Goals:
• Review the main topics we have covered in the course
• Describe the material you should be prepared to see on the final exam

Material that will be covered on the final exam:
• P. Sivilotti, *Introduction to Distributed Algorithms*, Chapters 1-12
• K. M. Chandy and J. Misra, *Distributed Algorithms*, Ch 7
  - [available on Moodle; covers program composition]
Course Summary: Weeks 1-5

Weeks 1-3: UNITY programs
- Predicate calculus, equivalence, quantification [HW #1]
- Program execution (UNITY semantics) [HW #2]
- Stability properties (next, stable, invariant, unless) [HW #2]
- Progress properties (transient, ensures, leadsto) [HW #3]
- Induction (metrics) and proofs of correctness [HW #3, 4]

Week 4: Intro multi-agent systems
- Logical clocks and vector clocks [HW #4]
- Diffusing computations [HW #4]

Week 5: Mutual exclusion
- Restrict access to a resource to a single process
- User processes + control protocols (composition)
Course Summary: Weeks 6-10

Week 6: Synchronization (for distributed systems)
- How do we synchronize a set of agents to perform a coordinated function [HW #5]
- Example: “dining philosophers” [HW #5]

Week 7: Specifications, composition / snapshots
- Program composition - $P = F \parallel G$ [HW #6]
- Snapshots - consistent cuts [HW #6]

Week 8: consensus with faults
- How do we expand concepts so far when there might be malicious (or failing) agents present [HW #7]
- Example: “Byzantine generals problem” [HW #7]

Week 9/10 (Thanksgiving): Paxos and distributed databases
- Maintaining consistent distributed databases (including possibility of faulty or malicious agents) [HW #7]
- Paxos algorithm [HW #8]
- Example: blockchain/bitcoin [HW #8]
Chapter 1: Booleans, Predicates, Quantification

Predicate calculus:
- Standard logical operators (right)
- Everywhere brackets: \([P(x)]\) means \(P(x)\) is true for all states \(x\)

Equivalence (and discrepancy)
- \([P \equiv Q]\) means logical values match
- \([P \not\equiv Q]\) means logical values differ

Quantification
- \((Q)\) means \(u \equiv Q t(i_0) \equiv Q t(i_1) \ldots \equiv Q t(i_N)\)
  where \(i_0, i_1, \ldots, i_N\) satisfy \(r(i)\)

Proof format: to show that \([A \equiv C]\)

\[
\begin{align*}
A & \\
\equiv & \{ \text{reason why } [A \equiv B] \} \\
B & \\
\equiv & \{ \text{reason why } [B \equiv C] \} \\
C & \\
\end{align*}
\]

Operator ordering
- \(\neg\) (logical negation)
- \(* /\) (arithmetic multiplication and division)
- \(+ -\) (arithmetic addition and subtraction)
- \(< > \leq \geq \neq\) (arithmetic comparison)
- \(\land \lor\) (logical and and or)
- next unless ensures \(\sim\)
- \(\Rightarrow \Leftarrow\) (logical implication and explication)
- \(\equiv \neq\) (logical equivalents and discrepancy)

Be careful about implications (direction):

\[
\text{transient.}(r = k \land r < M) \leftarrow \{ \text{weakening antecedent} \} \\
M > k \Rightarrow \max (r, M) > k \equiv \{ \text{property of } \max \} \text{ true}
\]
Chapter 2: The Computational Model

UNITY model provides (seemingly) simple description of programs
- Program = variables + actions [assignments] (that’s it!)
- Guarded assignment \((g \rightarrow a)\) allows modeling of finite state automata
- Distributed programs captured by nondeterministic execution model
- Termination = reaching a fixed point (variables remain constant)

Graph representations of programs
- Represent each state as a node, each action as an edge
- Remember: any action can be applied at any state (often omit edges)

Fairness
- Weak fairness: every action selected infinitely often
- Strong fairness: can’t ignore action forever

Things to remember
- The skip action can be applied at any point
Chapter 3: Reasoning About Programs (1 of 2)

Key elements of a specification

- **Safety**: properties that should always be true
  - \[ P \text{ next } Q \equiv (\forall a : a \in G : \{P\} \ a \ {Q}) \]
  - \[ \text{stable}(P) \equiv P \text{ next } P \]
  - \[ \text{invariant}(P) \equiv \text{initially}(P) \wedge \text{stable}(P) \]
  - \[ P \text{ unless } Q \equiv (\forall a : a \in F : \{P \wedge \neg Q\} a \ {P \lor Q}) \]

- **Progress**: properties that should eventually be true
  - \[ \text{transient}(P) \equiv (\exists a : a \in G : \{P\} a \ \{\neg P\}) \]
  - \[ P \text{ ensures } Q \equiv P \text{ unless } Q \wedge \text{transient}(P \wedge \neg Q) \]
  - Leads to:
    \[ P \text{ ensures } Q \implies P \leadsto Q \]
    \[ (P \leadsto Q) \wedge (Q \leadsto R) \implies P \leadsto R \]
    \[ (\forall i :: P_i \leadsto Q) \implies (\exists i :: P_i) \leadsto Q \]

Key elements of a proof

- **Fixed points**: points at which the computation terminates
- **Invariants**: properties preserved during execution
- **Metric**: bounded function used to measure progress
Chapter 3: Reasoning About Programs (2 of 2)

How to prove a program is correct

1. Write down the program as a UNITY program (collection of guarded commands)
2. Write down the fixed points (where you want the system to end up)
3. Write down the invariants to demonstrate safety
4. Find a metric (variant function) that shows progress

\[
(\forall m :: P \land M = m \ \text{next} \ (P \land M \leq m) \lor Q) \land (\forall m :: \text{transient}.(P \land M = m)) \\
\Rightarrow P \leadsto Q
\]

Frequently asked questions

- Q: what can I assume w/out proving? A: anything in Sivilotti or proved in class or HW
- Q: how much detail do we have to provide in a proof
  - A1: if the question asked for a “detailed” proof, include a step-by-step proof
  - A2: OK to summarize the key ideas, as long as you justify/don’t miss any cases
- Q: How do we figure out the invariants and metrics
  - A1: if you are given the algorithm, only method is trial and error
  - A2: if you are designing the algorithm, you can couple design and proof

Chapter 4 of Sivilotti provides examples of proofs for some simple programs
Chapter 5: Time, Clocks, and Synchronization

Channel model: FIFO, lossless, directed

Events, system timelines and logical time

- Can’t assume process clocks agree
- Make use of “logical time”

\[ A \rightarrow B \Rightarrow \text{time}.A < \text{time}.B \]

Algorithm for setting logical time

\[
\begin{array}{l}
\text{local event } A \rightarrow \quad \text{clock}.j := \text{clock}.j + 1 \\
\quad \quad ; \quad \text{time}.A := \text{clock}.j
\\
\text{send event } A \rightarrow \quad \text{clock}.j := \text{clock}.j + 1 \\
\quad \quad (\text{to } k) \quad ; \quad \text{time}.A, \text{time}.m := \text{clock}.j, \text{clock}.j \\
\quad \quad ; \quad \text{ch}.j.k := \text{ch}.j.k | m
\\
\text{rcv event } A \rightarrow \quad \text{clock}.j := \max \text{(time}.m, \text{clock}.j) + 1 \\
\quad \quad (m \text{ from } k) \quad ; \quad \text{time}.A, \text{ch}.j.k := \text{clock}.j, \text{tail(ch}.j.k)
\end{array}
\]

Properties

\[
(\forall A, j : A \text{ occurs at } j : \text{time}.A \leq \text{clock}.j)
\land (\forall m, j, k : m \in \text{ch}.j.k : (\exists A : A \text{ occurs at } j : \text{time}.A = \text{time}.m))
\land (\forall A, B :: A \rightarrow B \Rightarrow \text{time}.A < \text{time}.B)
\]

Vector clocks: \[ A \rightarrow B \equiv v\text{time}.A < v\text{time}.B \]
Chapter 6: Diffusing Computations (Gossip)

Basic idea: distribute information to all nodes

- Key problem is understanding when the algorithm has terminated (all nodes idle, no information in channels)
- Make use of a tree structure to propagate information

Properties

**safety:** invariant. \((\text{done } \Rightarrow (\forall u :: u \text{ has completed gossip }))\)

**progress:** \((\forall v : v \text{ nbr } I : \text{msg}(I,v)) \sim \text{done}\)

Algorithm

**initially** \(\text{idle}\)

\(\wedge (\forall v : u \text{ nbr } v : \neg \text{msg}(u,v))\)

**assign**

\((\parallel v : v \text{ nbr } u : \text{idle} \land \text{msg}(v,u) \rightarrow \parallel \text{parent}_u := v \parallel (\parallel w : w \text{ nbr } u \land w \neq v : \text{msg}(u,w) := \text{true} ) \parallel \text{state}_u := \text{active} ) \parallel \text{active} \land (\forall v : v \text{ nbr } u \land v \neq \text{parent}_u : \text{msg}(v,u) ) \rightarrow \parallel \text{msg}(u,\text{parent}_u) := \text{true} \parallel \text{state}_u := \text{complete} )\)

**Simplified channel model**

- Keep track of whether message is in channel
- Works because we only use channel once
Chapter 7: Mutual Exclusion

Key ideas:
- Distributed protocol for allow access to a shared resource (“critical section”)
- Two approaches: distributed atomic variables (Lamport + variants) or token-based
- User process specifications:
  - \( NC \) next \( NC \lor TRY \)
  - stable.\( TRY \)
  - \( CS \) next \( CS \lor NC \)
  - transient.\( CS \)
- Composite (system) specifications:
  - Safety: no two users (\( U_i \)) are in critical section (CS) at the same time
  - Progress: all agents will get a chance (as long as they keep requesting)
- Constraints:
  - \( (\forall u : \text{stable}(u.m=CS)) \) in \( os \)
  - \( (\forall u : \text{stable}(u.m=NC)) \) in \( os \)
Chapter 8: Dining Philosophers (Refinement)

Key ideas:

- Specifications for composed systems
  - Properties of the underlying process (user)
  - Properties of the composed system (user | os)
  - Constraints on access to user processes

- Design via successive refinement (R => P)
  - Refine properties to establish program structure
  - Each refinement solves problem from previous level (and satisfies the prior specs)
  - Final specification can be converted to code

Program description

\[ [H_p] \quad p.h \land \text{fork}(p,q) = q \rightarrow \text{req}(p,q) := q; \]

\[ [E_p] \quad p.h \land (\forall q : E(p,q) : \text{fork}(p,q) = p \land (\text{clean}(p,q) \lor \text{req}(p,q) = q)) \rightarrow p.state := \text{eating}; \]
\[ \quad \text{clean}(p,q) := \text{false}; \]

\[ [R_p] \quad \text{req}(p,q) = p \land \text{fork}(p,q) = p \land \neg\text{clean}(p,q) \land \neg p.e \rightarrow \text{fork}(p,q) := q; \]
\[ \quad \text{clean}(p,q) := \neg\text{clean}(p,q); \]
Chandy and Misra, Ch 7: Program Composition

Key ideas:

• Specifications for composed systems
  - Properties of the underlying process (user)
  - Properties of the composed system (user | os)
  - Constraints on access to user processes

• Design via successive refinement
  - Refine properties to establish program structure
  - Each refinement solves problem from previous level (and satisfies the prior specs)
  - Final specification can be converted to code

• Advantages of this approach
  - Maintain a formal proof structure throughout
  - Painful, but necessary for safety critical systems

Key ideas

• Conditional properties: properties that are part of a “program” (P in F)

• Allow composition of programs P = F \parallel G
  - Superposition, augmentation, variable sharing
Chapter 9: Snapshots

Problem statement

• Capture a consistent state of the system: a state that the system could have achieved during execution
• Key challenge is lack of global time => can get inconsistent information (can lead to double counting, lost data, etc)
• Basic property of consistent cut: all messages go from “inside” (prior to cut) to “outside” (after cut)

Solution #1: logical clocks (from Sivilotti)

• Record the state of each process at the same logical time
• Keep track of messages that are still in flight (compare sent/recv counts)

Solution #2: markers (focus of lecture)

• Send markers along the channels to “flush” out any messages that are in transit
• Initiator: record local state and send marker along each outgoing channel
• Process receiving marker records local state, mark state of incoming channel as empty, send markers along outgoing channel
• Process receiving subsequent marker: record messages received in channel since snapshot was taken; mark state of incoming channel as empty

Ch 10 and 11 in Sivilotti are applications of snapshots (good for review!)
Chapter 12: Byzantine Agreement

Failure models
- Fail-stop: processor fails and others know
- Crash (fail-silent): failure w/out notification
- Byzantine: failed process can be malicious

Specifications
- Safety: All correct (non-faulty) processes decide on a common (valid) value
- Progress: All non-faulty processes decide

Limits on agreement
- Asynchronous failures: if there are no time bounds available, fault tolerance impossible
- For synchronous agreement (rounds), can tolerate up to n/3-1 failed processes (byzantine)
- With signatures, can solve with enough rounds

Paxos algorithm for consensus with failure
- Can only prove safety, but progress OK in practice

Bitcoin is a variant using proof-of-work + randomization + incentives
From Day 1: Introduction to Distributed Computing

Main takeaway points
- Distributed systems (and hence distributed algorithms) are everywhere
- Debugging concurrent systems is much harder than debugging sequential programs
- For safety- (or business-) critical systems, formal proofs of correctness are key

In this class, we will learn to
- Model a distributed algorithm and how it executes
- Write specifications for correctness (safety, liveness)
- Prove that distributed algorithms are correct
CS/IDS 142 - Distributed Computing

Instructors: Richard Murray and Mani Chandy

PICK UP HANDOUTS AT
LECTURE HALL ENTRANCES

Announcements

- Final exam: due on 13 Dec (Fri) at 5 pm
  - Open book/notes, 3 hrs, take home
  - Piazza will be frozen on 10 Dec (Tue) at 65 pm
  - Solutions to HW #8 will be posted by 10 Dec (Tue) at ~6 pm (NLT 8 pm)

- Recitation sections in preparation for finals
  - 9 Dec (Sun), 5-6 pm in 106 ANB
  - 10 Dec (Mon), 5-6 pm in 243 ANB
  - 11 Dec (Tue), 5-6 pm in 243 ANB