Caltech

CS/IDS 142: Lecture 10.3
Course Review

Richard M. Murray

6 December 2019

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]
- L. Lamport, “Paxos Made Simple”, 2001.
- S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System”, 2008.

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]

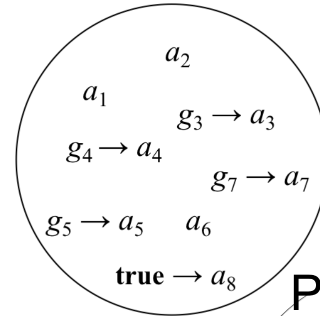
Week 4: Intro multi-agent systems

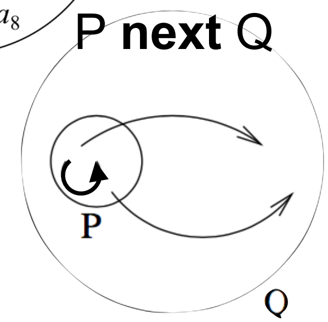
- Logical clocks and vector clocks [HW #4]
- Diffusing computations [HW #4]

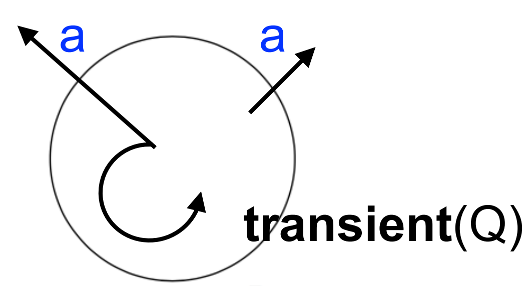
Week 5: Mutual exclusion

- Restrict access to a resource to a single process
- User processes + control protocols (composition)

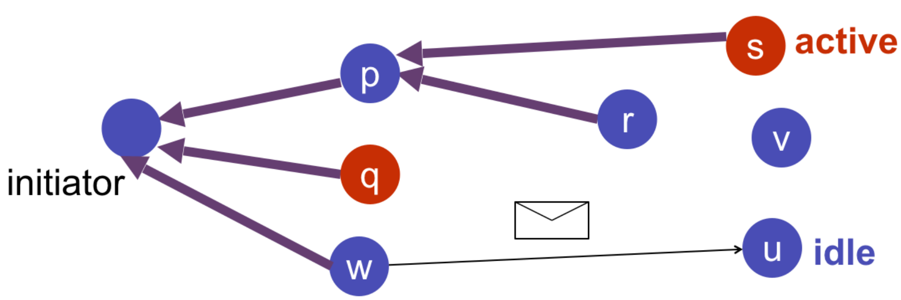
Midterm

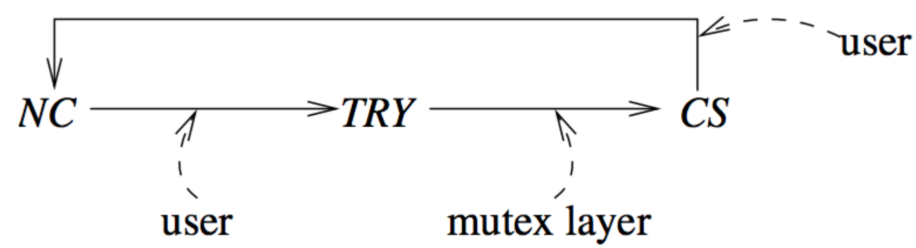






$$(\forall a : a \in G : \{P\} \quad a \quad \{Q\})$$





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

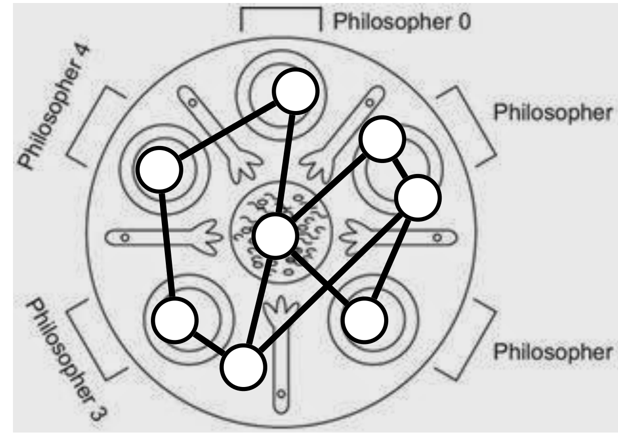
- Program composition - P = F ▯ G [HW #6]
- Snapshots - consistent cuts [HW #6]

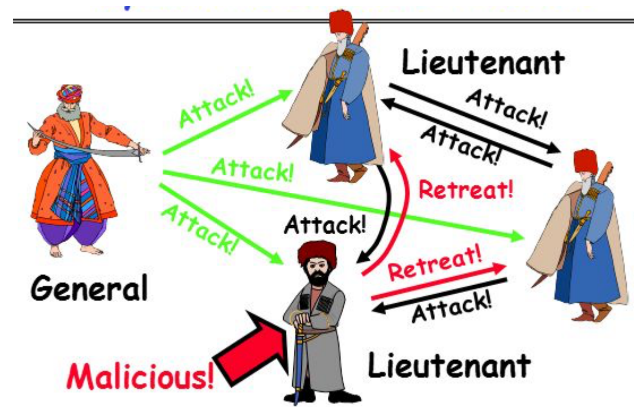
Week 8: consensus with faults

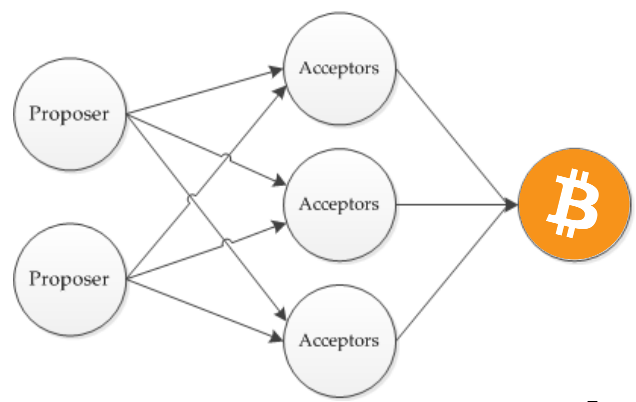
- How do we expand concepts so far when there might be malicious (or failing) agents present [HW #7]
- Example: “Byzantine generals problem” [HW #7]

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 discrepance)

- [P ≡ Q] means logical values match
- [P ≢ Q] means logical values differ

Quantification

- (**Q** i : r(i) : t(i)) means
  u **Q** $t(i_0)$ **Q** $t(i_1)$ … **Q** $t(i_N)$
  where i_0, i_1, … i_N satisfy r(i)

Proof format: to show that [A ≡ C]

$$\begin{array}{ll} & A \\ \equiv & \{\text{reason why } [A \equiv B]\} \\ & B \\ \equiv & \{\text{reason why } [B \equiv C]\} \\ & C \end{array}$$

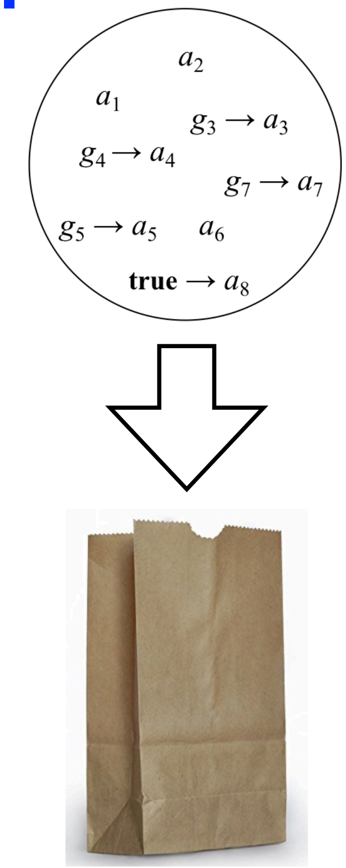
Operator ordering

- $\neg$ (logical negation)
- $*$ $/$ (arithmetic multiplication and division)
- $+$ $-$ (arithmetic addition and subtraction)
- $<$ $>$ $\leq$ $\geq$ $=$ $\neq$ (arithmetic comparison)
- $\wedge$ $\vee$ (logical and and or)
- **next** **unless** **ensures** $\leadsto$
- $\Rightarrow$ $\Leftarrow$ (logical implication and explication)
- $\equiv$ $\not\equiv$ (logical equivals and discrepance)

Be careful about implications (direction):

$$\begin{array}{ll} & \textbf{transient}.(r = k \wedge r < M) \\ \Leftarrow & \{\ \text{weakening antecedent}\ \} \\ & M > k \Rightarrow \textbf{max}\,(r, M) > k \\ \equiv & \{\ \text{property of}\ \ \textbf{max}\ \} \\ & \textbf{true} \end{array}$$

Chapter 2: The Computational Model



UNITY model provides (seemingly) simple description of programs

- Program = variables + actions [assignments] (that's it!)
- Guarded assignment (g → 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

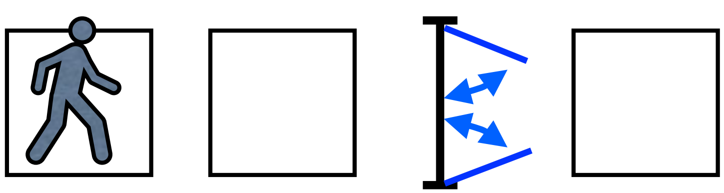
- Represent each state as a node, each action as an edge
- Remember: *any* action can be applied at *any* state (often omit edges)

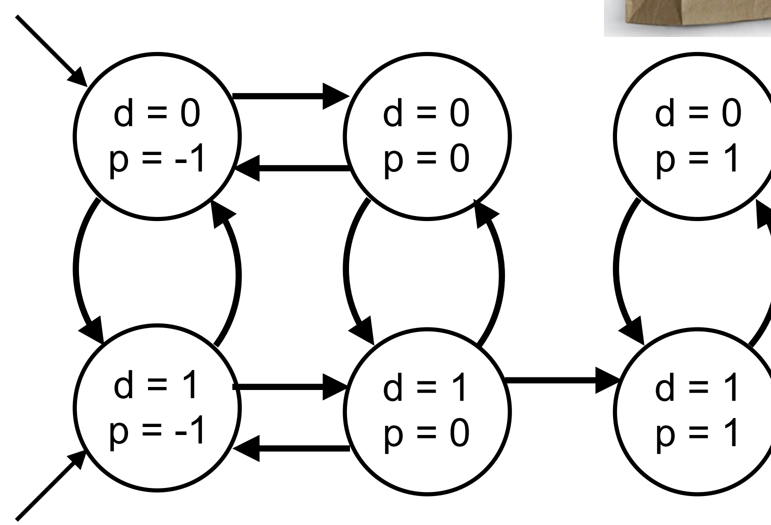
Fairness

- 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)

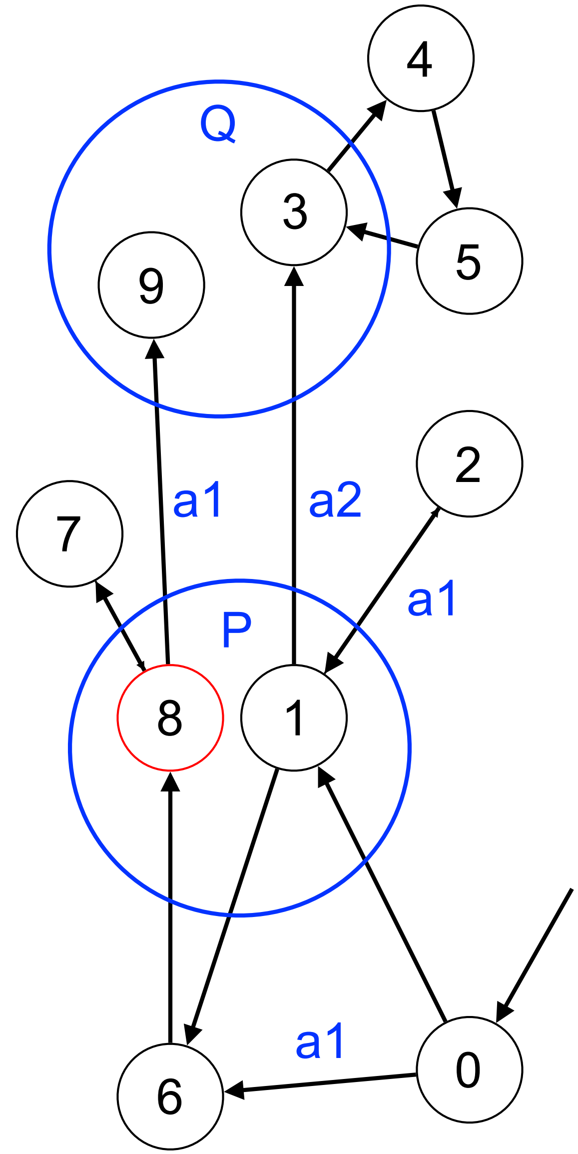
Key elements of a specification

- Safety: properties that should always be true
  - P **next** Q ≡ $(\forall a : a \in G : \{P\} \quad a \quad \{Q\})$
  - **stable**(P) ≡ P **next** P
  - $\textbf{invariant}(P) \equiv \textbf{initially}(P) \wedge \textbf{stable}(P)$
  - P **unless** Q ≡ (∀a: a ∈ F : {P ∧ ¬Q} a {P ∨ Q})
- Progress: properties that should eventually be true
  - **transient**(P) ≡ $(\exists a : a \in G : \{P\}\, a\, \{\neg P\})$
  - P **ensures** Q ≡ P **unless** Q ∧ **transient**(P ∧ ¬Q)
  - Leads to:

$$\begin{aligned} P \textbf{ ensures } Q \quad &\Rightarrow \quad P \leadsto Q \\ (P \leadsto Q) \wedge (Q \leadsto R) \quad &\Rightarrow \quad P \leadsto R \\ (\forall i :: P_i \leadsto Q) \quad &\Rightarrow \quad (\exists i :: P_i) \leadsto Q \end{aligned}$$

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



Hoare triple: {P} a {Q}

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

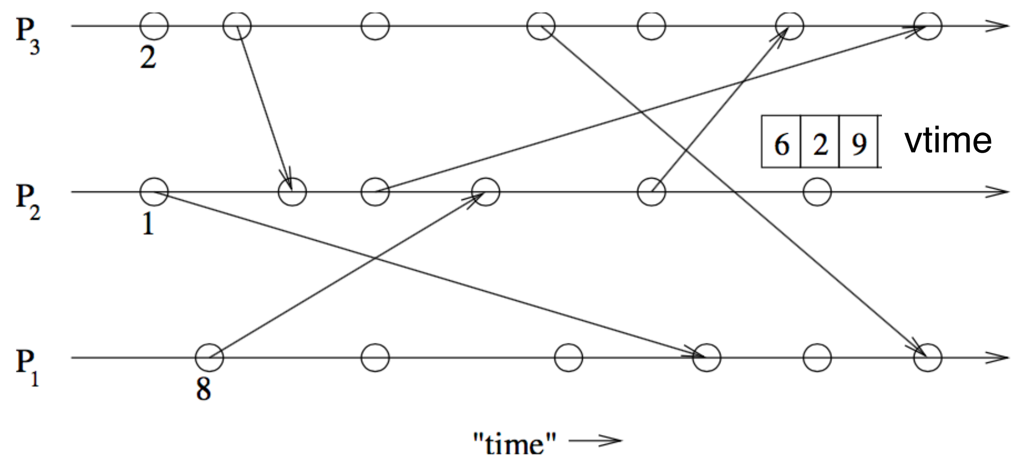
$$
\begin{aligned}
& (\forall m :: \; P \wedge M = m \;\textbf{next}\; (P \wedge M \leq m) \vee Q \;) \\
& \wedge \; (\forall m :: \; \textbf{transient}.(P \wedge M = m) \;) \\
\Rightarrow \quad & P \leadsto Q
\end{aligned}
$$

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"

$$A \longrightarrow B \;\Rightarrow\; time.A < time.B$$

Algorithm for setting logical time

$$\begin{array}{lll}
& \text{local event } A \longrightarrow & clock.j := clock.j + 1 \\
& ; & time.A := clock.j \\
[\!] & \text{send event } A \longrightarrow & clock.j := clock.j + 1 \\
& (\text{to } k) \quad ; & time.A, time.m := clock.j, clock.j \\
& ; & ch.j.k := ch.j.k \mid m \\
[\!] & \text{rcv event } A \longrightarrow & clock.j := \mathbf{max}\,(time.m, clock.j) + 1 \\
& (m \text{ from } k) \quad ; & time.A, ch.j.k := clock.j, tail(ch.j.k)
\end{array}$$

Properties

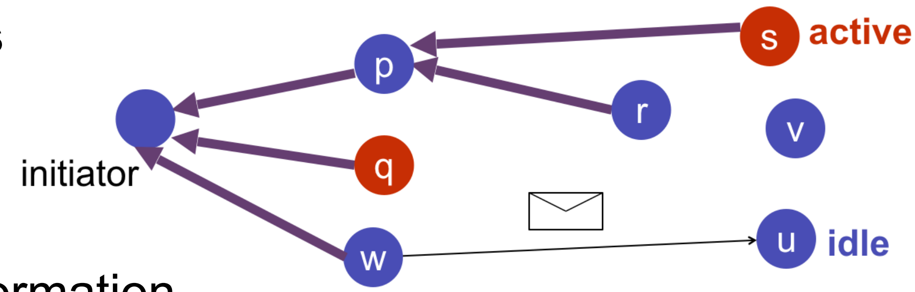
$$\begin{array}{ll}
& (\forall A, j \;:\; A \text{ occurs at } j \;:\; time.A \leq clock.j\;) \\
\wedge & (\forall m, j, k \;:\; m \in ch.j.k \;:\; (\exists A \;:\; A \text{ occurs at } j \;:\; time.A = time.m\;)\;) \\
\wedge & (\forall A, B :: A \longrightarrow B \Rightarrow time.A < time.B\;)
\end{array}$$

Vector clocks: $A \longrightarrow B \;\equiv\; vtime.A < vtime.B$

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: $\textbf{invariant}.(done \;\Rightarrow\; (\forall u :: \; u \text{ has completed gossip }))$

progress: $(\forall v \;:\; v \; nbr \; I \;:\; msg(I,v)\;) \;\rightsquigarrow\; done$

Algorithm

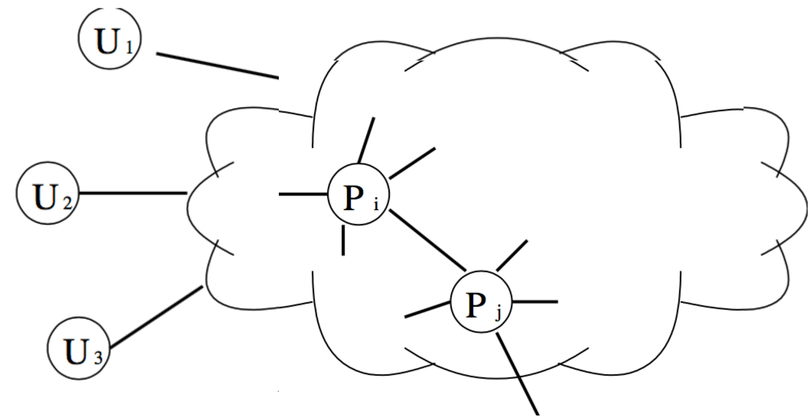
$parent_u$: process,
$state_u$: $\{idle, active, complete\}$,
$msg(a,b)$: channel from a to b,

$$
\begin{array}{ll}
\textbf{initially} & idle \\
 & \wedge \;\; (\forall v \;:\; u\; nbr\; v \;:\; \neg msg(u,v)\;) \\
\textbf{assign} & \\
\quad (\; [\!]\; v \;:\; v\; nbr\; u \;:\; idle \wedge msg(v,u) \longrightarrow & \\
\qquad parent_u := v & \\
\qquad \| \;\; (\,\|\; w \;:\; w\; nbr\; u \wedge w \neq v \;:\; msg(u,w) := \textbf{true}\;) & \\
\qquad \| \;\; state_u := active\;) & \\
\quad [\!]\; active \wedge (\forall v \;:\; v\; nbr\; u \wedge v \neq parent_u \;:\; msg(v,u)\;) \longrightarrow & \\
\qquad msg(u, parent_u) := \textbf{true} & \\
\qquad \| \;\; state_u := complete &
\end{array}
$$

Simplified channel model

- Keep track of whether message is in channel
- Works because we only use channel once

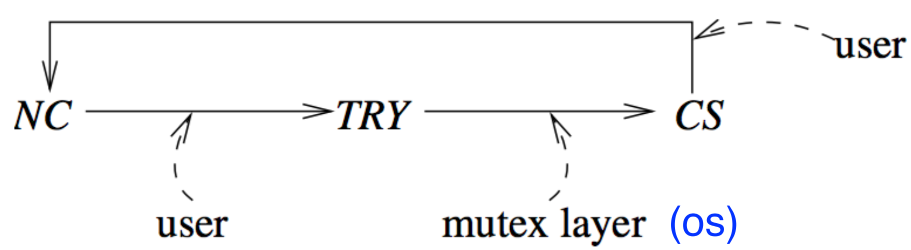
Chapter 7: Mutual Exclusion



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:

$$NC \textbf{ next } NC \vee TRY$$
$$\textbf{stable}.TRY$$
$$CS \textbf{ next } CS \vee NC$$
$$\textbf{transient}.CS$$

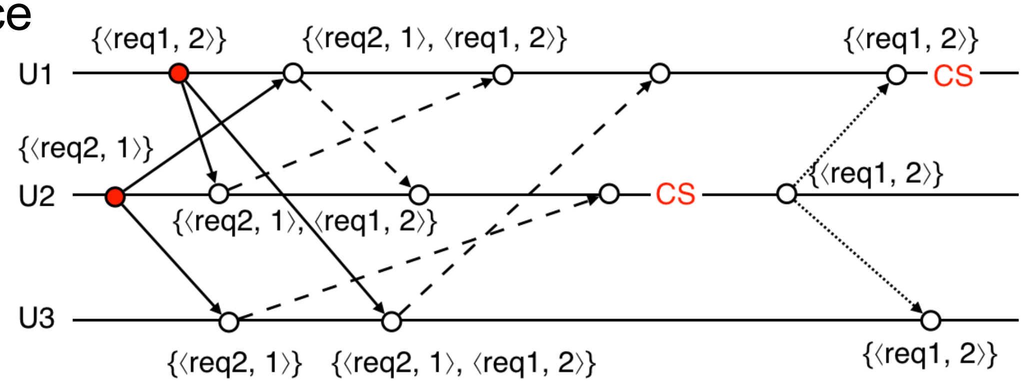


- *Composite* (system) specifications:
  - Safety: no two users (Ui) are in critical section (CS) at the same time
  - Progress: all agents will get a chance (as long as they keep requesting)
- Constraints:
  - (∀u : stable(u.m=CS)) in os
  - (∀u : stable(u.m=NC) in os

Composition properties:

TRY **next** TRY ∨ CS

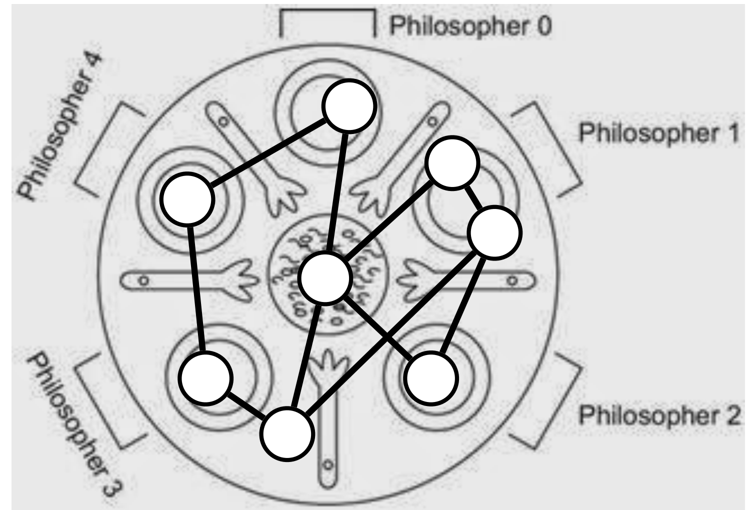
TRY ⇝ CS

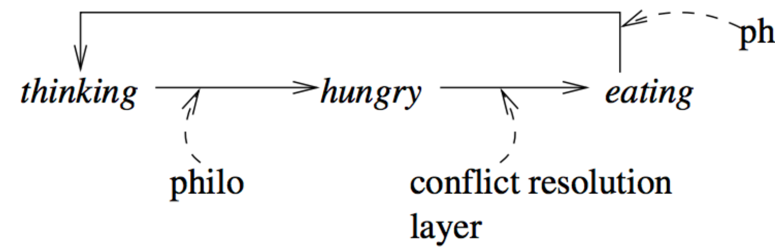


Chapter 8: Dining Philosophers (Refinement)

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



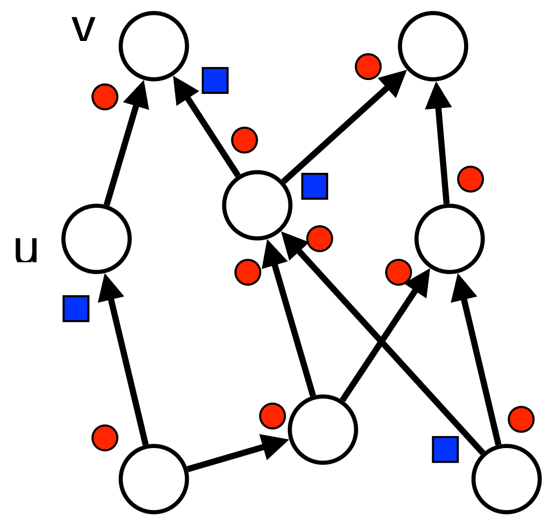


Program description

$$[H_p] \quad p.h \ \wedge \ fork(p,q) = q \longrightarrow req(p,q) := q;$$

$$[E_p] \quad p.h \ \wedge \ (\forall q \ : \ E(p,q) \ : \ fork(p,q) = p \ \wedge \ (clean(p,q) \ \vee \ req(p,q) = q)\ ) \longrightarrow p.state := eating; \ clean(p,q) := \textbf{false};$$

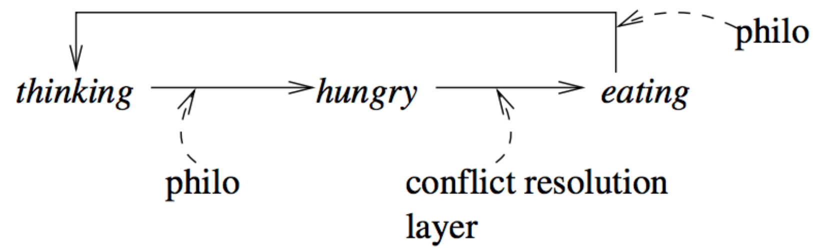
$$[R_p] \quad req(p,q) = p \ \wedge \ fork(p,q) = p \ \wedge \ \neg clean(p,q) \ \wedge \ \neg p.e \longrightarrow fork(p,q) := q; \ clean(p,q) := \neg clean(p,q);$$

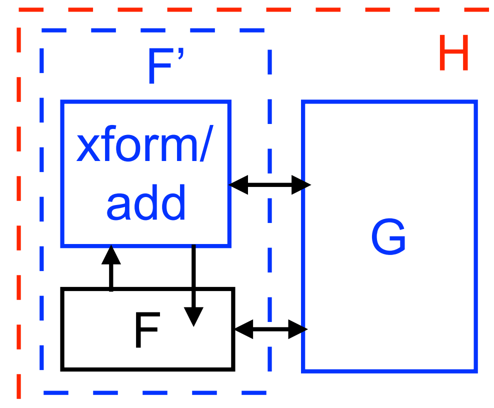


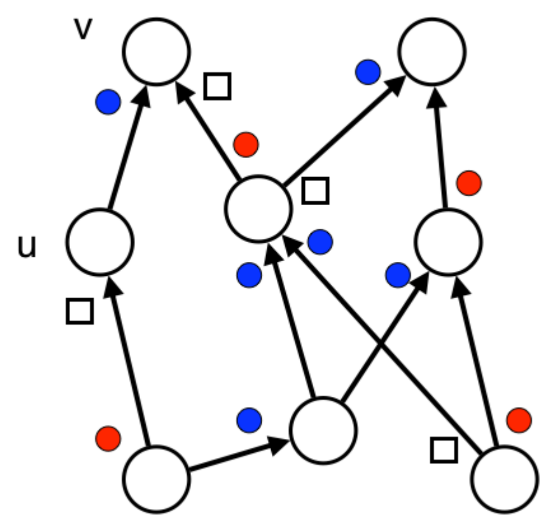
Chandy and Misra, Ch 7: Program Composition

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



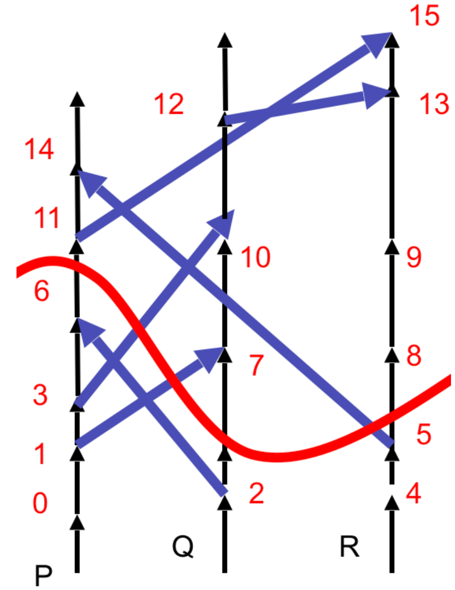




Key ideas

- Conditional properties: properties that are part of a "program" (P in F)
- Allow composition of programs P = F ▯ 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)

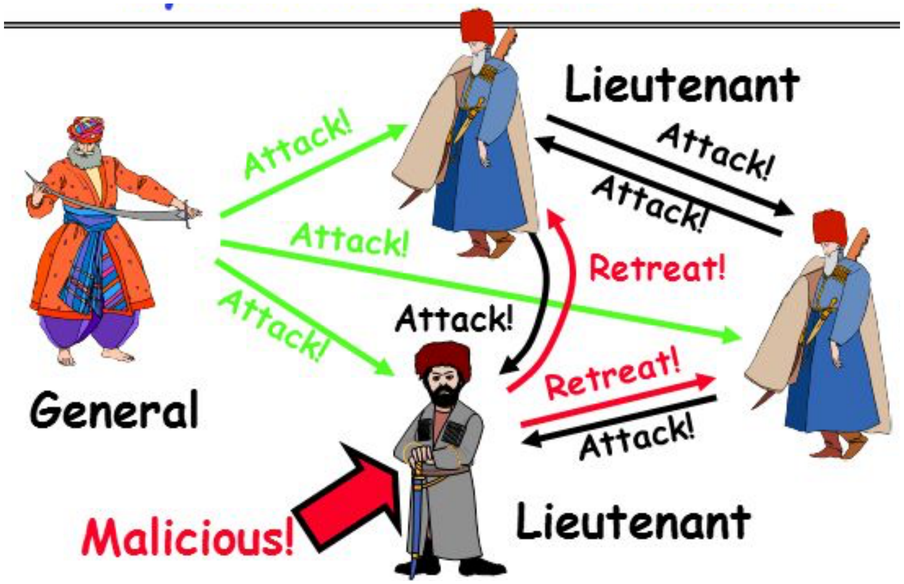
- Record the state of each process at the same logical time
- Keep track of messages that are still in flight (compare sent/recv counts)

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

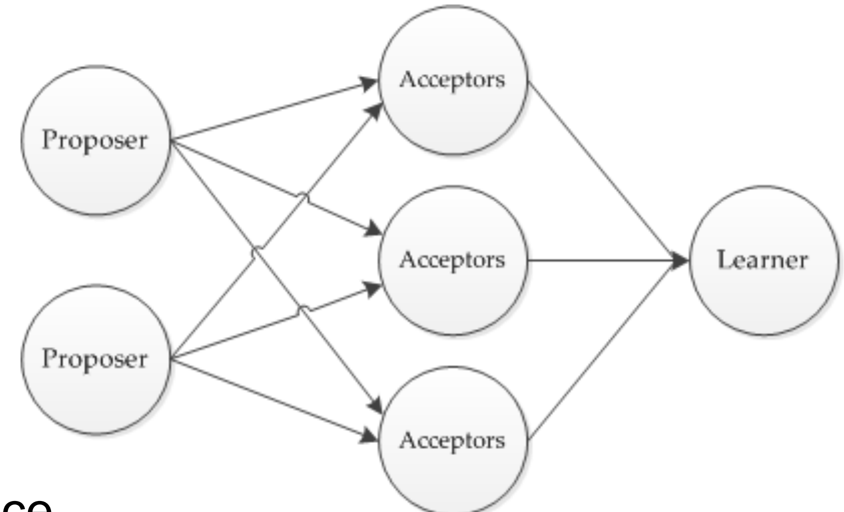
- Fail-stop: processor fails and others know
- Crash (fail-silent): failure w/out notification
- Byzantine: failed process can be malicious

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

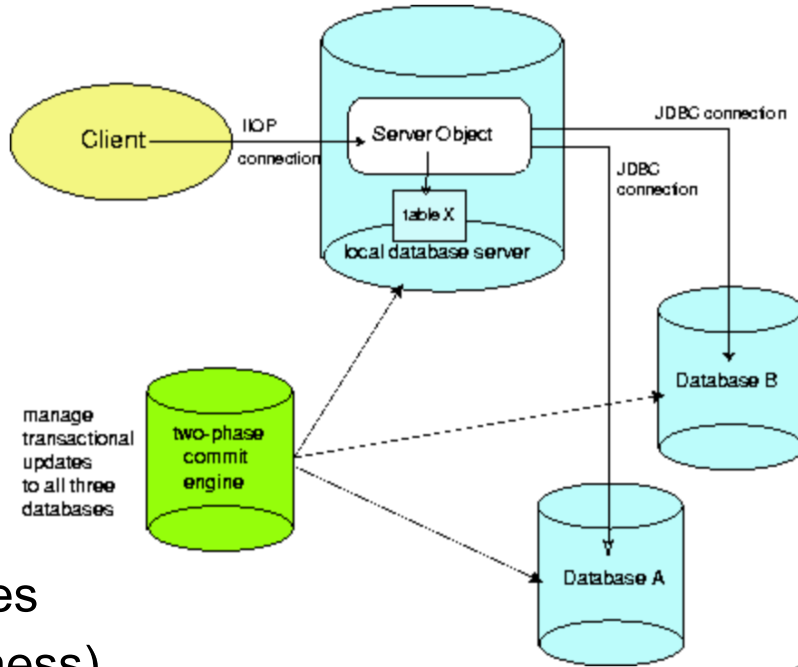
- Can only prove safety, but progress OK in practice

Bitcoin is a variant using proof-of-work + randomization + incentives

From Day 1: Introduction to Distributed Computing

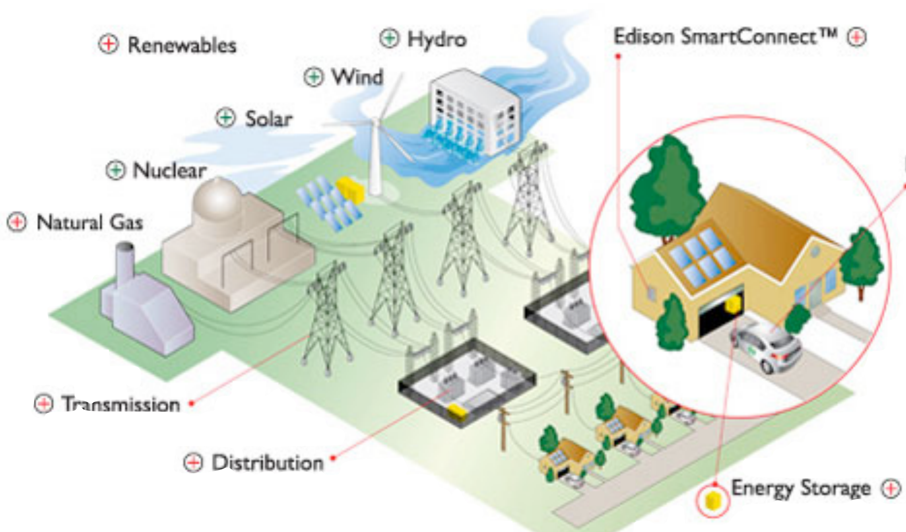
Main takeaway points

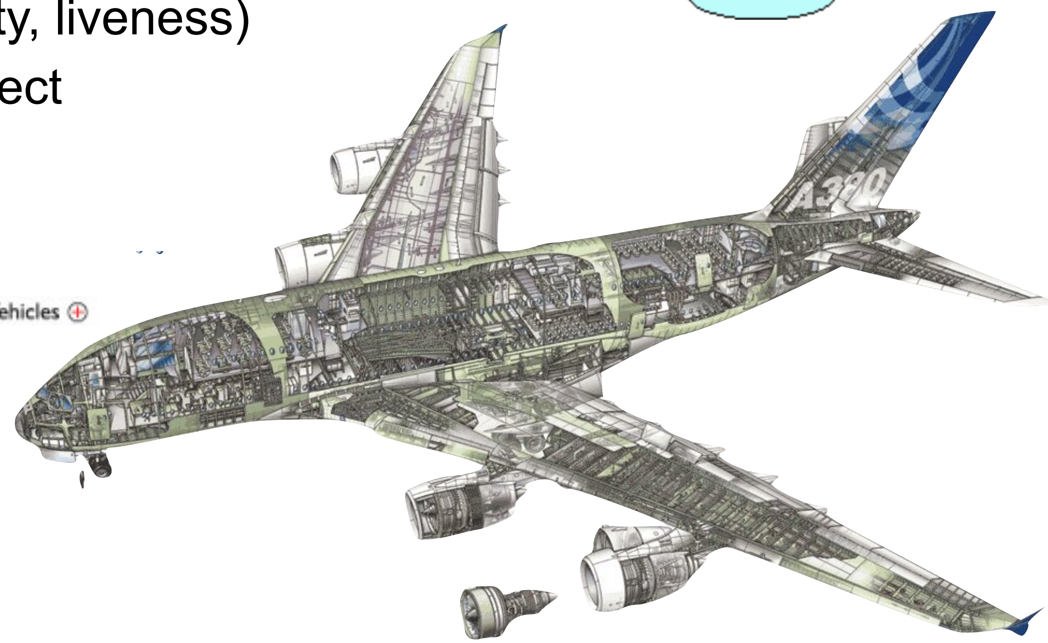
- Distributed systems (and hence distributed algorithms) are everywhere
- Debugging concurrent systems is much harder than debugging sequential programs
- For safety- (or business-) critical systems, formal proofs of correctness are key



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