pseudo-Code Algorithm 12

Peterson’s algorithm: Code for processor $p_i$, $i \in \{0, 1\}$

1. \text{VAR} \ Want[\forall j] := \text{false}; \ Priority := 0

/* Code for entry section: */

2. Want[i] := false

3. wait until Want[1 - i] = false or Priority = i

4. Want[i] := true \ // declare interest

5. if Priority = 1 - i then

6. \hspace{1em} if Want[1 - i] = true then goto line 2 \ // retreat

7. \hspace{1em} else

8. \hspace{2em} wait until Want[1 - i] = false \ // wait for exit

/* Critical section */

/* Code for exit section */

9. Priority := 1 - i \ // turn to other processor

10. Want[i] := false
Lemma 189. In Peterson’s Algorithm 12, if processor $p_i$ loops in line 3 (resp. loops in line 8 or reaches the CS), then $\text{Want}[i] := \text{false}$ (resp. $\text{Want}[i] := \text{true}$).

Proof. Obvious from the code.
Correctness Algorithm 12 (I)

Lemma 189. In Peterson’s Algorithm 12, if processor $p_i$ loops in line 3 (resp. loops in line 8 or reaches the CS), then $\text{Want}[i] := \text{false}$ (resp. $\text{Want}[i] := \text{true}$).

Proof. Obvious from the code. □

Theorem 189. Peterson’s Algorithm 12 provides mutual exclusion.
Correctness Algorithm 12 (I)

Lemma 189. In Peterson’s Algorithm 12, if processor $p_i$ loops in line 3 (resp. loops in line 8 or reaches the CS), then $\text{Want}[i] := \text{false}$ (resp. $\text{Want}[i] := \text{true}$).

Proof. Obvious from the code. □

Theorem 189. Peterson’s Algorithm 12 provides mutual exclusion.

Proof. Assume, by way of contradiction, that both $p_0$ and $p_1$ are in CS at some time $t$. By the previous lemma, both $\text{Want}[0] = \text{true}$ and $\text{Want}[1] = \text{true}$ at $t$.

Assume w.l.o.g. that, when entering CS,

- $p_1$’s last write $\text{Want}[1] := \text{true}$ happens before
- $p_0$’s last write $\text{Want}[0] := \text{true}$

From the code, $p_0$ can enter CS via line 6 or line 8, where it must read $\text{Want}[1] = \text{false}$ in both cases – a contradiction. □
Theorem 190. *Peterson’s Algorithm 12 provides no deadlock.*
Theorem 190. *Peterson’s Algorithm 12 provides no deadlock.*

*Proof.* Assume that both $p_0$ and $p_1$ get stuck in the entry section, with w.l.o.g. $p_1$ being the last process that enters. Let $P$ be the value of *Priority* at this time; note that this variable does not change any more.

If $P = 0$, then

- $p_1$ never reaches line 8, hence must loop forever within lines 2–6
- $p_0$ must eventually reach and loop forever in line 8 $\Rightarrow$ $Want[0] = \text{true by previous lemma}$
- $p_1$ must hence eventually reach and loop forever in line 3 $\Rightarrow$ $Want[1] = \text{false by previous lemma}$

$\Rightarrow p_0$ cannot loop forever in line 8, a contradiction.

$\square$
Proof. (cont.)

If \( P = 1 \), then

- \( p_0 \) was the last to execute line 9 in the exit section
- already \( \text{Priority} = 1 \) at the time \( p_0 \) entered the entry section
- same argument as above, with \( p_0 \) and \( p_1 \) reversed, yields contradiction.

If just one processor, say, \( p_0 \), gets stuck in the entry section without the other process entering CS subsequently,

- \( p_1 \) must eventually leave CS and stay forever in remainder section

\[ \Rightarrow \text{Want}[1] = \text{false forever, so } p_0 \text{ cannot loop forever due to lines 3, 6 and 8. Hence, it must enter CS.} \]
Correctness Algorithm 12 (IV)

Theorem 192. *Peterson’s Algorithm 12 provides no lockout.*
Theorem 192. *Peterson’s Algorithm 12 provides no lockout.*

*Proof.* Assume, by way of contradiction, that w.l.o.g. $p_0$ is starved and thus gets stuck in the entry section.

- If $p_1$ executes line 9 where it sets $\text{Priority} := 0$, it remains 0 forever, so $p_0$ passes the test in line 3 and skips line 6.
- $p_0$ must forever loop in line 8, waiting for $\text{Want}[1] = \text{false}$.
- Could only happen if $p_1$ gets stuck in entry section as well, which would violate no deadlock.

- If $p_1$ never executes line 9,
  - $p_1$ must remain forever in remainder section.
  - $\text{Want}[1] = \text{false}$, so $p_0$ cannot loop forever due to lines 3, 6 and 8. Hence, it enters CS.
Derive $n$-processor ME algorithm from 2-processor one

- Let processors compete pairwise, using $n/2$ instances of 2-processor ME algorithms
- Do the same for the $n/2$ "winners”, etc.

Corresponds to arranging processors as leaves of a tournament tree

- A process that got up $k$ levels in the tree passed the entry section of $k$ 2-processor ME algorithms
- Only one process can win at the root of the tree
  ⇒ enters the “real” critical section

Textbook shows: Algorithm 13 provides ME and no lockout
The algorithms seen so far need
- at least one SHM variable if powerful primitives like test-and-set are available
- \( O(n) \) R/W SHM variables

We will show now that any ME algorithm that guarantees no deadlock needs at least \( n \) R/W variables:
- Trivial if single-writer, since every process must write something to a dedicated variable to let others know
- Advanced lower bound proof for multiple writer variables
Some definitions:

- A processor covers a variable [at most one] in a configuration if it is about to write it [in the next event].
- For any set $P$ of processors, a configuration $C$ is $P$-quiescent if there exists a reachable quiescent configuration $D$ such that $C \sim^P D$. 
Preliminaries Lower Bound Proof

Some definitions:

- A processor covers a variable [at most one] in a configuration if it is about to write it [in the next event].

- For any set $P$ of processors, a configuration $C$ is $P$-quiescent if there exists a reachable quiescent configuration $D$ such that $C \sim_P D$.

Our lower bound proof will exploit the following:

- Every processor $p_k$ must inform the others that it wants to enter the CS.

- This must be done in a not-yet covered variable, since $p_k$’s writing to already covered variables could be overwritten [without the overwritten content being read!].
Lemma 196. Let $C$ be a reachable $p_i$-quiescent configuration for some $p_i$. Then there is a $p_i$-only schedule $\sigma$ such that $p_i$ is in $CS$ in $\sigma(C')$, and $p_i$ writes to at least one variable uncovered in $C'$ during $\sigma$. 
Preparation Lemma (I)

Lemma 196. Let $C$ be a reachable $p_i$-quiescent configuration for some $p_i$. Then there is a $p_i$-only schedule $\sigma$ such that $p_i$ is in CS in $\sigma(C')$, and $p_i$ writes to at least one variable uncovered in $C$ during $\sigma$.

Proof. Since $C$ is $p_i$-quiescent, there is a quiescent configuration $D$ with $C \overset{p_i}{\sim} D$. By no deadlock,

- if $p_i$ alone takes steps starting from $D$, it must eventually enter CS
- the same must happen when this schedule $\sigma$ is started from $C$

Assume, by way of contradiction, that $p_i$ only writes to variables already covered in $C$. Let

- $W$ be the set of variables covered by processors $\neq p_i$
- $P$ be a set of processors covering every variable in $W$ exactly once (recall that any $p_j$ can cover at most one variable)
Preparation Lemma (II)

Proof. (cont.) Starting from $C$,

- let every processor in $P$ take exactly one step $\Rightarrow$ every variable in $W$ is now overwritten
- then invoke no deadlock and unobstructed exit to show that every processor not in the remainder can get to it

Call the resulting schedule $\tau$ and note that the reached configuration $Q = \tau(C')$ is quiescent. Pick any processor $p_j \neq p_i$ and let $\pi$ be a $p_j$-only schedule starting from $Q$ that moves $p_j$ into CS.

- During the first steps of $\tau$, other processors overwrite anything somebody (like $p_i$ during $\sigma$) may have written

$\Rightarrow$ During $\tau$ and $\pi$, other processors cannot tell whether $p_i$ has executed $\sigma$ or not [although $\text{mem}(C') \neq \text{mem}(\sigma(C'))$]!

Hence, $p_j$ is in CS both in configuration $\tau\pi(C')$ and $\sigma\tau\pi(C')$ — but in the latter, $p_i$ is also in CS, a contradiction. $\square$
To show that one needs at least $n$ variables,

- the preparation lemma cannot simply be used successively, for every processor:
  - using it e.g. for $p_0$ need not lead to a configuration that is $P$-quiescent for the remaining processors
  - cannot employ preparation lemma again for $p_1$
Lower Bound Number of R/W Variables (I)

To show that one needs at least $n$ variables,

- the preparation lemma cannot simply be used successively, for every processor:
  - using it e.g. for $p_0$ need not lead to a configuration that is $P$-quiescent for the remaining processors
  - cannot employ preparation lemma again for $p_1$

But we can use the following lemma with $k = n$ and $C$ equal to the initial configuration for proving our lower bound:

**Lemma 198.** For any $1 \leq k \leq n$, let $P_k = \{p_0, \ldots, p_{k-1}\}$ and $P^k = \{p_k, \ldots, p_{n-1}\}$. For all reachable quiescent configurations $C$, there is a $P^k$-quiescent configuration $C_k$ reachable from $C$ by a $P_k$-only schedule such that the processors in $P_k$ cover $k$ distinct variables in $C_k$. 

182.702 Distributed Algorithms (Prof. Schmid), http://ti.tuwien.ac.at/ecs/teaching/courses/valg) – p. 198/315
**Lower Bound Number of R/W Variables (II)**

*Proof.* By induction. Basis is $k = 1$

- By preparation lemma, there is a $p_0$-only schedule $\sigma'$ where at least one write to variable $X$ is performed

- Let $C_1 = \sigma(C')$ be the configuration reached by the prefix $\sigma$ of all events in $\sigma'$ up to but excluding the first write

- $C_1$ covers $X$ and is $P^1$-quiescent since only $p_0$ took steps and $\text{mem}(C_1) = \text{mem}(C')$, as required.

Induction step: Assume lemma holds for $k \geq 1$ and show it for $k + 1$. For purposes of simpler explanation,

- we silently assume that every application of induction hypothesis causes same set $W_{k'}$ of $k'$ covered variables to appear

- will be removed subsequently, by using the fact that we can only have finite number of different sets of $k$ covered variables
Proof. (cont.) By inductive hypothesis, we can reach some $P^k$-quiescent $C_k$ where the processors in $P_k$ cover $W_k$. Starting from $C_k$,

- apply the $p_k$-only schedule $\sigma$ guaranteed by the preparation lemma to have additional variable $X$ covered
- But: $\sigma(C_k)$ not necessarily $P^{k+1}$-quiescent since $p_k$ might also have written to some already covered variables

Need more work: Similar to the proof of preparation lemma,

- let every processor in $P_k$ take exactly one step $\Rightarrow$ every variable in $W_k$ is now overwritten
- then invoke no deadlock and unobstructed exit to show that every processor in $P_k$ not in the remainder can get to it

Call the latter schedule $\tau$ and let $D_k = \sigma\tau(C_k) = \tau(\sigma(C_k))$
Proof. (cont.) We cannot invoke the inductive hypothesis starting from $D_k$, however, since it is not quiescent ($p_k$ not in remainder). Still,

- we could apply $\tau$ also to $C_k$, without applying $\sigma$ first
- the configuration $D_k^* = \tau(C_k)$ is quiescent $\Rightarrow$ we can apply inductive hypothesis
- by applying the hypothesized schedule $\sigma_k$ to $D_k^*$, we can reach a $P^k$-quiescent configuration $C_k^*$ where $P_k$ covers $W_k'$

Since obviously $D_k^* \forall p_j \neq p_k \sim D_k$,

- processors in $P_k$ do the same in $\text{exec}(D_k, \sigma_k)$ as in $\text{exec}(D_k^*, \sigma_k)$
- in the reached $P^{k+1}$-quiescent configuration $\sigma_k(D_k) =: C_{k+1}$, exactly $k + 1$ variables $W_{k+1} = W_k' \cup X$ are covered by processes in $P_{k+1}$
Lower Bound Number of R/W Variables (V)

Proof. (cont.) Unfortunately, we will usually have different sets

- $W_k$ of variables covered in $C_k$
- $W_k'$ of variables covered in $D_k^*$

$\Rightarrow$ We cannot claim $W_k \subseteq W_{k+1} = W_k' \cup X$ needed for our induction proof to go through

However, there are only finitely many different sets of $k$ variables:

- We just iterate our schedule $\tau \sigma_k$ sufficiently often
- There must be some schedule $\tau^1 \sigma_1^k \cdots \tau^x \sigma_x^k$ that produces $W_k' = W_k$ (a single one is sufficient for our proof)

Since $W_k' = W_k$, we have indeed constructed the sought configuration $C_{k+1}'$ and we are done
Fault-Tolerant Consensus
Processor Failures

Up to now, we did not consider failures. From now on,

- an unknown set $F$ of processors may be(come) faulty
- we do not know when a faulty processor becomes faulty

We just know

- how many processors $0 \leq f \leq n$ may at most be faulty during the entire execution ($|F| \leq f$)
- which kind of failures are allowed:
  - **Crash failures**: A processor simply stops executing events (also in the middle of a broadcast)
  - **Byzantine failures**: A processor can do whatever it wants (including sending arbitrary messages)
- Communication is still reliable [could be dropped]
The Consensus Problem

Every processor $p_i$ has

- an input value $x_i$ from some finite set (often binary)
- an output value $y_i$, initially undefined
- a consensus algorithm that computes a value for $y_i$

Required properties in every admissible execution:

- **Termination**: $y_i$ is irrevocably assigned a value at every non-faulty processor $p_i$ eventually

- **Agreement**: $y_i = y_j$ for all terminated non-faulty processors $p_i$ and $p_j$

- **Validity**: If $x_k = v$ for all processors $p_k$, then $y_i = v$ for every terminated non-faulty processor $p_i$
Lamport’s Byzantine Army

Consider several divisions of the Byzantine army, each commanded by a general, camped outside an enemy city.

- Every general has some local opinion of whether to attack, say, at noon, or not
- Byzantine army can win only if all (loyal) divisions are attacking together
- Generals can communicate via reliable messengers ⇒ need to execute a consensus protocol

Still,

- some of the Byzantine generals may be traitors, who
- send confusing messages to trigger an inconsistent attack of a subset of loyal generals only
Overview of Consensus Results

Synchronous message passing case:

<table>
<thead>
<tr>
<th>$f$-resilient Algorithm</th>
<th>Crash</th>
<th>Byzantine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of rounds:</td>
<td>$f + 1$</td>
<td>$f + 1$</td>
</tr>
<tr>
<td>Number or procs:</td>
<td>$n \geq f + 1$</td>
<td>$n \geq 3f + 1$</td>
</tr>
<tr>
<td>Message size:</td>
<td>poly</td>
<td>poly</td>
</tr>
</tbody>
</table>

Asynchronous case:

- **Impossible** in both message passing and SHM systems even for $f = 1$ crash failures
- Reason: Correct processors never know whether a still missing message from some process will ever arrive
Synchronous Model with Crashes (I)

Admissible executions in message passing systems:

- Processors faithfully execute their algorithm in lockstep rounds, with round $k$ consisting of
  - simultaneously send round-$k$ messages
  - delivery of all round-$k$ messages
  - simultaneous single comp-event, terminating round $k$ until they possibly crash

- Message sending for round $k$ happens at same time $t$, but causally after comp-event of round $k - 1$ [or in initial state for $k = 1$]

- Subtle issue with crash failures happening at time $t$
Synchronous Model with Crashes (II)

A processor $\in F$ that crashes in round $k$

- neither sends any round $k'$ message, for any round $k' \geq k + 1$,
- nor attempts to execute the comp-event of round $k'$
- may send round $k$ messages to an arbitrary subset of its destination processors [thereby causing difficulties with inconsistent reception]

Thus, a processor that

- fails by the end of round $k - 1 \Rightarrow$ does not send anything in round $k$
- fails exactly in round $k \Rightarrow$ may send to a subset in round $k$
Asynchronous Model with Crashes

Admissible executions in message passing systems:

- Every processor faithfully executes its algorithm until it possibly crashes
- A processor that crashes during its $k$-th comp-event
  - does not execute further comp-events
  - sends the $k$-th comp-messages to an arbitrary subset of its destination processors only
- all sent messages are eventually delivered

Same for SHM model; just drop message deliveries.
A Simple Consensus Algorithm 15

Every processor $p_i$ maintains a set $V$ of values seen so far, initially $V = \{x_i\}$

- add received new values to $V$ and forward them
- proceed for $f + 1$ rounds

Pseudo-code Algorithm 15 for $p_i$, $0 \leq i \leq n - 1$:

1. Initially $V_i = \{x_i\}$

2. for $k = 1$ to $f + 1$ do  // for $f + 1$ rounds
3.   send $V_i$ to all
4.   receive $V_j$ from $p_j$ for all $j$ (including $i$)
5. $V_i := \bigcup_{j=0}^{n-1} V_j$
6. $y_i := \min(V_i)$  // decide at end of round $f + 1$
Correctness Algorithm 15

Theorem 212. Algorithm 15 is a $f$-crash resilient synchronous consensus algorithm.
Correctness Algorithm 15

**Theorem 212.** Algorithm 15 is a $f$-crash resilient synchronous consensus algorithm.

**Proof.** Termination is trivial, we thus have to show:

- **Validity:** Obvious, since every $\min(V_i)$ must be some $p_j$’s $x_j$.
- **Agreement:** It suffices to show that if $x \in V_i$ at the end of round $f + 1 \Rightarrow x \in V_j$, for any non-faulty $p_i$ and $p_j$.

Let $r$ be first round where $x$ is added to any non-faulty $p_i$’s set $V_i$.

- If $r \leq f$, then $p_i$ sends $x$ in round $r + 1 \leq f + 1$ to $p_j$, which causes $p_j$ to add $x$ to $V_j$ and we are done.
- If $r = f + 1$, there must be a chain of $f + 1$ different processors $p_{i_1}, \ldots, p_{i_{f+1}}$ along which $p_{i_1}$’s initial value $x$ was forwarded to $p_i$. Still, we have at most $f$ faulty processors, so at least one must be correct, which contradicts minimality of $r = f + 1$. 

Rigorous Proof: Round Invariants (I)

For any \( r \geq 0 \), let

- \( V^r_p \) be the value of \( V_p \) at the end of round \( r \); \( V^0_p \) is the initial value of \( V_p \)
- \( Corr^r \) be the set of processors that have not crashed by the end of round \( r \); \( Corr^0 = \{p_0, \ldots, p_{n-1}\} \)
- Assume “virtual” round \( r = 0 \), ending in initial configuration
Rigorous Proof: Round Invariants (I)

For any $r \geq 0$, let

- $V_p^r$ be the value of $V_p$ at the end of round $r$; $V_p^0$ is the initial value of $V_p$

- $Corr^r$ be the set of processors that have not crashed by the end of round $r$; $Corr^0 = \{p_0, \ldots, p_{n-1}\}$

- Assume “virtual” round $r = 0$, ending in initial configuration

**Lemma 213.** If $V_p^{R-1} = V$ for all processors $p \in Corr^{R-1}$ for some $R \geq 1$, then $V_q^r = V$ for all $q \in Corr^r$ for any round $r \geq R - 1$.

**Proof.** Trivial induction on rounds, starting with round $R - 1$.  

Let a failure-free round (ff-round) $r \geq 1$ be a round where $\text{Corr}^{r-1} = \text{Corr}^r$.

**Lemma 214.** After any ff-round $r$, it holds that $V_p^{r} = V_q^{r}$ for all $p, q \in \text{Corr}^r$.

**Proof.** Consider $v \in V_p^r$ of $p \in \text{Corr}^{r-1} = \text{Corr}^r$. Clearly, $v$ has been sent to $p$ by some processor $s$ either

- in round $k \leq r - 1$; since $p \in \text{Corr}^r \subseteq \text{Corr}^{k+1}$, $v$ is sent to all processes in $\text{Corr}^r$ in round $k + 1$.
- in round $r$, but then $s \in \text{Corr}^{r-1} = \text{Corr}^r$, such that $v$ is sent to all processors in $\text{Corr}^r$ in round $r$.

Hence, $v \in V_q^r$ for any processor $q \in \text{Corr}^r$ as well. □
**Theorem 215.** Algorithm 15 is a $f$-crash resilient synchronous consensus algorithm.
Theorem 215. Algorithm 15 is a \( f \)-crash resilient synchronous consensus algorithm.

Proof. We have to show:

- **Termination**: Obvious.
- **Validity**: Follows immediately from first lemma.
- **Agreement**: Since we have \( f + 1 \) rounds but only at most \( f \) faulty processors,
  - at least one round \( r \) must be failure-free
  - from round \( r \) on, all processors \( p \) have the same set \( V_p \) by the first lemma
  - decide on same value \( \min(V_i^{f+1}) \).

\[ \square \]
A configuration $C$ [at the end of a round] of a binary consensus algorithm is called

- **0-decided** if some correct $p_i$ has already decided 0
- **1-decided** if some correct $p_i$ has already decided 1
- **0-valent** if all decided configurations $C'$ reachable from $C$ are 0-decided
- **1-valent** if all decided configurations $C'$ reachable from $C$ are 1-decided

Classify configurations $C$ as

- **univalent** if $C$ is either 1-valent or 0-valent
- **bivalent** if both a 0-decided and a 1-decided configuration can be reached from $C$
**Some more Definitions**

Let $\alpha$, $\alpha_1$ and $\alpha_2$ be admissible executions.

- $\text{dec}(\alpha)$ denotes the unique decision value of all correct processors.

- $\alpha|p_i$ denotes $p_i$'s view of the execution, consisting of:
  - all comp and del events at $p_i$
  - $p_i$'s state in the initial configuration of $\alpha$

- $\alpha_1$ is similar to $\alpha_2$ for some non-faulty $p_i$, denoted $\alpha_1 \sim p_i \alpha_2$, if $\alpha_1|p_i = \alpha_2|p_i$

- $\alpha_1$ is indirectly similar to $\alpha_2$, denoted $\alpha_1 \approx \alpha_2$, if there are executions $\beta_k$ and correct processors $p_k$ with:
  $\alpha_1 = \beta_1 \sim p_1 \beta_2 \sim p_2 \cdots \sim p_j \beta_{j+1} = \alpha_2$
Decisions in Similar Executions

We have the following key observations:

- If $\alpha_1 \overset{p_i}{\sim} \alpha_2$ then $\text{dec}(\alpha_1) = \text{dec}(\alpha_2)$
- If $\alpha_1 \approx \alpha_2$ then $\text{dec}(\alpha_1) = \text{dec}(\alpha_2)$
Decisions in Similar Executions

We have the following key observations:

1. If $\alpha_1 \sim p_i \alpha_2$ then $\text{dec}(\alpha_1) = \text{dec}(\alpha_2)$
2. If $\alpha_1 \approx \alpha_2$ then $\text{dec}(\alpha_1) = \text{dec}(\alpha_2)$

Subsequently, we consider message-passing systems with

- $n \geq f + 2$ processors [can be extended to $n \geq f + 1$]
- at most $f \geq 0$ crash failures

and study binary synchronous consensus algorithms

- that send a message to all processors in every round
- and keep a full message history in the local state [can be dropped by reduction]

- in failure-sparse executions: At most 1 crash per round
Synchronous Configuration Trees

Consider all admissible executions $\text{exec}(C^0, \sigma)$ of a full-history synchronous algorithm

- starting from some fixed initial state $C^0$
- with at most one (additional) crash per round

All reachable configurations can be arranged in a configuration tree, with

- vertices representing (unique) configurations
- edges representing (unique) rounds + failure patterns

Vertex $C^{k-1}$ has a successor $C^k_{q,F_q}$ for

- every not-yet crashed processor $q$
- every subset $F_q$ (including $F_q = \emptyset$) of processors that do not receive $q$’s message in round $k$
Theorem 220. Any consensus algorithm $A$ for $n \geq f + 2$ processors that is resilient to $f \geq 1$ crash failures requires at least $f + 1$ rounds in some admissible execution.
**Theorem 220.** Any consensus algorithm $A$ for $n \geq f + 2$ processors that is resilient to $f \geq 1$ crash failures requires at least $f + 1$ rounds in some admissible execution.

**Proof.** The proof consists of two parts, which will be proved as independent lemmas subsequently:

- There is an $f - 1$-round (sparse) execution $\alpha_{f-1}$ that ends in a bivalent (and hence undecided) configuration.

- By extending $\alpha_{f-1}$ by one additional round, at least one correct processor is still undecided.

Hence, $f$ rounds are not enough for all correct processors to decide. \(\square\)
Lemma 221. Algorithm $\mathcal{A}$ has a bivalent initial configuration.
Bivalent Initial Configuration Lemma (I)

Lemma 221. *Algorithm A has a bivalent initial configuration.*

Proof. For the sake of contradiction, assume that all initial configurations are univalent. Clearly,

- \( I_{0*} \) where all processors \( p_i \) start with input value \( x_i = 0 \) ⇒ must be 0-valent by validity
- \( I_{1*} \) where all processors \( p_i \) start with input value \( x_i = 1 \) ⇒ must be 1-valent by validity

Hence, toggling \( x_0, x_1, \ldots \) one after the other starting from \( I_{0*} \) reveals that there is

- some 0-valent initial configuration \( I_0 \), and
- some 1-valent initial configuration \( I_1 \)

that differ in a single \( x_i \) only. \( \square \)
Bivalent Initial Configuration Lemma (II)

Proof. (cont.)

Now consider the (sparse) admissible schedule \( \sigma \) where

- \( p_i \) crashes initially and all other processors are correct
- all correct processors have decided in \( \sigma(I_0) \Rightarrow \) decision must be 0 as \( I_0 \) is 0-valent

Let \( \alpha_0 = \text{exec}(I_0; \sigma) \) be the resulting admissible execution and consider \( \alpha_1 = \text{exec}(I_1; \sigma) \).

- \( \alpha_1 \) is indistinguishable from \( \alpha_0 \) for any \( p_j \neq p_i \)
- \( \alpha_1 \xrightarrow{p_j} \alpha_0 \Rightarrow \text{dec}(\alpha_1) = \text{dec}(\alpha_0) = 0 \)

However, \( \text{dec}(\alpha_1) \) should be 1 since configuration \( I_1 \) where \( \alpha_1 \) starts is 1-valent, which provides the required contradiction. \( \square \)
Lemma 223. For each $k$, $0 \leq k \leq f - 1$, there is a $k$-round execution of $A$ that ends in a bivalent configuration.
Bivalent Successor Configuration (I)

Lemma 223. For each \( k, 0 \leq k \leq f - 1 \), there is a \( k \)-round execution of \( A \) that ends in a bivalent configuration.

Proof. By induction.

The basis \( k = 0 \) is provided by the previous lemma.

For the induction step, assume that \( \alpha_{k-1} \) is the (sparse) \( k - 1 \)-round execution ending in a bivalent configuration \( C'_{k-1} \), according to the induction hypothesis. Note: \( k - 1 \leq f - 2 \) here. 2 cases:

- There is some (sparse) 1-round extension of \( \alpha_{k-1} \) that ends in a bivalent configuration \( \Rightarrow \) we are done.

- All (sparse) 1-round extensions of \( \alpha_{k-1} \) lead to univalent configuration \( \Rightarrow \) use contradiction proof.
Bivalent Successor Configuration (II)

Proof. (cont.)

Assuming that all 1-round extensions of $\alpha_{k-1}$ lead to univalent configuration, consider two different ones:

- $\beta_k$ where no crash occurs in round $k$ and w.l.o.g. a 1-valent configuration is reached
- $\gamma_k$ where one crash occurs in round $k$ and a 0-valent configuration is reached (since $C_{k-1}$ is bivalent, $\gamma_k$ must exist).

In $\gamma_k$,

- let $p_i$ be the processor that crashes in round $k$, and
- $q_1, \ldots, q_m, 1 \leq m \leq n$, be the processors that do not get a message from $p_i$.  

$\square$
Bivalent Successor Configuration (III)

Proof. (cont.)

Now define $\alpha^j_k$, $0 \leq j \leq m$, as the one round extension of $\alpha_{k-1}$ where $p_i$ does not send a message to $q_1, \ldots, q_j$. Clearly,

1. $\alpha^0_k = \beta_k$ and reaches a 1-valent configuration $C^0_k$
2. $\alpha^m_k = \gamma_k$ and reaches a 0-valent configuration $C^m_k$

Somewhere in $C^0_k, C^1_k, \ldots, C^m_k$ there must be a switch from 1-valent to 0-valent. Let $j$ be the appropriate index, such that

1. configuration $C^j_k$ reached by $\alpha^j_k$ is 1-valent
2. configuration $C^{j+1}_k$ reached by $\alpha^{j+1}_k$ is 0-valent
3. only processor $q_{j+1}$ sees a difference between $\alpha^j_k$ and $\alpha^{j+1}_k$
Bivalent Successor Configuration (IV)

Proof. (cont.)

Since at most \( k \leq f - 1 \) processes can have crashed in any \( \alpha_k^x \),

- \( q_{j+1} \) may additionally crash at the beginning of round \( k + 1 \), without exceeding the failure bound \( f \)
- kills the only witness of the difference between \( \alpha_k^j \) and \( \alpha_k^{j+1} \)

Consider the admissible extensions \( \delta_k^j \) of \( \alpha_k^j \) and \( \delta_k^{j+1} \) of \( \alpha_k^{j+1} \) where \( q_{j+1} \) crashes at the beginning of round \( k + 1 \). For any correct \( p_\ell \),

- \( \delta_k^j \; p_\ell \; \delta_k^{j+1} \) such that the decision values must be the same
- Contradiction, since the configurations reached by \( \alpha_k^j \) and \( \alpha_k^{j+1} \) had different valences.

This confirms that some one-round extension \( \alpha_k \) of \( \alpha_{k-1} \) must indeed end in a bivalent configuration. \( \square \)
Lemma 227. If $\alpha_{f-1}$ is an $f - 1$-round (sparse) execution of $A$ that ends in a bivalent configuration $C_{f-1}$, then there is a 1-round extension in which some correct processor has not decided.
No Decision in Round $f$ (I)

Lemma 227. If $\alpha_{f-1}$ is an $f-1$-round (sparse) execution of $A$ that ends in a bivalent configuration $C_{f-1}$, then there is a 1-round extension in which some correct processor has not decided.

Proof. If there is a 1-round extension with at most one crash in round $f$ that ends in a bivalent configuration, we are done. Otherwise, consider the 1-round extensions

- $\beta_f$ where no crash occurs in round $f$ and w.l.o.g. a 1-valent configuration is reached
- $\gamma_f$ where one crash occurs in round $f$ and a 0-valent configuration is reached (since $C_{f-1}$ is bivalent, $\gamma_f$ must exist).

Let $p_i$ be the unique processor that fails in round $f$ in $\gamma_f$. $\square$
No Decision in Round $f$ (II)

Proof. (cont.)

The processor $p_i$ crashing in round $f$ fails to send a message to some processor $p_j$, which must be correct since

- $p_j$ cannot crash in round $f$ as $p_i$ does so (sparse execution)
- not all candidates $p_j$ can have crashed before round $f$ (as otherwise $\beta_f$ and $\gamma_f$ would be indistinguishable for all correct processes $\Rightarrow$ cannot lead to configurations with different valences)

Consider a third 1-round extension $\delta_f$ of $\alpha_{f-1}$ that is

- the same as $\gamma_f$, except that
- $p_i$ succeeds to send a message to some correct $p_k \neq p_j$
- Note: Such a $p_k$ must exist since $n \geq f + 2$, and $\delta_f$ may be $\gamma_f$. 

\[\square\]
No Decision in Round $f$ (III)

Proof. (cont.)

Since $\beta_f \overset{p_k}{\sim} \delta_f$ as well as $\delta_f \overset{p_j}{\sim} \gamma_f$, it follows that in $\delta_f$

- the decision of $p_k$ at the end of round $f$ can only be 1 (or undefined)
- the decision of $p_j$ at the end of round $f$ can only be 0 (or undefined)
- their decision must be the same $\Rightarrow$ cannot both be defined

Note that $\beta_f \overset{p_k}{\sim} \delta_f \nRightarrow \beta_{f+1} \overset{p_k}{\sim} \delta_{f+1}$, hence

- if $p_k$ is undefined at the end of round $f$ in $\delta_f$,
  $\Rightarrow$ it need not decide 1 in some later round following $\delta_f$ even though the configuration reached by $\beta_f$ was 1-valent
  $\Rightarrow$ proof does not contradict the possibility of $\beta_f$ reaching a 1-valent and $\gamma_f$ reaching a 0-valent configuration
Synchronous Byzantine Consensus (I)

We now increase the adverse capabilities of faulty processors:

- They need not adhere to the algorithm at all
- They can send any message, even inconsistently, to any receiver
- They can collude in an attempt to maximize their adverse power.

Also need to adapt validity condition:

- We cannot assume anything about the initial value $x_k$ of a Byzantine processor $p_k$

  **Byzantine validity**: If $x_k = v$ for all correct processors $p_k$, then $y_i = v$ for every terminated correct processor $p_i$
Synchronous Byzantine Consensus (II)

Upcoming results:

- Exponential Information Gathering (EIG) algorithm
- Phase King algorithm
- \( n \geq 3f + 1 \) lower bound for required number of processors
- \([f + 1 \text{ lower bound for number of rounds still applies}\)
Naive Approach

Recall Algorithm 15:
- Just forward all values received in round $k - 1$ in round $k$
- Decide on minimum value at the end of round $f + 1$

Problem: Byzantine faulty processor can
  - inconsistently send different values, in any round
  - “drive” any number of correct processors towards some value
  ⇒ easily violate agreement

Just replacing minimum by majority does not help ⇒ need additional ideas
EIG Algorithm (I)

Requirements and properties:
- \( n \geq 3f + 1 \)
- \( f + 1 \) rounds

Principle of operation: Trace sources of information
- Every processor \( p_i \) sends its \( x_i \) to all in the first round
- Forwarding stage: \( f \) additional rounds where every \( p_j \) forwards the information obtained in the previous round ("\( p_j \) says that \( p_k \) says that \ldots that \( p_i \) sent value \( x_i \)"")
- Decision stage: At the end of round \( f + 1 \), compute decision based on the values received in forwarding stage
EIG Algorithm (II)

Every node maintains a labeled tree data structure with $f + 2$ levels (height $f + 1$):

- The level-0 root has the empty label $\varepsilon$

- A level-$k$ node, $1 \leq k \leq f + 1$, is labeled with a unique variation (without replacement) $\pi = i_1 i_2 \cdots i_k$ of processor indices $\in \{0, \ldots, n - 1\}$

- The leaves are at level $f + 1$

- Every node at level $k < f + 1$ has degree $n - k$

- $\text{tree}_i(\pi)$ denotes the value stored in $p_i$'s tree node with label $\pi$

- A node with label $\pi = \pi' i_k$ (and the edge leading to it) corresponds to processor $p_{i_k}$ as it gets its data from $p_{i_k}$. 
**EIG Algorithm (III)**

**Forwarding stage:**
- Every $p_i$ stores $x_i$ into the root of its tree
- In round $k$, $1 \leq k \leq f + 1$, processor $p_i$
  - sends level $k - 1$ of its tree to all
  - stores in its node with label $\pi'_{i_k}$ the value $v$ received from $p_{i_k}$ from its level-$k - 1$ node with label $\pi'$ (or $v_{\perp}$ in case of no or an erroneous message)
- means “$p_{i_k}$ says that $p_{i_{k-1}}$ says that ... that $p_{i_2}$ says that $p_{i_1}$ sent $v$”

**Decision stage:**
- At the end of round $f + 1$, processor $p_i$ decides $y_i = \text{resolve}_i(\varepsilon)$
The recursive majority vote $\text{resolve}_i$ is defined as

- $\text{resolve}_i(\pi) = \text{tree}_i(\pi)$ if $\pi$ is a leaf
- $\text{resolve}_i(\pi)$ is the majority of $\text{resolve}_i(\pi'')$ for all children $\pi'' = \pi_k$ of $\pi$ (or $\nu_\perp$ if no majority exists)

$\Rightarrow$ Corresponds to building up a resolve tree that has the same leafs as the forwarding tree
EIG Algorithm (IV)

The recursive majority vote $\text{resolve}_i$ is defined as

1. $\text{resolve}_i(\pi) = \text{tree}_i(\pi)$ if $\pi$ is a leaf
2. $\text{resolve}_i(\pi)$ is the majority of $\text{resolve}_i(\pi'')$ for all children $\pi'' = \pi_k$ of $\pi$ (or $\bot$ if no majority exists)

$\Rightarrow$ Corresponds to building up a resolve tree that has the same leafs as the forwarding tree

A few additional definitions for our analysis:

1. A node $\pi$ is common if $\text{resolve}_i(\pi) = \text{resolve}_j(\pi)$ for all non-faulty $p_i$ and $p_j$
2. A subtree has a common frontier if there is a common node on every path from the root to its leaves
Lemma 237. If the subtree rooted at node $\pi$ has a common frontier, then $\pi$ is common.
Lemma 237. *If the subtree rooted at node \( \pi \) has a common frontier, then \( \pi \) is common.*

*Proof.* By induction on the level of \( \pi \). If \( \pi \) is a leaf, the statement follows directly from the definition of a common frontier.

Induction step: Assume \( \pi \) is a node at level \( \ell \), and that the lemma holds for nodes at level \( \ell + 1 \). If \( \pi \) was not common,

- every subtree rooted at a child \( \pi_k \) of \( \pi \) must have a common frontier
- since every child \( \pi_k \) has level \( \ell + 1 \), the induction hypothesis reveals that they must all be common
- All non-faulty processors resolve the same value for all children and hence for \( \pi \), i.e., \( \pi \) must be common.

\( \square \)
Lemma 238. For all tree node labels $\pi$ and correct processors $p_i, p_j, p_k$, we have $\text{tree}_i(\pi_j) = \text{tree}_j(\pi)$, and hence $\text{tree}_i(\pi_j) = \text{tree}_k(\pi_j)$. 
Lemma 238. For all tree node labels $\pi$ and correct processors $p_i, p_j, p_k$, we have $\text{tree}_i(\pi j) = \text{tree}_j(\pi)$, and hence $\text{tree}_i(\pi j) = \text{tree}_k(\pi j)$.

Proof. Since $p_j$ is correct, it faithfully sends its value $\text{tree}_j(\pi)$ to $p_i$. Since the latter is also correct, it stores this value in $\text{tree}_i(\pi j)$. $\square$
Lemma 238. For all tree node labels $\pi$ and correct processors $p_i, p_j, p_k$, we have $\text{tree}_i(\pi_j) = \text{tree}_j(\pi)$, and hence $\text{tree}_i(\pi_j) = \text{tree}_k(\pi_j)$.

Proof. Since $p_j$ is correct, it faithfully sends its value $\text{tree}_j(\pi)$ to $p_i$. Since the latter is also correct, it stores this value in $\text{tree}_i(\pi_j)$. \qed

Lemma 238. For every tree node label $\pi = \pi' j$ and correct processors $p_j, p_i$, it holds that $\text{resolve}_i(\pi) = \text{tree}_i(\pi)$ at every non-faulty $p_i$. 
Lemma 238. For all tree node labels $\pi$ and correct processors $p_i, p_j, p_k$, we have $\text{tree}_i(\pi_j) = \text{tree}_j(\pi)$, and hence $\text{tree}_i(\pi_j) = \text{tree}_k(\pi_j)$.

Proof. Since $p_j$ is correct, it faithfully sends its value $\text{tree}_j(\pi)$ to $p_i$. Since the latter is also correct, it stores this value in $\text{tree}_i(\pi_j)$. □

Lemma 238. For every tree node label $\pi = \pi'j$ and correct processors $p_j, p_i$, it holds that $\text{resolve}_i(\pi) = \text{tree}_i(\pi)$ at every non-faulty $p_i$.

Proof. By induction on the level of $\pi$, starting from the leaves:

- **Induction basis**: If $\pi$ is a leaf, the lemma holds by definition of recursive majority.

- **Induction step**: If $\pi = \pi'j$ ending in correct $p_j$ is a non-leaf, it has at most level $f$ and hence at least degree $n - f$. □
Analysis EIG Algorithm (III)

Proof. (cont.)

Induction step: If $\pi = \pi' j$ ending in correct $p_j$ is a non-leaf, it has at most level $f$ and hence at least degree $n - f$.

- Since $n \geq 3f + 1$, $\pi$ has a majority of children $\pi k$ corresponding to a correct $p_k$.
- Applying the induction hypothesis reveals, at any correct $p_i$, 
  \[ resolve_i(\pi k) = tree_i(\pi k) \]
- Since, by the previous lemma,
  \[ tree_i(\pi k) = tree_k(\pi) = tree_i(\pi) \]
  this implies $resolve_i(\pi k) = tree_i(\pi)$
- Hence, all of $\pi$’s non-faulty children and thus $\pi$ resolve to $tree_i(\pi)$ as asserted.
Analysis EIG Algorithm (IV)

**Theorem 240.** Every $\pi = \pi'j$ ending in a correct processor $p_j$ is common.

**Proof.** For correct processors $p_i$, $p_k$, our previous results establish:

1. By the last but one lemma, $\text{tree}_i(\pi) = \text{tree}_k(\pi)$
2. By the previous lemma,
   $$\text{resolve}_i(\pi) = \text{tree}_i(\pi) = \text{tree}_k(\pi) = \text{resolve}_k(\pi).$$
Theorem 241. For $n \geq 3f + 1$, EIG solves consensus in presence of up to $f$ Byzantine failures.
Theorem 241. For $n \geq 3f + 1$, EIG solves consensus in presence of up to $f$ Byzantine failures.

Proof. Validity: If all non-faulty processors start with the same input $v$, a majority of children $j$ of the root at any non-faulty $p_i$ satisfy $\text{resolve}_i(j) = \text{tree}_i(j) = \text{tree}_j(\varepsilon) = v$ by our lemmas. Hence, $\text{resolve}_i(\varepsilon) = v$ as well.

Agreement: Each path from a child of the root to a leaf involves $f + 1$ nodes that correspond to different processors. Hence, at least one processor on every path from the root to the leaves is correct \(\Rightarrow\) the corresponding node is common by Theorem 240. The root has a common frontier.

Hence, the root must be common, which completes our proof. \qed
Less Costly Alternative to EIG?

Recall: The EIG algorithm has

- optimal time complexity \((f + 1)\) rounds
- optimal resilience \(n \geq 3f + 1\)
- exponential message complexity

Alternative idea: Don’t trace sources of information but just disseminate values as in the crash-tolerant Algorithm 15

- decide on majority value if “overwhelming majority” exists
- rely on a single correct processor’s value otherwise
Phase King Algorithm (I)

Operates in \( f + 1 \) phases of 2 rounds each
- First round: Disseminate current preference values system-wide
- Second round: Use the rotating coordinator principle to select single correct processor (the “king”) if no “overwhelming majority” exists

The Phase King algorithm
- solves consensus with polynomial message complexity
- with sub-optimal round complexity \( (2(f + 1)) \)
- and sub-optimal resilience \( (n \geq 4f + 1) \)
Phase King Algorithm (II)

Pseudo-code Algorithm 16 for $p_i$, $0 \leq i \leq n - 1$:

1. $v := x$  // Init preference to own proposed value

2. for $k = 1$ to $f + 1$ do  // for $f + 1$ phases (2 rounds each)

3. /* round 2k-1 */

4. send $\langle v \rangle$ to all processors

5. receive $\langle v_j \rangle$ from all $p_j$

6. $maj :=$ majority among $v_j$ ($v_\perp$ if none)

7. $mult :=$ multiplicity of $maj$ among $v_j$

8. /* round 2k */

9. if $i = k$ then send $\langle maj \rangle$ to all processors

10. receive $\langle king-maj \rangle$ from $p_k$ ($v_\perp$ if none)

11. if $mult > n/2 + f$ then $v := maj$ else $v := king-maj$

12. $y := v$  // decide at end of phase $f + 1$
Analysis of Phase King Algorithm (I)

We say

\( p_i \) prefers value \( v \) at the beginning of phase \( k \) [= the end of phase \( k - 1 \), with phase 0 representing the initial configuration] if

\[ v_{i}^{2k-2} = v \] at the end of round \( 2k - 2 \)
We say

- $p_i$ prefers value $v$ at the beginning of phase $k$ [$= \text{the end of phase } k - 1$, with phase 0 representing the initial configuration] if

- $v_i^{2k-2} = v$ at the end of round $2k - 2$

Lemma 245 (Persistence of agreement). *If all correct processors prefer $v$ at the beginning of phase $1 \leq k \leq f + 1$, then they all prefer $v$ at the end of phase $k$.*
Analysis of Phase King Algorithm (I)

We say

\[ p_i \text{ prefers value } v \text{ at the beginning of phase } k \text{ [= the end of phase } k - 1, \text{ with phase 0 representing the initial configuration]} \text{ if} \]

\[ v_i^{2k-2} = v \text{ at the end of round } 2k - 2 \]

Lemma 245 (Persistence of agreement). If all correct processors prefer \( v \) at the beginning of phase \( 1 \leq k \leq f + 1 \), then they all prefer \( v \) at the end of phase \( k \).

Proof. By the code,

- every processor receives at least \( n - f \) copies of \( v \) in the first round of phase \( k \)
- \( n - f > n/2 + f \) since \( n > 4f \), so all processors prefer \( v \) at the end of phase \( k \)
Analysis of Phase King Algorithm (II)

Persistence of agreement already implies

- Validity
- Termination
Persistence of agreement already implies

- Validity
- Termination

For agreement: Since there are $f + 1$ phases

- Every phase has a different king

$\Rightarrow$ There is at least one phase $g$ with a correct king

It only remains to be shown that all correct processors prefer same value at the end of phase $g$ . . .
Lemma 247. Let $g$ be a phase with a correct king $p_g$. Then all correct processors finish phase $g$ with the same preference value.
Analysis of Phase King Algorithm (III)

Lemma 247. Let $g$ be a phase with a correct king $p_g$. Then all correct processors finish phase $g$ with the same preference value.

Proof. 2 exhaustive cases:

- All correct processors $p_j$ use $\text{king-maj}_j$ for their preference. Since $p_g$ is correct, $\text{king-maj}_j$ must be the same at all $p_j$.
- Suppose some $p_i$ uses $\text{maj}_i$ for its preference, then
  - $p_i$ must have received $> n/2 + f$ messages containing $\text{maj}_i$ in the first round
  - every other processor $p_j$, including $p_g$, must have received $> n/2$ of those messages as well, and thus set $\text{maj}_j = \text{maj}_i$
  $\Rightarrow$ every processor $p_j$ sets $\text{king-maj}_j = \text{maj}_i$ as well
  $\Rightarrow$ every processor $p_j$ assigns $v_j = \text{maj}_i$. 

\[\square\]
Lower Bound for Number of Processors

Contradicting intuition,

- a majority of correct processors is NOT sufficient
- EIG needed $n \geq 3f + 1$ processors
Lower Bound for Number of Processors

Contradicting intuition,
- a majority of correct processors is NOT sufficient
- ElG needed $n \geq 3f + 1$ processors

Why is this?
Lower Bound for Number of Processors

Contradicting intuition,
- a majority of correct processors is NOT sufficient
- EIG needed \( n \geq 3f + 1 \) processors

Why is this?

Recall illustrating example:
- Consider \( f = 1 \)
- Try to synchronize the clocks of 3 processors \( p_0, p_1, p_2 \), one of which (say, \( p_0 \)) is Byzantine

Problem: \( p_0 \) may send different information to \( p_1 \) and \( p_2 \).
Lower Bound for $f = 1$ (I)

**Theorem 249.** There is no algorithm that solves consensus in presence of a single Byzantine failure in a system of 3 processors.
Lower Bound for $f = 1$ (I)

**Theorem 249.** There is no algorithm that solves consensus in presence of a single Byzantine failure in a system of 3 processors.

*Proof.* Suppose there is some binary consensus algorithm $\mathcal{A} = (A, B, C)$ for three processors

- $p_0$ executes code $A$, $p_1$ and $p_2$ execute $B$ and $C$, respectively
- arrange six non-faulty processors in a ring $(A, B, C, A, B, C)$
- assign the input values $(1, 1, 0, 0, 0, 1)$ to those processors and let them execute their algorithm

Clearly, the resulting execution $\alpha_6$ does not necessarily solve consensus in this six processor system, BUT ...
Lower Bound for $f = 1$ (II)

Proof. (cont.)

$\alpha_6$ ensures that

- every processor has a fixed, well-defined behavior
- every algorithm locally perceives a system that looks like a three-processor system (with one Byzantine faulty processor)

For example, $\alpha^{p_0, p_1}_6 = \alpha_6 | \{p_0, p_1\} = \alpha_3^{1} | \{p_0, p_1\}$

- A single neighbor is “split” on two processors $\Rightarrow$ acts Byzantine w.r.t. the others
- Every two non-split processors $(p_0, p_1)$ should reach agreement
Proof. (cont.)
As a consequence,

- every two “consecutive” three-processor systems have one processor common, like $p_1$ in

$$\alpha_6^{p_0,p_1} = \alpha_6|\{p_0, p_1\} = \alpha_3^1|\{p_0, p_1\}$$
$$\alpha_6^{p_1,p_2} = \alpha_6|\{p_1, p_2\} = \alpha_3^2|\{p_1, p_2\}$$

- $p_1$ has the same view $\alpha_3^1|p_1 = \alpha_3^2|p_1$ in both $\Rightarrow$ same decision

Consider $\alpha_3^1$, $\alpha_3^2$ and $\alpha_3^3$:

- Validity enforces decision 1 in $\alpha_3^1$ and 0 in $\alpha_3^3$
- Unique decision in $\alpha_3^2$ should be the same as in both $\alpha_3^1$ and $\alpha_3^3$ $\Rightarrow$ Contradiction
Lower Bound for arbitrary $f$

**Theorem 252.** There is no algorithm that solves consensus in presence of $f$ Byzantine failure in a system of $n \leq 3f$ processors.
Lower Bound for arbitrary $f$

**Theorem 252.** There is no algorithm that solves consensus in presence of $f$ Byzantine failure in a system of $n \leq 3f$ processors.

**Proof.** We use a simple simulation (reduction) argument:

- Assume such an algorithm $\mathcal{A}$ exists
- Consider a system of 3 processors, where each processor executes $\mathcal{A}$ for at most $n/3$ “sub-processors” (e.g. in round robin order)
- Let a processor terminate if any of its sub-processor algorithms terminate, returning the latter’s decision

Obviously,

- If $\leq 1$ processor is Byzantine, $\leq n/3$ sub-processors are
- $\mathcal{A}$ should achieve consensus $\Rightarrow$ this contradicts the 3-processor impossibility, however.
Consensus in Asynchronous Systems

Asynchronous systems of \( n \) processors:

- Processors and communication are asynchronous
- At most \( f \) processors may fail by crashing, i.e.,
  - work correctly up to some comp-event \( \phi_k \)
  - do not execute further comp-events \( \phi_l \) with \( l > k \)
- MP: send the \( k \)-th comp-message to an arbitrary subset of destination processors only
- MP: Communication is completely reliable

Will show: Consensus is impossible both in SHM and MP systems even if \( f = 1 \)
Overview of Upcoming Results

Wait-free case $f = n - 1$

- Wait-free $\simeq$ algorithms must not wait for messages since they could block
- Impossibility easier to show since many faulty processes

General case $f = 1$

- Same as wait-free case for $n = 2$
- Show impossibility for arbitrary $n$ by clever reduction

Above results shown for SHM systems.

- Impossibility for MP systems by simple reduction
- [Well-known direct proof by Fischer, Lynch & Paterson]
Asynchronous Bivalence Proofs: Definitions

A configuration $C$ in an admissible execution is called

- **0-decided** if some (correct or faulty) $p_i$ has already decided 0
- **1-decided** if some $p_i$ has already decided 1
- **0-valent** if all decided configurations $C'$ reachable from $C$ are 0-decided
- **1-valent** if all decided configurations $C'$ reachable from $C$ are 1-decided

Classify configurations $C$ as

- **univalent** if $C$ is either 1-valent or 0-valent
- **bivalent** if both a 0-decided and a 1-decided configuration can be reached from $C$
Asynchronous Configuration Trees (SHM)

Consider all admissible executions \( \text{exec}(C^0, \sigma) \) of an asynchronous wait-free SHM algorithm

- starting from some fixed initial state \( C^0 \)
- with arbitrary infinite schedule \( \sigma \) (no restriction)

All reachable configurations can be arranged in a configuration tree, with

- vertices representing (unique) configurations [encode number of steps taken by every processor in configuration]
- edges represent steps
- every vertex \( C \) has exactly \( n \) successors \( C_i = i(C') \), \( 0 \leq i \leq n - 1 \), corresponding to \( p_i \) taking the next step
Lemma 257. Let $C_1$ and $C_2$ be two univalent configurations of a wait-free binary consensus algorithm. If $C_1 \sim p_i C_2$ for some correct $p_i$, then both configurations have the same valence.
Preparation Lemma

**Lemma 257.** Let $C_1$ and $C_2$ be two univalent configurations of a wait-free binary consensus algorithm. If $C_1 \sim p_i C_2$ for some correct $p_i$, then both configurations have the same valence.

**Proof.** Consider an infinite $p_i$-only schedule $\sigma$ starting from $C_1$:

- $p_i$ must decide in $\text{exec}(C_1, \sigma)$ because algorithm is wait-free
- Since $C_1$ is $v$-valent for some $v \in \{0, 1\}$, the decision must be $v$

Now apply $\sigma$ to $C_2$:

- Yields a feasible execution since $p_i$ starts from same configuration
- $p_i$ must also decide in $\text{exec}(C_2, \sigma)$ and its decision must also be $v$. 

$\square$
Lemma 258. Every wait-free binary consensus algorithm has a bivalent initial configuration.
Bivalent Initial Configuration

Lemma 258. Every wait-free binary consensus algorithm has a bivalent initial configuration.

Proof. Consider the following initial configurations:

- $I_0$ where all processors $p_i$ start with input value $x_i = 0 \implies$ must be 0-valent by validity
- $I_1$ where all processors $p_i$ start with input value $x_i = 1 \implies$ must be 1-valent by validity

Now consider initial configuration $I_{01}$ where $x_0 = 0$ and $x_i = 1$ for $1 \leq i \leq n - 1$. Assume, by way of contradiction, that it is univalent:

- $I_{01} \not{\sim} I_0 \implies I_{01}$ must be 0-valent by preparation lemma
- $I_{01} \not{\sim} I_1 \implies I_{01}$ must be 1-valent by preparation lemma

$\implies$ Contradiction; so $I_{01}$ must be bivalent.
Lemma 259. Every bivalent configuration of a wait-free binary consensus algorithm has at least one bivalent successor configuration.
Lemma 259. Every bivalent configuration of a wait-free binary consensus algorithm has at least one bivalent successor configuration.

Proof. Every configuration $C$ has exactly $n$ possible successor configurations $C_k$, depending on which of $p_0, \ldots, p_{n-1}$ takes the next step.

- Assume, by way of contradiction, that all $C_k$ are univalent.
- Since $C$ is bivalent, there must be $i$ and $j$ such that $C_i = i(C)$ and $C_j = j(C)$ are 0-valent and 1-valent, respectively.

Distinguish 2 possible cases...
Bivalent Successor Configuration (II)

Proof. (cont.) Distinguish 2 possible cases:

- If the steps $i$ and $j$ commute (read/write different registers or read the same one), $i(j(C')) = j(i(C')) \Rightarrow i(C')$ and $j(C')$ cannot have different valences

- If $i$ writes some register and $j$ reads or writes it, consider $i(C')$ and $i(j(C'))$:
  - $i(j(C'))$ is 1-valent since $j(C')$ is 1-valent
  - $i(C')$ is 0-valent
  - $i(C') \sim^p i(j(C')) \Rightarrow$ should have same valence by preparation lemma.

\[\square\]
Theorem 261. There is no SHM wait-free binary consensus algorithm for \( n \) processors.
Impossibility Wait-Free Consensus

**Theorem 261.** There is no SHM wait-free binary consensus algorithm for \( n \) processors.

**Proof.** We know from earlier lemmas:

- There is a bivalent initial configuration
- Every bivalent configuration has at least one bivalent successor configuration

Hence there is at least one non-terminating execution. \( \square \)
Impossibility $1$-resilient Consensus?

The above impossibility proof was easy. Why?

- Wait-free property ($f = n - 1$) gives adversary much power
- Configuration tree has simple structure

How to make things more complicated?

- Non-trivial admissibility conditions make configuration tree complex (not “closed”)
  - Could adapt SHM bivalence proof for $f = 1$ (using schedules incorporating round robin exec.)
  - MP systems further complicated by message delivery requirement: Fischer, Lynch and Patterson’s famous proof even more complex
Consensus Impossibility for $f = 1$

Alternative solution: Use (clever) reduction:

- Assume that there is a $n$-processor consensus algorithm $\mathcal{A}$ that can cope with $f = 1$ crashes.
- Use $\mathcal{A}$ to construct a 2-processor consensus algorithm that can cope with a single crash, by letting:
  - Simulating processors $p_0, p_1$ simulate the execution of
  - Simulated processors $q_0, \ldots, q_{n-1}$

Naive solution:

- Let $p_0, p_1$ simulate $n/2$ simulated processors each.
- Does not work, since crash of simulating processor would result in $f = n/2$. 

182.702 Distributed Algorithms (Prof. Schmid), http://ti.tuwien.ac.at/ecs/teaching/courses/valg) – p. 263/315
Principle of BG Simulation (I)

Both simulating processors \( p_0, p_1 \) asynchronously execute code for all \( q_0, \ldots, q_{n-1} \) in round-robin order:

- W.l.o.g. code of any \( q_j \) structured as sequence of (non-atomic) steps. The \( k \)-th step of \( q_j \), accessing a SHM variable at \( q_\ell \), executed by \( p_i \) consists of
  - reading \( q_j \)'s \( k-1 \)-state from SHM variable \( Q_j[k-1] \), and reading \( q_\ell \)'s last state \( h \) from \( Q_\ell[h] \)
  - performing the state transition of \( q_j \)
  - writing \( q_j \)'s entire new state into \( q_j \)'s dedicated SHM variable \( SQ^i_j[k] \) ("suggestion")

- For every step of \( q_i \), the faster \( p_i \) wins in determining the step’s global result \( Q_j[k] := SQ^i_j[k] \)
Principle of BG Simulation (II)

Determination of winner for $q_j$’s $k$-th step:

- Simulating $p_i$ first writes own suggestion $SQ^i_j[k]$, and then checks whether other simulating processor $p_{1-i}$ has not yet written its suggestion $SQ^{1-i}_j[k]$

- Let $flag[0]$ and $flag[1]$ be the boolean (or $\perp$) results of those checks for $i = 0$ and $i = 1$, respectively. If
  - $flag[i] = \text{true}$ and $flag[1-i] = \text{false}$ then winner is $p_i$
  - If $flag[i] = \text{false}$ and $flag[1-i] = \text{false}$ then winner is, say, always $p_0$
  - The case $flag[i] = \text{true}$ and $flag[1-i] = \text{true}$ is impossible by construction (both write before read!)

- Writing before reading each other implements wait-free ordering of events [impossible in message passing!]
Observation 1:

- If \( \text{flag}[i] = \text{true} \), then \( p_i \) is always winner
- If simulating processor \( p_i \) is fast, in the sense that it sets \( \text{flag}[i] \) before \( p_{1-i} \) writes \( SQ_{j-i}^{i}[k] \), then \( \text{flag}[i] = \text{true} \)

Observation 2:

- If \( \text{flag}[0] = \text{false} \), then winner depends on \( \text{flag}[1] \)
  - winner is \( p_0 \) if \( \text{flag}[1] = \text{false} \)
  - winner is \( p_1 \) if \( \text{flag}[1] = \text{true} \)
- \( p_0 \) can be **blocked** from executing further steps for \( q_j \) if \( p_1 \) crashes after writing \( SQ_{j}^{1}[k] \) but before assigning \( \text{flag}[1] \)
Principle of BG Simulation (IV)

Both in original algorithm and in BG simulation: Different times for
- reading the last state $h$ of $q_\ell$ from $Q_\ell[h]$
- “writing” $Q_j[k]$ (= determining winner)

Subtle problem: Non-atomicity of simulated steps of $q_j$
- A single step of the algorithm executed by $q_j$ is atomic (zero-time)
- A single step of $q_j$ in the simulation is non-atomic

Correctness of simulation requires proof that reading last state $h$ of $q_\ell$ from $Q_\ell[h]$ is consistent with last writing of $q_\ell$:
- Happens after writing $Q_\ell[h]$ but before writing $Q_\ell[h + 1]$.
- Follows from Lemma 5.22 in the textbook.
Principle of BG Simulation (V)

Resulting 2-process consensus algorithm at simulating processors $p_0, p_1$:

- Initial configuration of $q_j$ at $p_i$ takes $p_i$’s input value
- Note: $q_j$’s state at $p_0$ and $p_1$ are different if $x_0 \neq x_1$. This does not harm, however, since only one wins
- $p_i$’s consensus algorithm terminates if any simulated $q_j$’s consensus algorithm terminates
- Result is this $q_j$’s output value

This algorithm should be able to tolerate a single crash . . .
Principle of BG Simulation (VI)

Why does this result in an admissible execution for the simulated algorithm?

- Every simulated processor $q_j$ performs infinitely many steps if both $p_0$ and $p_1$ are alive.
- At most one of $p_0$ and $p_1$ may crash while it executes some $q_j$’s step.
- Only this $q_j$’s algorithm may block forever, all other $q_\ell$ with $\ell \neq j$ execute infinitely many steps on the remaining simulating processor.

**Theorem 269.** *There is no $n$ processor consensus algorithm for R/W asynchronous SHM that can tolerate even a single crash failure.*

**Proof.** See textbook for detailed proofs.
Theorem 270. There is no $n$ processor consensus algorithm for asynchronous message passing systems that can tolerate even a single crash failure.
Impossibility MP Consensus (I)

Theorem 270. There is no \( n \) processor consensus algorithm for asynchronous message passing systems that can tolerate even a single crash failure.

Proof. We again use reduction, by simulating an MP system atop of a SHM system:

- For every ordered pair of processors, there is a single-writer single-reader R/W link register (unbounded range)

- Sender appends new message to prior content of all outbound link registers

- Receiver polls all inbound link registers in round-robin fashion to get new messages

- Additional receive delay does not matter since we are dealing with asynchronous system
Proof. (cont.)

If there was a MP consensus algorithm $\mathcal{A}$ that tolerates a single crash,

\begin{itemize}
  \item this simulation in conjunction with $\mathcal{A}$ would yield a SHM consensus algorithm that tolerates a single crash
  \item such a SHM algorithm does not exist $\Rightarrow$ contradiction.
\end{itemize}

$\Box$
Causality
Causality of Events in MP Systems

A single execution $\phi^1, \phi^2, \ldots$ imposes a total order of events
- usually not the only possible execution of an algorithm
- loses *causality information* since it also orders independent events

Consider the *space-time diagram* of an execution, which
- contains only comp-events (that may send and/or receive messages)
- shows end-to-end delays only, i.e., hides del-events

Example of independent events:
- comp-event $\phi^1_0$ at processor $p_0$ sending message $m$
- comp-event $\phi^3_2$ at processor $p_2$ sending message $m'$
Happened-Before Relation (MP)

Event $\phi$ happens before event $\phi'$ in execution $\alpha$, denoted as $\phi \xrightarrow{\alpha} \phi'$, if either

- $\phi$ and $\phi'$ are comp-events by the same processor and $\phi$ occurs before $\phi'$
- $\phi$ is comp-event where message $m$ is sent and $\phi'$ is comp-event where $m$ is received
- there is some event $\phi''$ such that $\phi \xrightarrow{\alpha} \phi''$ and $\phi'' \xrightarrow{\alpha} \phi'$

The happened-before relation

- captures (possible) internal causality
- allows to identify independent (concurrent) events:

$$
\phi \parallel_{\alpha} \phi' \iff (\phi \neq \phi') \land (\phi \not\xrightarrow{\alpha} \phi') \land (\phi' \not\xrightarrow{\alpha} \phi)
$$
Some more Definitions

Definition 275. Given an execution fragment $\alpha = \text{exec}(C, \sigma)$ [possibly: involving the comp-events only], a permutation $\pi$ of $\sigma$ is a causal shuffle if

- all processors have the same view: $\sigma|_{p_i} = \pi|_{p_i}$ for all $0 \leq i \leq n - 1$
- a message is received after it is sent in $\pi$
Some more Definitions

**Definition 275.** Given an execution fragment $\alpha = \text{exec}(C, \sigma)$ [possibly: involving the comp-events only], a permutation $\pi$ of $\sigma$ is a causal shuffle if

- all processors have the same view: $\sigma|_{p_i} = \pi|_{p_i}$ for all $0 \leq i \leq n - 1$
- a message is received after it is sent in $\pi$

**Lemma 275.** Given some execution fragment $\alpha = \text{exec}(C, \sigma)$,

- any total ordering of the events in $\sigma$ that is consistent with the happens-before relation of $\alpha$ is a causal shuffle
- for any causal shuffle $\pi$ of $\sigma$, $\alpha' = \text{exec}(C, \pi)$ is an execution fragment that is similar to $\alpha$ (i.e., similar for every processor).
How can Processors Observe Causality?

By **timestamping** events, using either
- [High-resolution] real-time clocks
- Logical clocks ("Lamport clocks")
- Vector clocks

**Real-time and logical clocks**
- ensure $\phi \rightarrow \phi' \Rightarrow TS(\phi) < TS(\phi')$
- do not fully capture causality since $TS(\phi) < TS(\phi') \not\Rightarrow \phi \rightarrow \phi'$
- lack a **gap detection property**: If $TS(\phi) < TS(\phi')$, is there some $\phi''$ with $\phi \rightarrow \phi'' \rightarrow \phi'$?
- Both problems can be solved by using vector clocks
Logical Clocks (1)

Lamport Clocks: Every process maintains an integer variable $LT$ that is used for timestamping events and messages

- Initially, $LT := 0$
- $LT$ is updated in each comp-event $\phi$ to $LT(\phi)$ as follows:
  - If a message $m$ with timestamp $TS(m)$ is received in $\phi$, $LT(\phi) := \max\{LT, TS(m)\} + 1$
  - If no message is received in $\phi$, then $LT(\phi) := LT + 1$
  - If message $m$ is sent in $\phi$, then $m$ gets timestamp $TS(m) = LT(\phi)$ (after updating)
Logical Clocks (2)

Theorem 278. Let $\alpha$ be an execution and $\phi$, $\phi'$ be two comp-events in $\alpha$. If $\phi \xrightarrow{\alpha} \phi'$ then $LT(\phi) < LT(\phi')$. 
Theorem 278. Let $\alpha$ be an execution and $\phi, \phi'$ be two comp-events in $\alpha$. If $\phi \xrightarrow{\alpha} \phi'$ then $LT(\phi) < LT(\phi')$.

Proof. We only have to check all cases of the happened-before relation:

- If $\phi$ and $\phi'$ occur on the same processor, $LT(\phi) < LT(\phi')$ holds since logical time is monotonically increasing at every processor.
- If $\phi$ sends message $m$ and $\phi'$ receives $m$, then $LT(\phi')$ is at least one larger than $LT(\phi)$.
- $LT(\phi) < LT(\phi')$ for events that depend transitively on each other follows from transitivity of $<$. 

□
Principle Vector Clocks

Employ elaborate timestamps $VC(\phi)$ of event $\phi$:

- Reflexive closure $\rightarrow_r$ of happens-before:
  \[
  \phi \rightarrow_r \phi' \iff (\phi \rightarrow \phi') \lor (\phi = \phi')
  \]

- Use entire causal past $(\downarrow \phi) = \{\phi' | \phi' \rightarrow_r \phi\}$ of event $\phi$ as its timestamp $VC(\phi)$

- Encode $VC(\phi)$ as a vector of $n$ integers, the $i$-th component holding the index $c_i$ of the last comp-event $\phi_i^{c_i}$ at $p_i$ with $\phi_i^{c_i} \rightarrow_r \phi$

Define partial ordering of $VC$:

- $VC \leq VC'' \equiv VC[k] \leq VC''[k]$ for $1 \leq k \leq n$,
- $VC < VC'' \equiv (VC \neq VC'') \land (VC \leq VC'')$,
- $VC, VC''$ incomparable if $(VC \not\leq VC'') \land (VC'' \not\leq VC)$
Implementing Vector Clocks

Every process maintains $VC_i = (c_1, \ldots, c_n)$ [initially $VC_i = (0, \ldots, 0)$] used for timestamping events

- $VC_i$ is updated in each comp-event $\phi_i$ at $p_i$ to $VC(\phi_i)$:
  - If message $m$ with timestamp $TS(m)$ is received in $\phi_i$, then
    - $\forall j \neq i : VC_i(\phi_i)[j] := \max\{VC_i[j], TS(m)[j]\}$
    - $VC_i(\phi_i)[i] := VC_i[i] + 1$ (could also use $\max$ here)
  - if no message is received in $\phi_i$, then
    - $VC_i(\phi_i)[i] := VC_i[i] + 1$
  - if message $m$ is sent in $\phi_i$, then $m$ is timestamped with $VC_i(\phi_i)$ (after updating)

- Abbreviate $VC(\phi) = VC_i(\phi)$, where $p_i$ is the processor where $\phi$ occurs
Properties Vector Clocks (I)

Basic properties:

- $VC_i(\phi_i)[i]$ holds number of events at $p_i$ up to and including $\phi_i$

- $VC_i(\phi_i)[j], j \neq i$, holds
  - the number of events at $p_j$ that causally precede $\phi_i$
  - the index $VC_j(\phi_j)[j]$ of the last event $\phi_j$ at $p_j$ that causally precedes $\phi_i$

- If $VC_i(\phi_i)[i] > VC_j(\phi_j)[i]$, then $\phi_i \not\rightarrow_r \phi_j$

- If $VC_i(\phi_i)[i] \leq VC_j(\phi_j)[i]$, then $\phi_i \rightarrow_r \phi_j$ [since $VC_j$ can learn its $i$-th coordinate only via a chain of messages leading from $\phi_i$ to $\phi_j$]

- $|(\downarrow \phi_i)| = \sum_{j=1}^{n} VC_i(\phi_i)[j]$ is the total number of events that causally precede $\phi_i + 1$ [due to reflexivity].
Properties Vector Clocks (II)

**Theorem 282.** Let $\alpha$ be an execution and $\phi$, $\phi'$ be two comp-events in $\alpha$. Vector clocks satisfy the following properties:

- **Strong clock condition:** $\phi \xrightarrow{\alpha} \phi' \iff V C(\phi) < V C(\phi')$

- **Simple strong clock condition (if $\phi_i$ and $\phi_j$ occur at different processors $p_i \neq p_j$):** $\phi_i \xrightarrow{\alpha} \phi_j \iff V C_i(\phi_i)[i] \leq V C_j(\phi_j)[i]$

- **Concurrency:**
  $$\phi |\alpha\phi' \iff (V C(\phi) \not\leq V C(\phi')) \land (V C(\phi') \not\leq V C(\phi))$$
  (incomparable vector clock timestamps)

- **Simple concurrency (if the events $\phi_i$ and $\phi_j$ occur at different processors $p_i \neq p_j$):** $\phi_i |\alpha\phi_j \iff (V C_i(\phi_i)[i] > V C_j(\phi_j)[i]) \land (V C_j(\phi_j)[j] > V C_i(\phi_i)[j])$
**Properties Vector Clocks (III)**

**Proof.** The direction $\phi \xrightarrow{\alpha} \phi' \Rightarrow VC(\phi) < VC(\phi')$ follows easily from applying the definition of $VC$ to the three cases of the happened-before relation (similar to the proof of the logical clocks).

To show $VC(\phi) < VC(\phi') \Rightarrow \phi \xrightarrow{\alpha} \phi'$, we assume $VC(\phi) < VC(\phi')$ but $\phi \not\xrightarrow{\alpha} \phi'$ and distinguish 2 cases:

1. If $\phi' \xrightarrow{\alpha} \phi$, direction $\Rightarrow$ of the strong clock condition (see above) reveals $VC(\phi') < VC(\phi)$, which contradicts our assumption.

2. If $\phi || \phi'$ are concurrent, $\phi = \phi_i$ and $\phi' = \phi_j$ at $p_i \neq p_j$ and $VC(\phi_i) < VC(\phi_j)$
   - If $VC(\phi_i)[i] = \ell$, then $VC(\phi_j)[i] < \ell$ since otherwise $\phi_i \in (\downarrow \phi_j)$, which contradicts $\phi_i || \phi_j$.
   - Still, $VC(\phi_j)[i] < \ell = VC(\phi_i)[i]$ contradicts $VC(\phi_i) < VC(\phi_j)$. 
Proof. (cont.)

Finally,

- the proof of the simple strong clock condition is a trivial adaption of the above proof
- the concurrency properties are just the negations of the (simple) strong clock condition applied to $\phi \not\xrightarrow{\alpha} \phi'$ \land $\phi' \not\xrightarrow{\alpha} \phi$. 

\[\square\]
Properties Vector Clocks (V)

Weak gap detection property:

- For \( k \neq j \), suppose \( VC_i(\phi_i)[k] < VC_j(\phi_j)[k] \)

Then, \( \phi_j \) must have seen some event \( \phi_k \) not seen by \( \phi_i \), i.e., \( \phi_k \not\in (\downarrow \phi_i) \) but \( \phi_k \in (\downarrow \phi_j) \)

Hence, \( VC_i(\phi_i)[k] < VC_j(\phi_j)[k] \) implies that \( \exists \phi_k \) such that \( \neg(\phi_k \xrightarrow{\alpha} \phi_i) \land (\phi_k \xrightarrow{\alpha} \phi_j) \)

Weak gap detection property does not allow to conclude \( \phi_i \xrightarrow{\alpha} \phi_k \xrightarrow{\alpha} \phi_j \) in general

**BUT:** If \( i = k \), i.e., \( \phi_k := \phi_i' \), we have \( \neg(\phi_i' \xrightarrow{\alpha} \phi_i) \Rightarrow \phi_i \xrightarrow{\alpha} \phi_i' \). Hence, \( \phi_i \xrightarrow{\alpha} \phi_i' \xrightarrow{\alpha} \phi_j \) !

Consequently, if \( VC_i(\phi_i)[i] < VC_j(\phi_j)[i] \) for \( i \neq j \), then some event from \( p_i \) is still missing at \( p_j \)
Causally Ordered Broadcast (I)

Implement causal broadcast primitive atop standard unreliable message passing communication. At $p_i$:

- Top interface: $bc\text{-}send_i(M)$ and $bc\text{-}recv_i(M, j)$
- Bottom interface: $send_i(M, j)$ and $recv_i(M, j)$

Basic idea:

- Timestamp messages with $VC_i$ of sender process $p_i$
- Before a message $M_j$ with timestamp $V$ (received via $recv_i(M_j, j)$ from sender $p_j \neq p_i$) is delivered (by triggering $bc\text{-}recv_i(M_j, j)$): Check whether there is a causally preceding message still in transit
- Use weak gap detection property: Call $bc\text{-}recv_i(M, j)$ if $VC_i[\ell] \geq V[\ell]$ for all $\ell \neq j$, and $VC_i[j] = V[j] - 1$
Causally Ordered Broadcast (II)

Implementation details:

- Maintain “special-purpose” \( VC_i \) at \( p_i \), where
  - only \( bc\text{-}send_i(M) \) increments \( VC_i[i] \)
  - only \( bc\text{-}recv_i(M, j) \) increments \( VC_i[j] \)

- Timestamp messages with \( VC_i \) at sender \( p_i \) before sending via bottom layer interface

- Maintain a set of pending messages received via the bottom layer interface but not yet delivered
Pseudo-Code Causally Ordered Broadcast

Code for processor \( p_i \), \( 0 \leq i \leq n - 1 \):

1. \( VC := (0, \ldots, 0) \); \( pending := \emptyset \) /* Initialization */

2. When \( bc\text{-}send(M) \) occurs:
   3. \( VC[i] := VC[i] + 1 \) // Increment own component \( i \)
   4. trigger \( bc\text{-}recv(M, i) \) // local delivery
   5. trigger \( send(\langle M, VC \rangle, j) \) to every \( j \neq i \)

6. When \( recv(\langle M_j, V \rangle, j) \) occurs:
   7. \( pending := pending \cup \{ \langle M_j, V \rangle \} \)

8. When \( (\langle M_j, V \rangle \in pending) \) where
   9. \( (V[j] = VC[j] + 1) \wedge (\forall \ell \neq i, j : V[\ell] \leq VC[\ell]) \)
   10. \( pending := pending \setminus \{ \langle M_j, V \rangle \} \)
   11. \( VC[j] := VC[j] + 1 \) Increment remote component \( j \)
A Note on External Causality

Consider three processes $p_0, p_1, p_2$ in a distributed control system for a steam pipe:

- $p_0$ detects “pipe rupture” and sends message $m$ to $p_2$
- $p_1$ detects “pressure drop” in pipe and sends alarm message $m'$
- $p_2$ gets $m'$ and decides to apply heat, before it gets $m$

Happened-before relation captures internal causality only:

- Actual message delivery $m', m$ indicates “pressure drop” $→$ “apply heat” $→$ “pipe rupture”
- In reality “pipe rupture” $→$ “pressure drop” due to external causality $⇒$ delivery order should have been $m, m'$. 
Vector Clocks Memory Complexity

Vector clocks are powerful, but quite expensive in terms of memory overhead $O(n)$

Question: Can we do better?
Vector Clocks Memory Complexity

Vector clocks are powerful, but quite expensive in terms of memory overhead $O(n)$.

Question: Can we do better?

We will prove that, in order to capture causality, a vector with $n$ entries is mandatory. A smaller vector would fail in some executions.

We consider the following simple execution . . .
Consider execution $\alpha$ where every process $p_i$, $0 \leq i \leq n - 1$,

- sends a single message to all other processors except $p_{i-1}$ (taken mod $n$), all having the same delay
- messages sent one-by-one, to processors with increasing indices $p_{i+1}, p_{i+2}, \ldots, p_{i-2}$
- messages from other processors are received one-by-one,
  - from processors with decreasing indices $p_{i-1}, p_{i-2}, \ldots, p_{i+2}$
  - only after all messages have been sent by $p_i$

Let $a_i$ denote $p_i$’s first send event and $b_i$ the last receive event.
Lemma 292. For every $p_i$, $0 \leq i \leq n - 1$, in execution $\alpha$, we have

- $a_{i+1} \parallel_\alpha b_i$
- $a_{i+1} \xrightarrow{\alpha} b_j$ for every $j \neq i$
Lemma 292. For every $p_i$, $0 \leq i \leq n - 1$, in execution $\alpha$, we have

- $a_{i+1} \parallel_\alpha b_i$
- $a_{i+1} \not\rightarrow b_j$ for every $j \neq i$

Proof. From the construction of $\alpha$, it is immediately apparent that

- there is no transitive causality, since all messages are sent before any message is received
- $a_{i+1} \parallel_\alpha b_i$ follows since $p_{i+1}$ does not send a message to $p_i$
- $a_{i+1} \not\rightarrow b_j$ holds, since
  - both $a_{i+1}$ and $b_j$ occur on the same processor in case of $j = i + 1$
  - otherwise, a message is sent by $p_{i+1}$ to $p_j$ at or after event $a_{i+1}$, which is received by $p_j$ at or before $b_j$
Theorem 293. If $V C$ is a function that maps every event in $\alpha$ to a $k$-dimensional real vector in a manner that captures causality, then $k \geq n$. 
Theorem 293. If $V_C$ is a function that maps every event in $\alpha$ to a $k$-dimensional real vector in a manner that captures causality, then $k \geq n$.

Proof. Fix some $i$. Since $a_{i+1} \parallel_\alpha b_i$ by the previous lemma,

- $V_C(a_{i+1}) \not\subseteq V_C(b_i)$ and $V_C(b_i) \not\subseteq V_C(a_{i+1})$
- $\Rightarrow \exists r$ such that $V_C(b_i)[r] < V_C(a_{i+1})[r]$

Denoting $r = \ell(i)$,

- we have defined a function $\ell : \{0, \ldots, n-1\} \rightarrow \{0, \ldots, k-1\}$
- we show $k \geq n$ by showing that $\ell$ is one-to-one.

$\square$
VC Memory Complexity Lower-Bound (IV)

Proof. (cont.)

Assume, by way of contradiction that \( \ell \) is not one-to-one,

- there must be two indices \( i, j \) with \( \ell(i) = \ell(j) = r \), satisfying

\[
VC(b_i)[r] < VC(a_{i+1})[r] \quad \text{and} \quad VC(b_j)[r] < VC(a_{j+1})[r]
\]

By the previous lemma, \( a_{i+1} \xrightarrow{\alpha} b_j \) for every \( j \neq i \), so

\[
VC(b_i)[r] < VC(a_{i+1})[r] \leq VC(b_j)[r] < VC(a_{j+1})[r].
\]

Since \( a_{j+1} \xrightarrow{\alpha} b_i \) as well, we should rather have

\[
VC(a_{j+1})[r] \leq VC(b_i)[r]
\]

\( \Rightarrow \) Contradiction.
Applications like

- distributed monitoring & debugging
- global predicate evaluation

need to access the **global** system state, e.g. for

- displaying some distributed data when hitting a breakpoint
- computing some expression involving distributed data.

Problem with asynchronous systems:

- Concurrency does not allow instantaneous snapshot of global state
- What can we do?
Global State of a Distributed Computation (II)

Cut $\vec{C} = (c_0, \ldots, c_{n-1})$ of a distributed computation:

- Made up of initial prefixes $\phi_i^{c_i}$, of size $c_i$, of all $p_i$’s events
- Frontier of $\vec{C}$ is $(\phi_0^{c_0}, \phi_1^{c_1}, \ldots, \phi_{n-1}^{c_{n-1}})$
- Global state $\Sigma \vec{C} = \Sigma^{c_0,\ldots,c_{n-1}}$ defined by $\vec{C}$ is $(q_0^{c_0}, q_1^{c_1}, \ldots, q_{n-1}^{c_{n-1}})$, where $q_i^{c_i}$ is $p_i$’s state after $\phi_i^{c_i}$. 

Global State of a Distributed Computation (II)

Cut $\vec{C} = (c_0, \ldots, c_{n-1})$ of a distributed computation:

- Made up of initial prefixes $\phi_i^{c_i}$, of size $c_i$, of all $p_i$'s events
- Frontier of $\vec{C}$ is $(\phi_0^{c_0}, \phi_1^{c_1}, \ldots, \phi_{n-1}^{c_{n-1}})$
- Global state $\Sigma\vec{C} = \Sigma^{c_0,\ldots,c_{n-1}}$ defined by $\vec{C}$ is $(q_0^{c_0}, q_1^{c_1}, \ldots, q_{n-1}^{c_{n-1}})$, where $q_i^{c_i}$ is $p_i$'s state after $\phi_i^{c_i}$.

$\vec{C}$ could involve local states $q_i^{c_i}$ and $q_j^{c_j}$, where

- $q_j$ has been sampled so late after sampling $q_i$ that it causally depends on the $c_i + 1$-st event at $p_i$

$\Rightarrow q_i$ and $q_j$ contain data never seen simultaneously in the real execution

$\Rightarrow$ inconsistent snapshot of distributed data.
A cut $\widetilde{C}$ is consistent if $\phi^{c_i+1}_i \not\rightarrow \phi^{c_j}_j$ for $0 \leq i, j \leq n - 1$.

Equivalent definitions:

- all messages received inside $\widetilde{C}$ are also sent from within $\widetilde{C}$
- $\forall e \in \widetilde{C}, \forall e' \rightarrow_r e \Rightarrow e' \in \widetilde{C}$ (left-closure)
- $\forall e \in \widetilde{C} : VC(e) \leq \widetilde{C}$
Consistent Cuts

A cut $\vec{C}$ is consistent if $\phi_i^{c_{i+1}} \not\rightarrow \phi_j^{c_j}$ for $0 \leq i, j \leq n - 1$.

Equivalent definitions:

- all messages received inside $\vec{C}$ are also sent from within $\vec{C}$
- $\forall e \in \vec{C}, \forall e' \rightarrow_r e \Rightarrow e' \in \vec{C}$ (left-closure)
- $\forall e \in \vec{C}: VC(e) \leq \vec{C}$

Lemma 297. A cut $\vec{C}$ is consistent if $VC(\phi_i^{c_i})[i] \geq VC(\phi_j^{c_j})[i]$, $1 \leq i, j \leq n$. 
Consistent Cuts

A cut $\vec{C}$ is consistent if $\phi^{c_{i}+1}_{i} \not\rightarrow \phi^{c_{j}}_{j}$ for $0 \leq i, j \leq n - 1$.

Equivalent definitions:

- all messages received inside $\vec{C}$ are also sent from within $\vec{C}$
- $\forall e \in \vec{C}, \forall e' \rightarrow_r e \Rightarrow e' \in \vec{C}$ (left-closure)
- $\forall e \in \vec{C} : VC(e) \leq \vec{C}$

Lemma 297. A cut $\vec{C}$ is consistent if $VC(\phi^{c_{i}}_{i})[i] \geq VC(\phi^{c_{j}}_{j})[i]$, $1 \leq i, j \leq n$.

Proof. The simple strong clock condition yields $\phi^{c_{i}+1}_{i} \not\rightarrow \phi^{c_{j}}_{j} \iff VC(\phi^{c_{i}+1}_{i})[i] = VC(\phi^{c_{i}}_{i})[i] + 1 > VC(\phi^{c_{j}}_{j})[i]$. □
Lattice of Consistent Global States

Lattice of all consistent global states of a distributed computation:

- Global states reachable in a given asynchronous computation
- Generated by all causal shuffles, which correspond to different paths in the lattice
Finding Maximum Consistent Cut

Suppose we are given some arbitrary cut \( \vec{C}' = (c'_0, \ldots, c'_{n-1}) \).

Simple vector clock algorithm to determine (unique) maximal consistent cut \( \vec{C} \) preceding \( \vec{C}' \):

- Every \( p_i \) starts out from \( c_i = c'_i \), backwards in his event sequence, until \( VC(\phi_i^{c_i}) \leq \vec{C}' \) [or \( c_i = 0 \) if there is none]

- \( \vec{C} = (c_0, \ldots, c_{n-1}) \) made up of those \( c_i \)'s is the sought maximum consistent cut

- Proof of correctness is left as an exercise.
Chandy & Lamport Snapshot Algorithm (I)

Constructs consistent cut on-the-fly.

Prerequisites:
- FIFO links
- Only a single message received per comp-event
- Processor $p_0$ initiates the snapshot algorithm, by sending itself a special snapshot message

Achieved properties:
- Algorithm records consistent global state $(q_0, \ldots, q_{n-1})$
- Constructs also channel state $\chi_{j,i}$ of link from $p_j$ to $p_i$ (= messages sent by $p_j$ before its snapshot that arrive after $p_i$'s snapshot)
Chandy & Lamport Snapshot Algorithm (II)

Algorithm for processor $p_i$, $0 \leq i \leq n - 1$:

1. On reception of the first snapshot-message (from process $p_f$)
   - record own state $q_i$
   - relay snapshot-message to all $p_j$, $j \neq i$
   - set $p_f$’s channel state $\chi_{f,i} = \emptyset$
   - set $p_j$’s channel state $\chi_{j,i} = \emptyset$ and start recording messages from $p_j$ in $\chi_{j,i}$

2. On reception of additional snapshot-message from process $p_s$
   - stop recording messages in $\chi_{s,i}$
Theorem 302. The Chandy & Lamport algorithm constructs a consistent cut and the appropriate channel state.
Chandy & Lamport Snapshot Algorithm (III)

**Theorem 302.** The Chandy & Lamport algorithm constructs a consistent cut and the appropriate channel state.

**Proof.** Consistent cut $\vec{C}$ delivered, since otherwise $\exists i, j : \phi_{j}^{c_{j}+1} \rightarrow \phi_{i}^{c_{i}}$

- $\exists$ chain of messages starting outside $\vec{C}$ (at $p_j$) and ending inside (at $p_i$) $\Rightarrow \exists$ message $m$ sent outside $\vec{C}$ and received inside $\Rightarrow$ Contradiction, since $m$ has been sent after snapshot-message, so must arrive after the snapshot messages by the FIFO property $\Rightarrow$ should be outside the cut.

Correct channel state delivered:

- Only messages sent before collecting $p_s$’s state recorded in $\chi_{s,i}$, since otherwise $p_i$ would have got $p_s$’s snapshot-message earlier
- Only a message received after collecting $p_i$’s state (but before getting snapshot-message from $p_s$) is recorded
Clock Synchronization
Hardware Clocks

Extend processor $p_i$ by local hardware clock $HC_i$

- $HC_i : t \rightarrow T$ maps real-time $t$ to clock time $T$
- $HC_i(t)$ available to $p_i$’s transition function
- Sequence of clock readings for $p_i$’s events must be
  - monotonically increasing
  - unbounded for infinite sequences

Many conceivable clocks, with different quality:

- Ideal clocks: $HC_i(t) = t$
- Clocks with drift $\rho$:
  \[(t_2 - t_1)(1 - \rho) \leq HC_i(t_2) - HC_i(t_1) \leq (t_2 - t_1)(1 + \rho)\]
- Simple counters: $HC_i(t) = \#\text{comp}_i$-events executed by $t$
Shifting of Timed Executions

We consider timed executions of systems with drift-free clocks $HC_i(t) = t + c_i$, with unknown constant offset $c_i$:

- Hardware clock readings $HC_i(t^k_i) = t^k_i + c_i$ must be consistent with occurrence real-times $t^k_i$ of events $\phi_i^k$.
- no message deliver event occurs before its send event

Shift $\alpha' = shift(\alpha, \bar{x})$ of a timed execution $\alpha$ by some $\bar{x} = (x_0, \ldots, x_{n-1})$:

- Shift all events $\phi_i^k$, $k \geq 1$, of $p_i$ in real-time by $x_i$.
- Event $\phi_i^k$ occurring at $t^k_i$ in $\alpha$ occurs at $t^k_i' = t^k_i + x_i$ in $\alpha'$.
  - requires $HC'_i(t^k_i') = HC'_i(t^k_i + x_i) = HC'_i(t^k_i)$
  - only allowed if no message delivered before sent
Shifting Lemma

Lemma 306. Let $\alpha$ be a timed execution and $\alpha' = \text{shift}(\alpha, \vec{x})$ for shifting vector $\vec{x}$, in case of $HC_i(t) = t + c_i$. Then, for any $0 \leq i, j \leq n - 1$,

- $HC'_i(t) = HC_i(t) - x_i$ (shift right $\Rightarrow$ $HC'$ behind at same time)
- every message from $p_i$ to $p_j$ with delay $\delta$ in $\alpha$ has delay $\delta' = \delta - x_i + x_j$ in $\alpha'$
Shifting Lemma

**Lemma 306.** Let $\alpha$ be a timed execution and $\alpha' = \text{shift}(\alpha, \vec{x})$ for shifting vector $\vec{x}$, in case of $HC_i(t) = t + c_i$. Then, for any $0 \leq i, j \leq n - 1$,

- $HC'_i(t) = HC_i(t) - x_i$ (shift right $\Rightarrow HC'$ behind at same time)
- every message from $p_i$ to $p_j$ with delay $\delta$ in $\alpha$ has delay $\delta' = \delta - x_i + x_j$ in $\alpha'$

**Proof.** The first statement follows from

- $HC_i(t) = T = HC'_i(t + x_i)$ by definition
- $HC'_i(t + x_i) = HC'_i(t) + x_i$ by the no drift assumption.

For the second statement, consider message $m$ sent by $p_i$ at real-time $t^s$ and received by $p_j$ at time $t^r$ in $\alpha$; it has delay $\delta = t^r - t^s$. In $\alpha'$,

- sending occurs at real-time $t^s + x_i$ and reception occurs at $t^r + x_j$
- the delay is $\delta' = t^r + x_j - t^s - x_i = \delta - x_i + x_j$ as asserted.
The Clock Synchronization Problem

Adjusted clock $AC_i(t) = HC_i(t) + adj_i(t)$ of $p_i$:

- Hardware clock $HC_i$ cannot be manipulated by $p_i$
- State variable $adj_i$ that can be used to adjust the clock

Properties clock synchronization algorithm with skew $\epsilon$ (and no failures): In every admissible execution.

- every processor terminates by some time $t_f$
- $|AC_i(t) - AC_j(t)| \leq \epsilon$ for $t \geq t_f$ and every pair of processors $p_i, p_j$

Some additional definitions:

- Precision $\pi$ such that $|AC_i^{[-1]}(T) - AC_j^{[-1]}(T)| \leq \pi$
- Message delays $\delta \in [d - u, d]$, with uncertainty $u$
The 2-Processor Case

Very simple approach for synchronizing $p_1$’s clock to $p_0$’s:

- $p_0$ sets $adj_0 = 0$ and sends $T_0 = AC_0(t_0) = HC_0(t_0)$ to $p_1$
  at real-time $t_0$

- $p_1$ sets $AC_1(t_1) := T_0 + d - u + X$ at the real-time $t_1 \in [t_0 + d - u, t_0 + d]$ when it gets $p_0$’s message

- clearly, $AC_0(t_1) = T_0 + (t_1 - t_0) \in T_0 + d - u + [0, u]$

Resulting skew $\epsilon = AC_0(t_1) - AC_1(t_1) \in [-X, u - X]$, which is $\epsilon = u/2$ when choosing $X = u/2$

We will show that one cannot do better. In what follows,

- let $t$ be any time after termination

- abbreviate $AC_i(t)$ by $AC_i$, $AC'_i(t)$ by $AC'_i$, etc.
2-Processor Lower Bound $\epsilon \geq \frac{u}{2}$

**Theorem 309.** Any 2-processor clock synchronization algorithm $A$ has a skew $\epsilon$ of at least $\frac{u}{2}$.
**Theorem 309.** Any 2-processor clock synchronization algorithm $\mathcal{A}$ has a skew $\epsilon$ of at least $u/2$.

*Proof.* Consider admissible timed execution $\alpha$, where

- all messages $p_0 \rightarrow p_1$ have delay $d - u$, all messages $p_1 \rightarrow p_0$ have delay $d$
- Since $\mathcal{A}$ has skew $\epsilon$, $AC_0 \geq AC_1 - \epsilon$

Now consider $\alpha' = \text{shift}(\alpha, \vec{x})$ for $\vec{x} = (-u, 0)$:

- $\alpha'$ is admissible, and $AC_1' \geq AC_0' - \epsilon$ since $\mathcal{A}$ has skew $\epsilon$
- By the shifting lemma, $AC_0' = AC_0 + u$ and $AC_1' = AC_1$, which implies $AC_1 \geq AC_0 + u - \epsilon$

Putting the blue inequalities together, we obtain $AC_0 \geq AC_0 + u - 2\epsilon$ and hence $2\epsilon \geq u$. $\square$
Lemma 310. Consider any admissible timed execution $\alpha$ of a clock synchronization algorithm with skew $\epsilon$, where all messages $p_i \rightarrow p_j$ have delay $d - u$ and all messages $p_j \rightarrow p_i$ have delay $d$ (for $i < j$). For every $1 \leq k \leq n - 1$, $AC_{k-1} \leq AC_k - u + \epsilon$. 
**Preparation Lemma**

**Lemma 310.** Consider any admissible timed execution $\alpha$ of a clock synchronization algorithm with skew $\epsilon$, where all messages $p_i \rightarrow p_j$ have delay $d - u$ and all messages $p_j \rightarrow p_i$ have delay $d$ (for $i < j$). For every $1 \leq k \leq n - 1$, $AC_{k-1} \leq AC_k - u + \epsilon$.

**Proof.** Fix any $k$ and consider $\alpha' = \text{shift}(\alpha, \vec{x})$ where $x_i = -u$ for $0 \leq i \leq k - 1$ and $x_i = 0$ otherwise.

- $\alpha'$ is admissible since any message from $p_i \rightarrow p_j$ (resp. $p_j \rightarrow p_i$) for $i < j$ has
  - delay $d - u$ (resp. $d$), as in $\alpha$, if $j \leq k - 1$ or $i \geq k$  
  - delay $d$ (resp. $d - u$) if $i \leq k - 1 < j$

- $AC'_{k} \geq AC'_{k-1} - \epsilon$ since $A$ has skew $\epsilon$

By the shifting lemma, $AC'_{k-1} = AC_{k-1} + u$ and $AC'_k = AC_k$, which implies $AC_k \geq AC_{k-1} + u - \epsilon$ as asserted.
Theorem 311. Any $n$-processor clock synchronization algorithm $A$ has a skew $\epsilon$ of at least $u(1 - 1/n)$. 
\textbf{Theorem 311.} Any \(n\)-processor clock synchronization algorithm \(A\) has a skew \(\epsilon\) of at least \(u\left(1 - \frac{1}{n}\right)\).

\textit{Proof.} Consider an admissible timed execution \(\alpha\), where for \(i < j\):
\begin{itemize}
  \item all messages \(p_i \rightarrow p_j\) have delay \(d - u\)
  \item all messages \(p_j \rightarrow p_i\) have delay \(d\)
\end{itemize}

From preparation lemma, we know that \(AC_{k-1} \leq AC_k - u + \epsilon\) for any \(1 \leq k \leq n - 1\). Hence,
\[
AC_0 \leq AC_1 - u + \epsilon \leq AC_2 - 2u + 2\epsilon \leq \cdots \leq AC_{n-1} - (n - 1)(u - \epsilon)
\]

In addition, by skew \(\epsilon\) of \(A\), \(AC_{n-1} \leq AC_0 + \epsilon\)
\[
\Rightarrow AC_{n-1} \leq AC_{n-1} - (n - 1)u + n\epsilon, \text{ from where the theorem follows.}\]
We will now show that the lower bound $\epsilon \geq u(1 - 1/n)$ is tight.

**Pseudo-code Algorithm 20 for $p_i$, $0 \leq i \leq n - 1$:**

1. Initially $d[i] = 0$

2. At first computation step:
   - send $HC(t)$ to all processors

3. On receiving message containing $T$ from $p_j$:
   - $d[j] := T + d - u/2 - HC(t)$

4. if message has been received from all processors then

5. $adj := \frac{1}{n} \sum_{k=0}^{n-1} d[k]$
Theorem 313. *The simple* \( n \)-*processor clock synchronization algorithm has a skew* \( \epsilon \leq u \left( 1 - \frac{1}{n} \right) \).
**Theorem 313.** The simple \( n \)-processor clock synchronization algorithm has a skew \( \epsilon \leq u \left( 1 - \frac{1}{n} \right) \).

**Proof.** Consider any admissible timed execution \( \alpha \), and abbreviate \( HC_i = HC_i(t) \) for some time \( t \) after termination. From the simple 2-processor case, we know:

- \( HC_i + d_i[j] = HC_j + err^j_i \) with \(-u/2 \leq err^j_i \leq u/2\)
- \( HC_i + d_i[k] - HC_j - d_j[k] = err^k_i - err^k_j \) with \(-u \leq err^k_i - err^k_j \leq u\)

We proceed by evaluating

\[
D = |AC_i - AC_j| = \left| HC_i + \frac{1}{n} \sum_{k=0}^{n-1} d_i[k] - HC_j - \frac{1}{n} \sum_{k=0}^{n-1} d_j[k] \right|
\]
Simple \( n \)-Proc. Clock Synchronization (III)

\textbf{Proof.} (cont.)

Some algebra yields

\[
D = \frac{1}{n} \left| HC_i - HC_j - d_j[i] + HC_i + d_i[j] - HC_j \right|
+ \sum_{k=0, k \neq i, j}^{n-1} HC_i + d_i[k] - HC_j - d_j[k]
\]

\[
= \frac{1}{n} \left| -err_j^j + err_i^j + \sum_{k=0, k \neq i, j}^{n-1} (err_i^k - err_j^k) \right|
\]

\[
\leq \frac{1}{n} \left[ u/2 + u/2 + (n - 2)u \right] = u \left( 1 - \frac{1}{n} \right).
\]

\[\square\]
Simple $n$-Proc. Clock Synchronization (III)

Proof. (cont.)

Some algebra yields

$$D = \frac{1}{n} \left| HC_i - HC_j - d_j[i] + HC_i + d_i[j] - HC_j \right.$$  

$$+ \sum_{k=0, k \neq i,j}^{n-1} HC_i + d_i[k] - HC_j - d_j[k] \right|$$

$$= \frac{1}{n} \left| -err_j^j + err_i^j + \sum_{k=0, k \neq i,j}^{n-1} (err_i^k - err_j^k) \right|$$

$$\leq \frac{1}{n} \left[ u/2 + u/2 + (n - 2)u \right] = u \left( 1 - \frac{1}{n} \right).$$

$\square$
The End (Basics)