
Reliability in heavily-connected systems
Jason Hickey
Computer Science

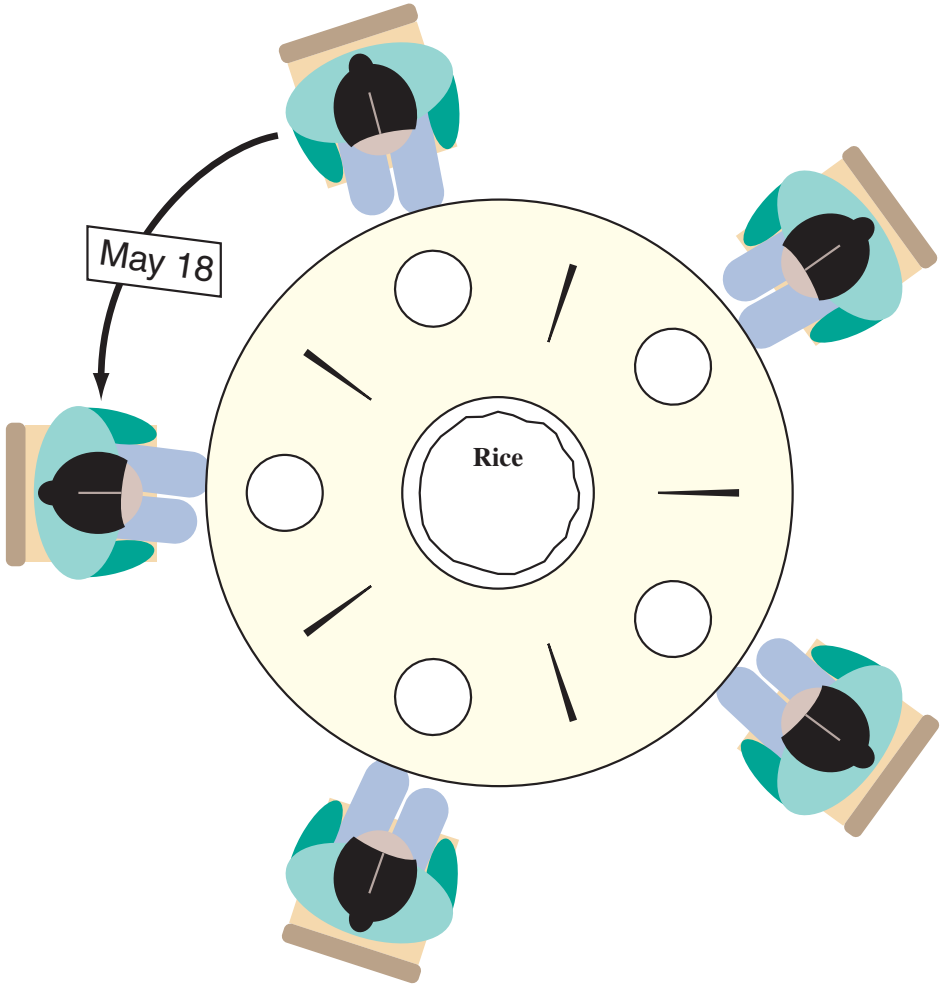
# *Reliability and Robustness*

- Languages, Models, and Style
- Reliability
  - Models of computation
  - Formal methods and analysis
- Robustness
- Rice's theorem: "Everything interesting about general programs is uncomputable"

# *Reliability*

- Scheduling
  - Problem: Find the earliest time for three people to meet
  - Algorithm: Send email with earliest possible meeting time, until the choice converges

# Scheduling




- Pass a note
- Each time, write next possible time on or after the time on the note

# Scheduling model

- UNITY (Chandy & Misra)
  - A program is a set of assignments
  - Assignments are evaluated in arbitrary, fair, order
- Three people F, G, H, with schedules f, g, h
  - f, g, h are monotonically increasing
  - “idempotent” (f(t) = f(f(t))

**Program** $P_2$

**initiall** y $t = 0$

**assign** $t \leftarrow f(t)$

$\mid t \leftarrow g(t)$

$\mid t \leftarrow h(t)$

**end**

# *Does it work?*

- Prove it
- Find the smallest t where t=f(t), t=g(t), t=h(t)
- An invariant: at all times, for any s<t, s is *not* a possible meeting time
  - Base case: trivial
  - Induction step: if f(t)>t, then all times s<f(t) are not possible meeting times

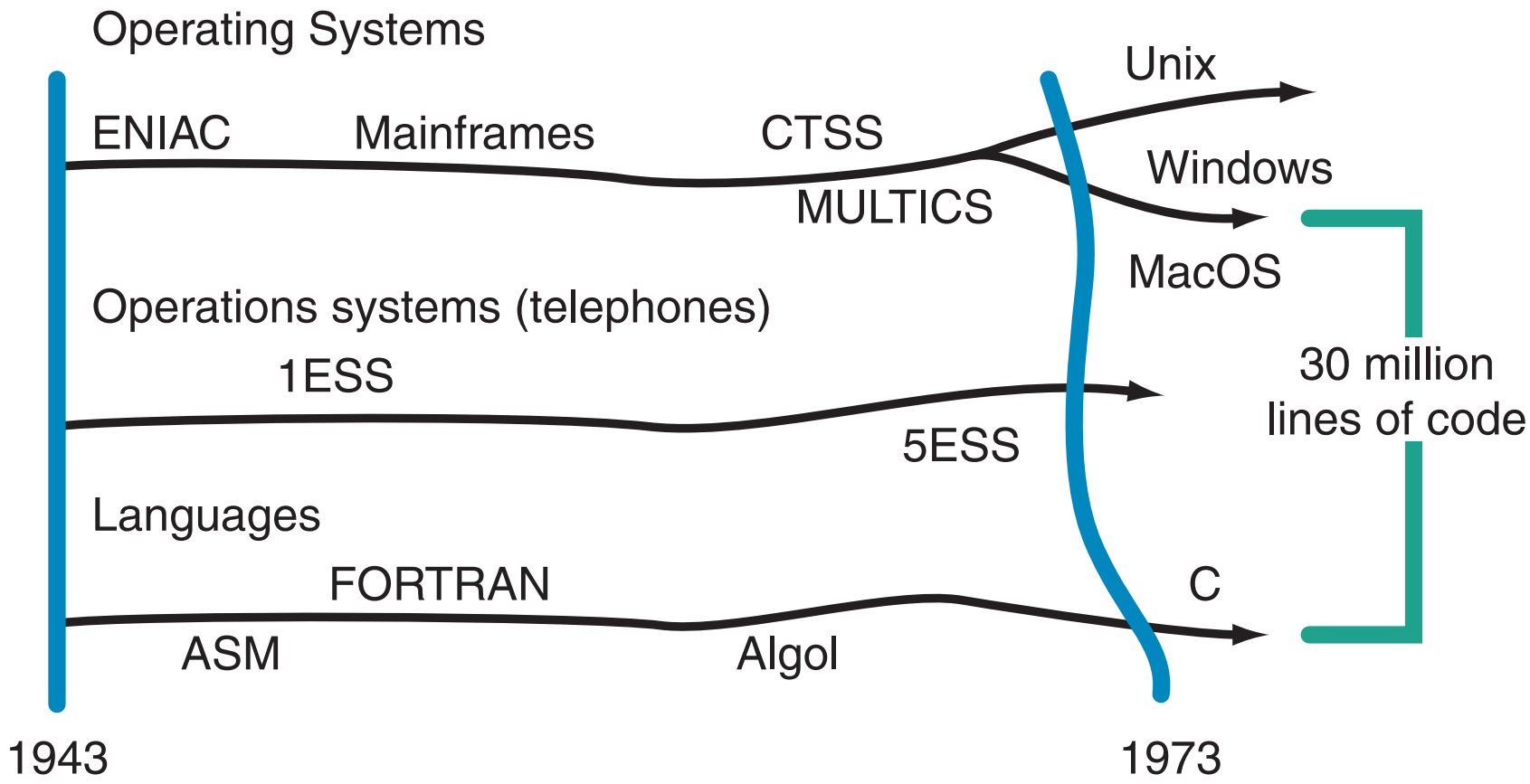
# Robustness

- Tweak the program by adding uncertainty

**Program** $P_2$
**initially** $t = 0$
**assign** $t \leftarrow f(t)$
$\quad | \; t \leftarrow g(t)$
$\quad | \; t \leftarrow h(t)$
$\quad | \; t \leftarrow t + 1$
**end**

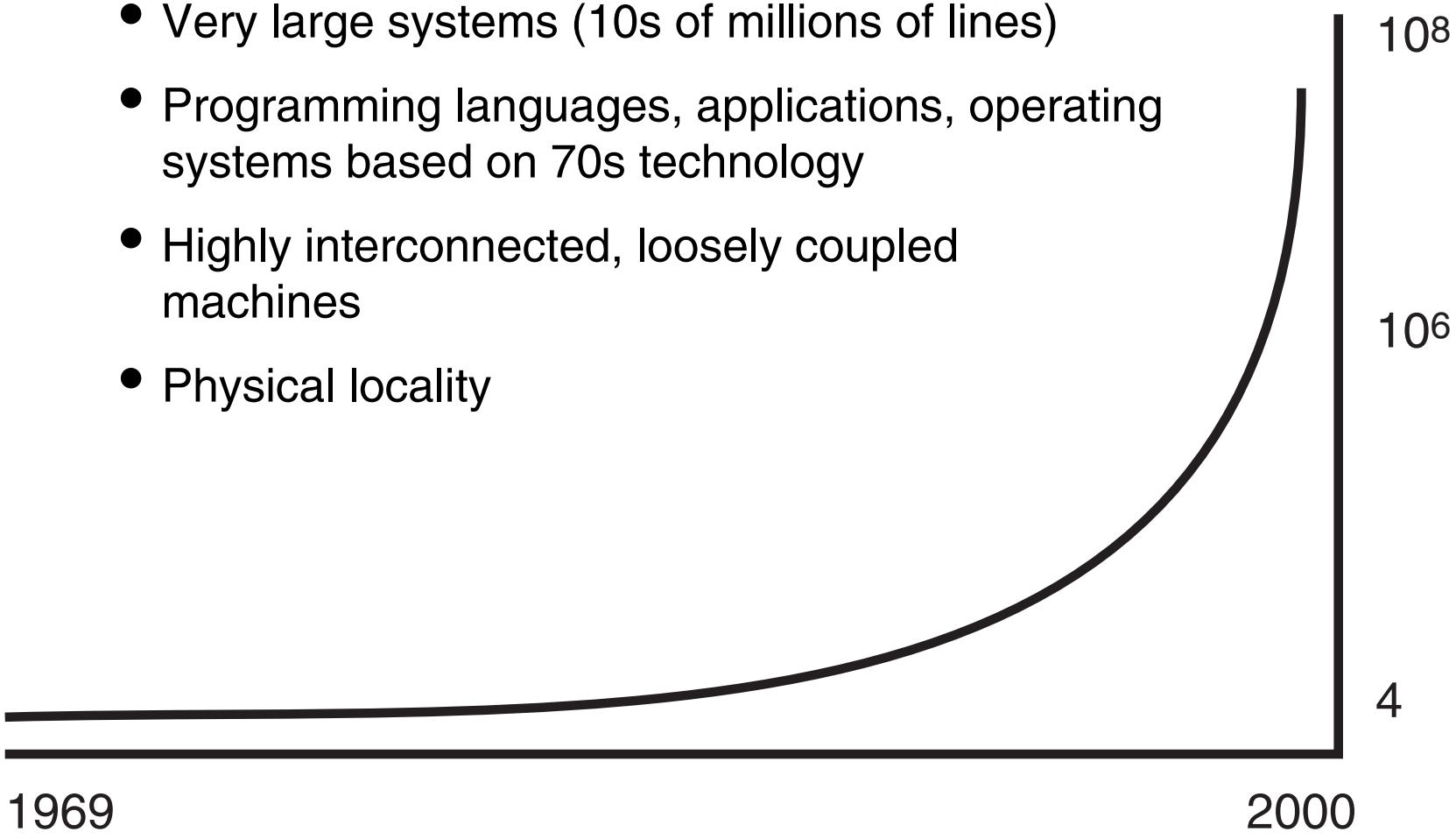
Wrong!

# *A simplified timeline*



# *Current state*

- Very large systems (10s of millions of lines)
- Programming languages, applications, operating systems based on 70s technology
- Highly interconnected, loosely coupled machines
- Physical locality



# *Distributed computing*

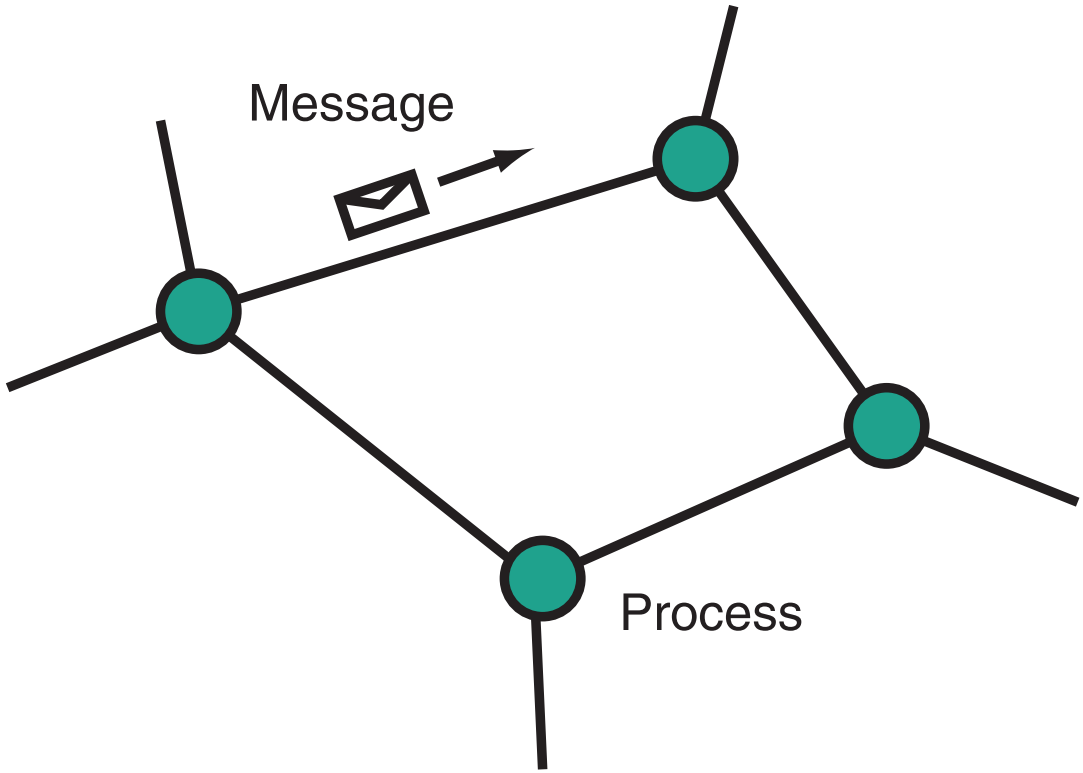
- Distributed programming is fundamentally different from serial programming
  - Computation is *isolated*
  - *Nondeterministic*
  - *Faulty*
- What are *models* of distributed programs?

# *Models*

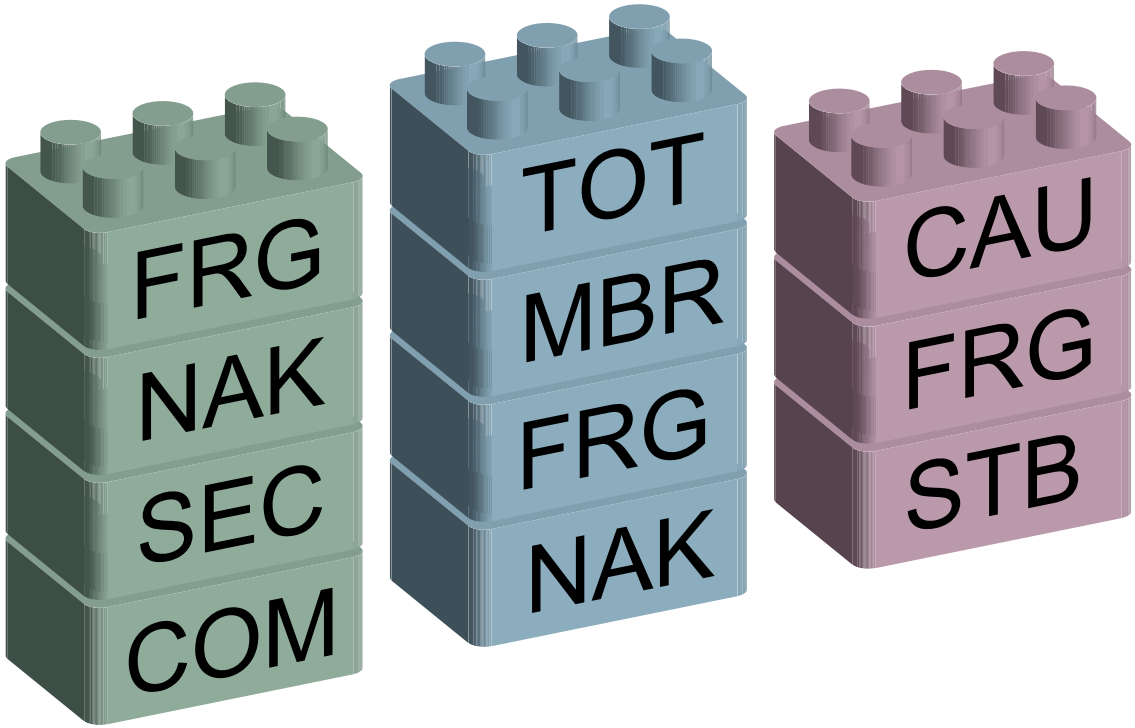
- A *model* is the mathematical concept that determines the structure and architecture of a system
  - UNITY: a system is a collection of assignment statements
  - Simple model; easy to prove safety of a system
  - Not compositional

# *Other models*

- CSP: sequential communicating processes
  - An assembly language of distributed programs

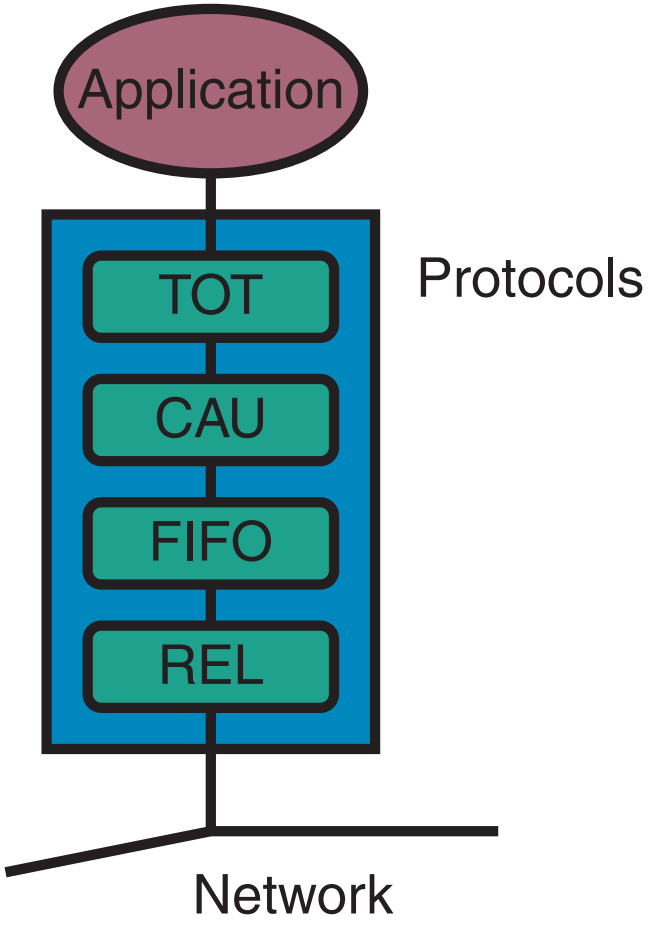


# *A better model*




- Modular “pluggable” components
- Compositionality guaranteed by the model

# *Applications from components*




- Protocols guarantee *virtual synchrony* between processes
- Each component has a set of *properties*
- Compositionality guarantees properties are preserved in the system

# *Processes are automata*

- Each process has a state
- A set of *actions* in reaction to the environment
- Safety properties are *invariants* of the automaton

- Each property of a system is defined by an automaton
- Virtual synchrony
  - All processes act in *logical* lock-step
  - Consistency is guaranteed
  - Implementation requires
    - FIFO message ordering
    - Synchronous fault detection

# A FIFO automaton

| FIFO | |
|---|---|
| Actions: | $\text{SEND}(m)$, $\text{RECV}(m)$, for $m \in \mathcal{M}$ |
| State: | $sent \in \mathcal{M}$ *List*, initially empty, |
| | $received \in \mathcal{M}$ *List*, initially empty |
| $\text{SEND}(m)$ | |
| Eff: append $m$ to $sent$ | |
| $\text{RECV}(m)$ | |
| Pre: $\lvert received \rvert < \lvert sent \rvert$ | |
| $sent[\lvert received \rvert + 1] = m$ | |
| Eff: append $m$ to $received$ | |

# A VIEW automaton

EVS_VIEW

State: for each $p \in PID$:
$all\text{-}viewids_p \in View\ Set$, initially $\{v_p\}$

EVS-NEWVIEW$_p(v)$

Pre: let $v' = current\text{-}view_p$ in
$v.id > v'.id$ 1h
$p \in v.set$ 2h
$\forall q \in v.set.$ 3h
if $pred\text{-}view_{q,v} \neq \perp$ then
$pred\text{-}view_{q,v} = v' \vee pred\text{-}view_{q,v}.set \cap v'.set = \{\}$

Eff: $all\text{-}views_p = all\text{-}views_p \cup \{v\}$

NUPRL

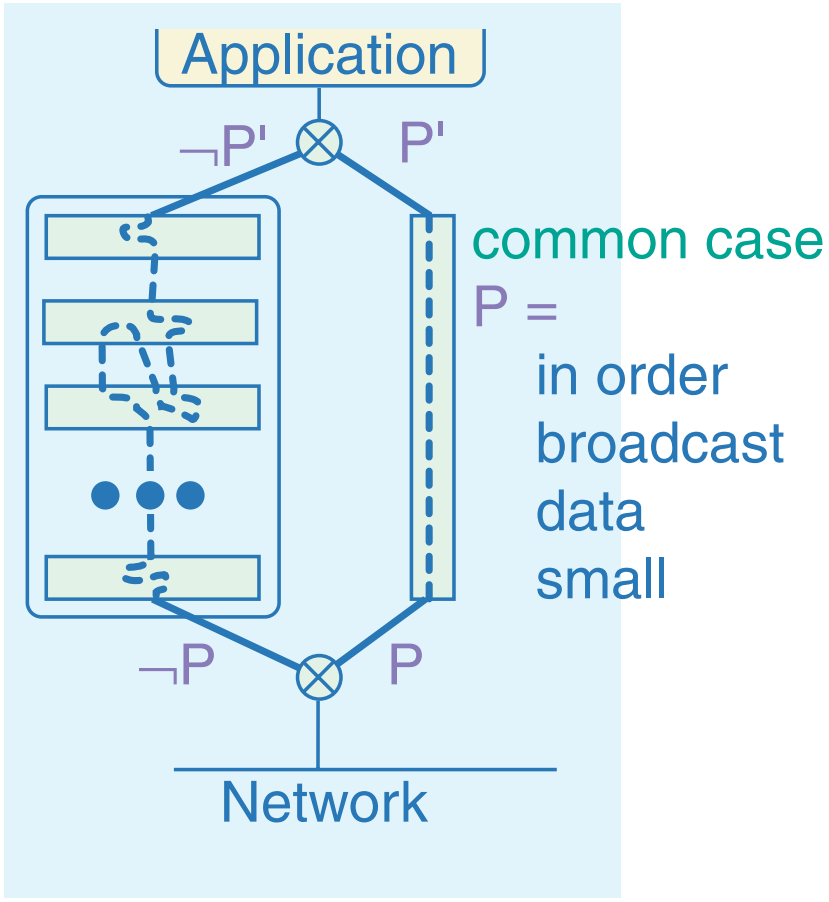
# Complete virtual synchrony

$$EVS \equiv \text{GROUP\_FIFO} \cap \text{VIEW} \cap \text{VIEW\_MSG}$$

NUPRL

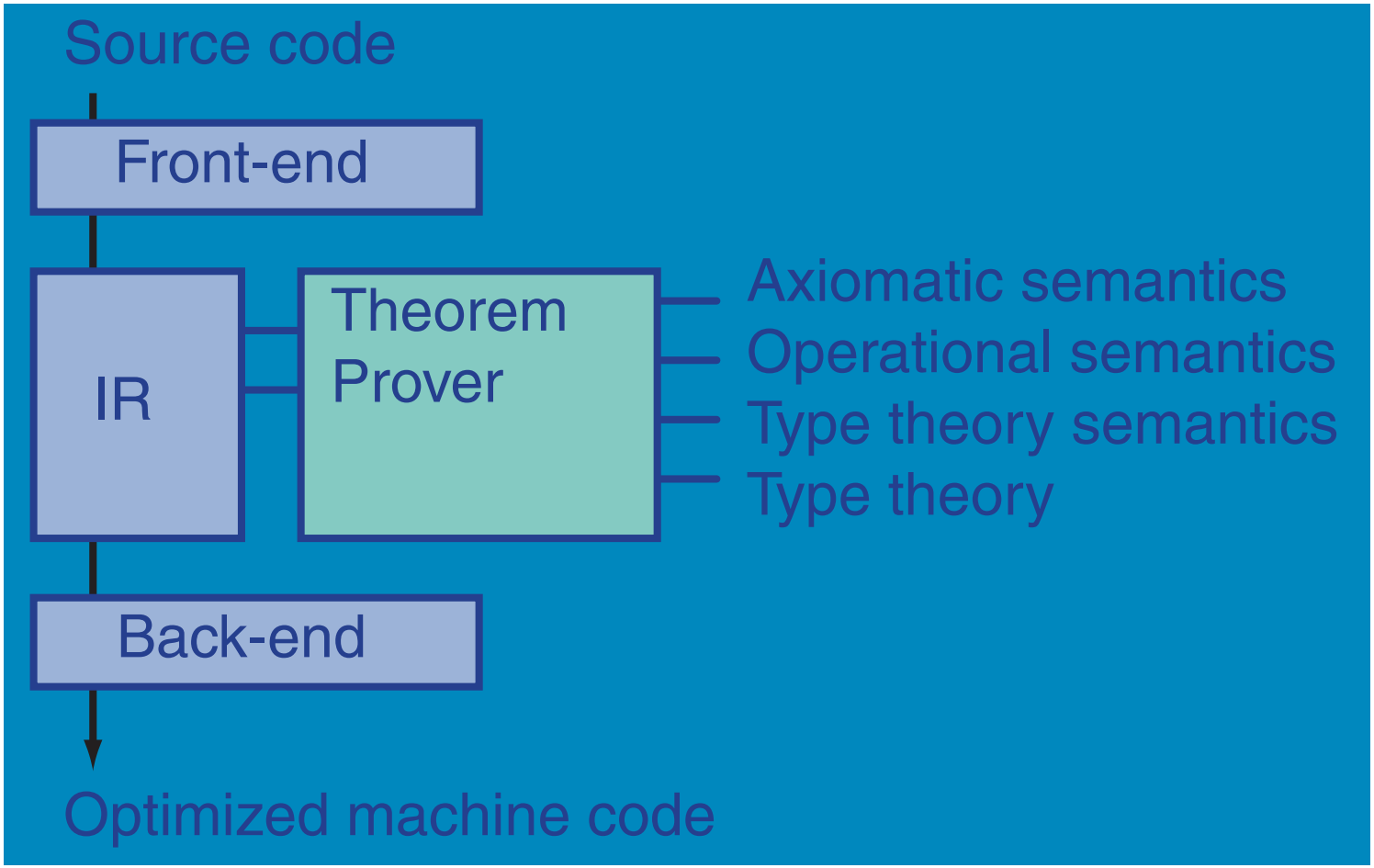
- The mathematics guarantees that the composed system preserves properties
  - *All* processes observe FIFO message order
  - *All* processes observe the same set of faults

# *Formal automation*




- Protocols are pluggable components
- ~70 components, 1000s protocols

# *Extensible compilers*



# *Problems and paths*

- Formal methods provide tools
  - *models* to help understand problems
  - *compilers* to automatically generate code
  - *guarantees* about reliability
- But, the tools are hard to use
  - Deep knowledge of logic and semantics
- How do we apply the knowledge?

# *FC: a "functional" C compiler*

- C programs are ubiquitous
  - But they are poorly understood
  - They have weak properties
  - Not compositional
- But they are everywhere
- FC: provide a formal foundation for C programs

# FC

- Model: the λ-calculus
  - Simple
  - Functional
  - Safe

$$e \quad ::= \quad v \mid e_1(e_2) \mid \lambda v.e$$

$$(\lambda v.e_1)\ e_2 \longrightarrow e_1[e_2/v]$$

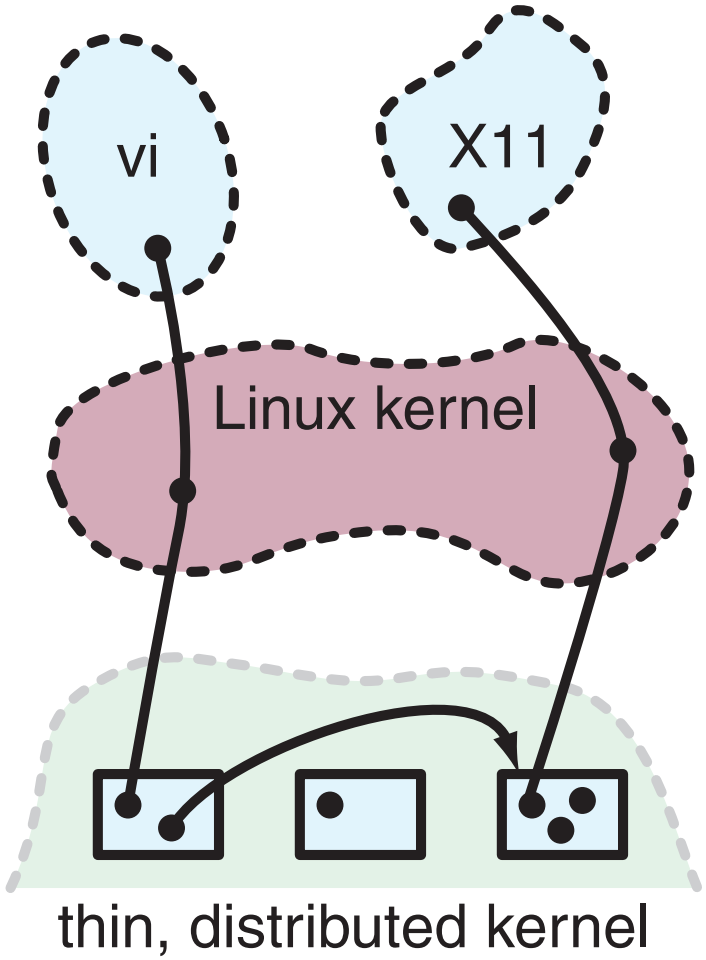
- Compiler is formal; it can be extended

# *Extensions*

- Programs are *safe*
  - No program accesses memory that it does not own,
  - No program executes code that it does not own
  - (Programs may still self-destruct)
- C allows
  - arbitrary coercion
  - pointer arithmetic
- The critical step in to introduce checking (some of it at run time)

# *Safety*

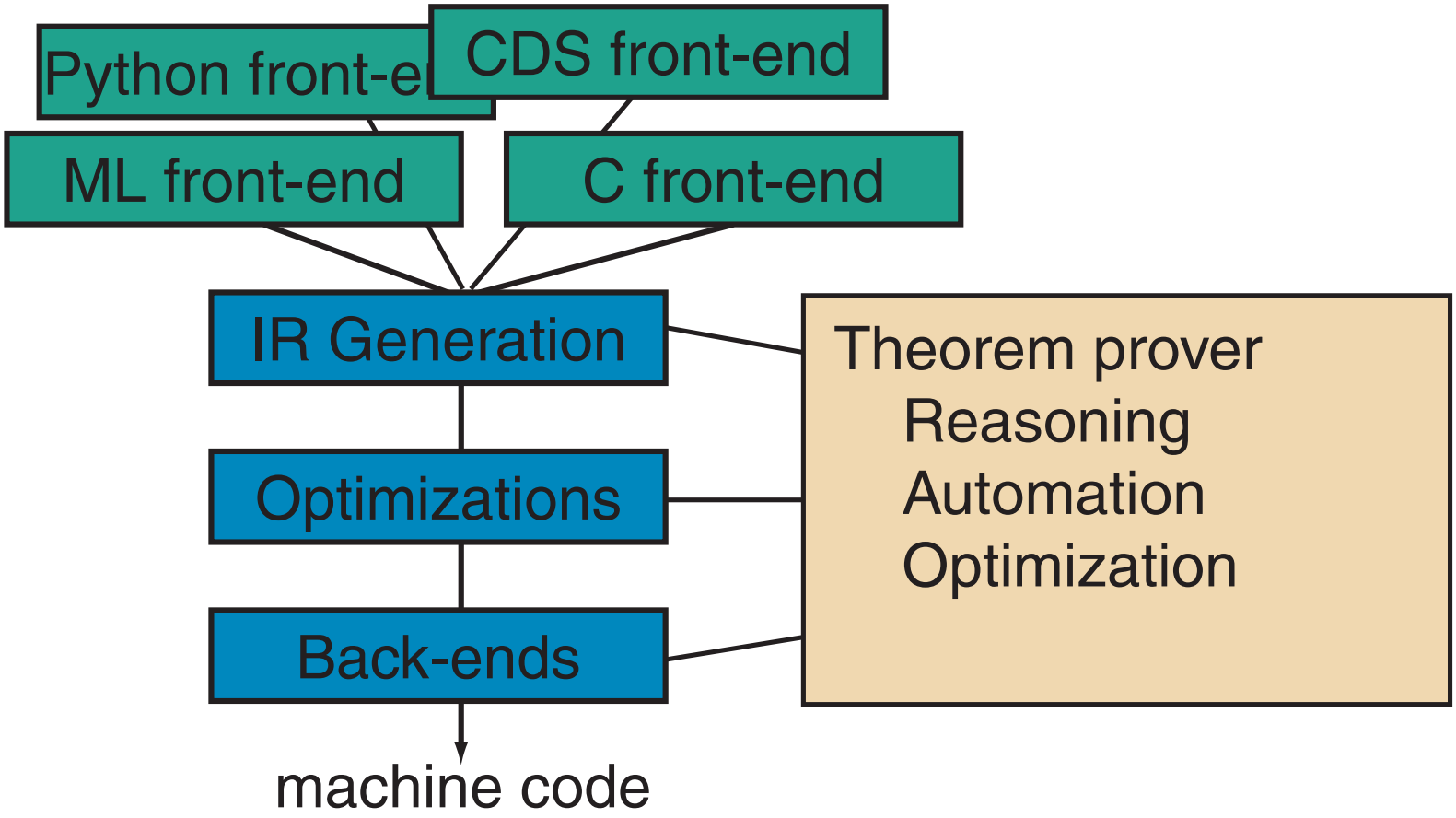
- No need for the kernel-user distinction
- OS can be stripped down all the way to the hardware
- System design is a matter of choice (and performance)



# *Distributed systems*

- The *problem* was how to design and use heavily-connected distributed systems (and maintain reliability)
- Three parts:
  - Process mobility: processes *should* migrate to resources
  - Distributed *virtual* synchronous operations
  - Heterogeneous multi-language environments

# *Multi-language environments*

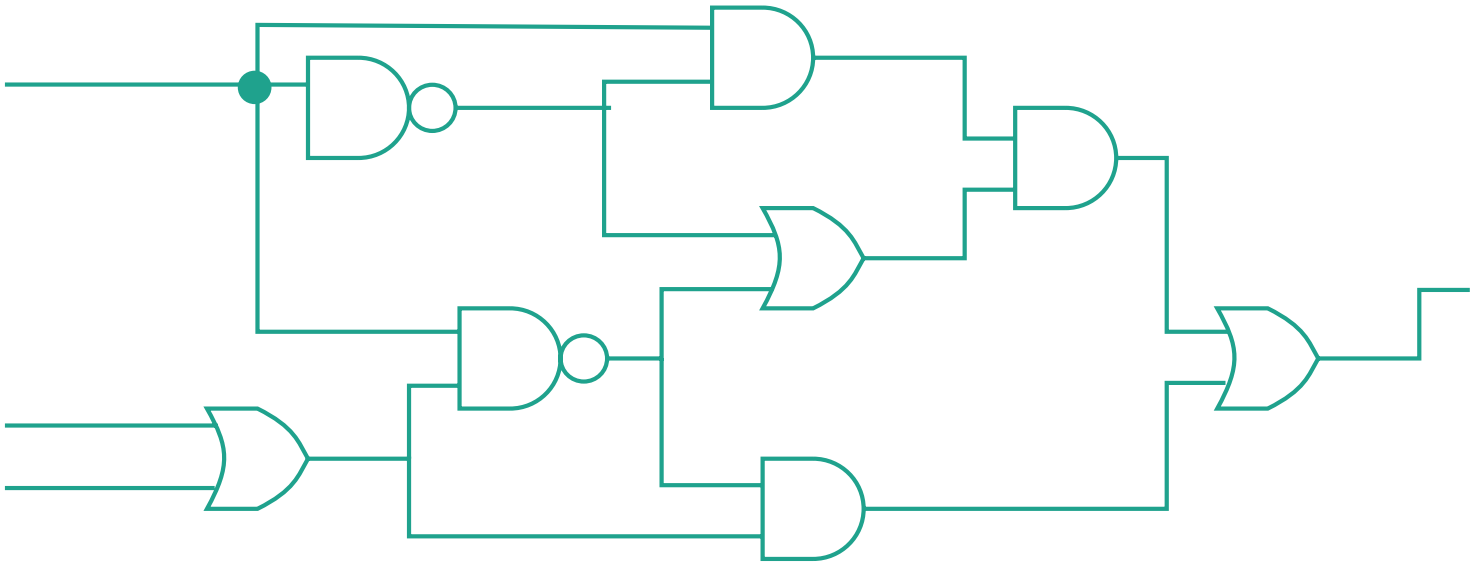


# *Robustness*

- A definition: the result of a computation should change only a little if
  - the input is changed a little
  - the program is changed a little
- Topological definition
  - The program should define a continuous function

# *P-Time functions*

- A P-time complete problem is feed-forward circuit evaluation



# *NP-hard functions*

- 3SAT: determine if there is a steady-state of a circuit with feedback
- There is no reasonable topology in which to define continuity (unless P=NP)
  - NP-hard problems are not robust by definition