Department of Computer Science

PURDUE UNIVERSITY

# CS505: Distributed Systems

## Lecture 4: System Models and Reasoning about Basic Communication

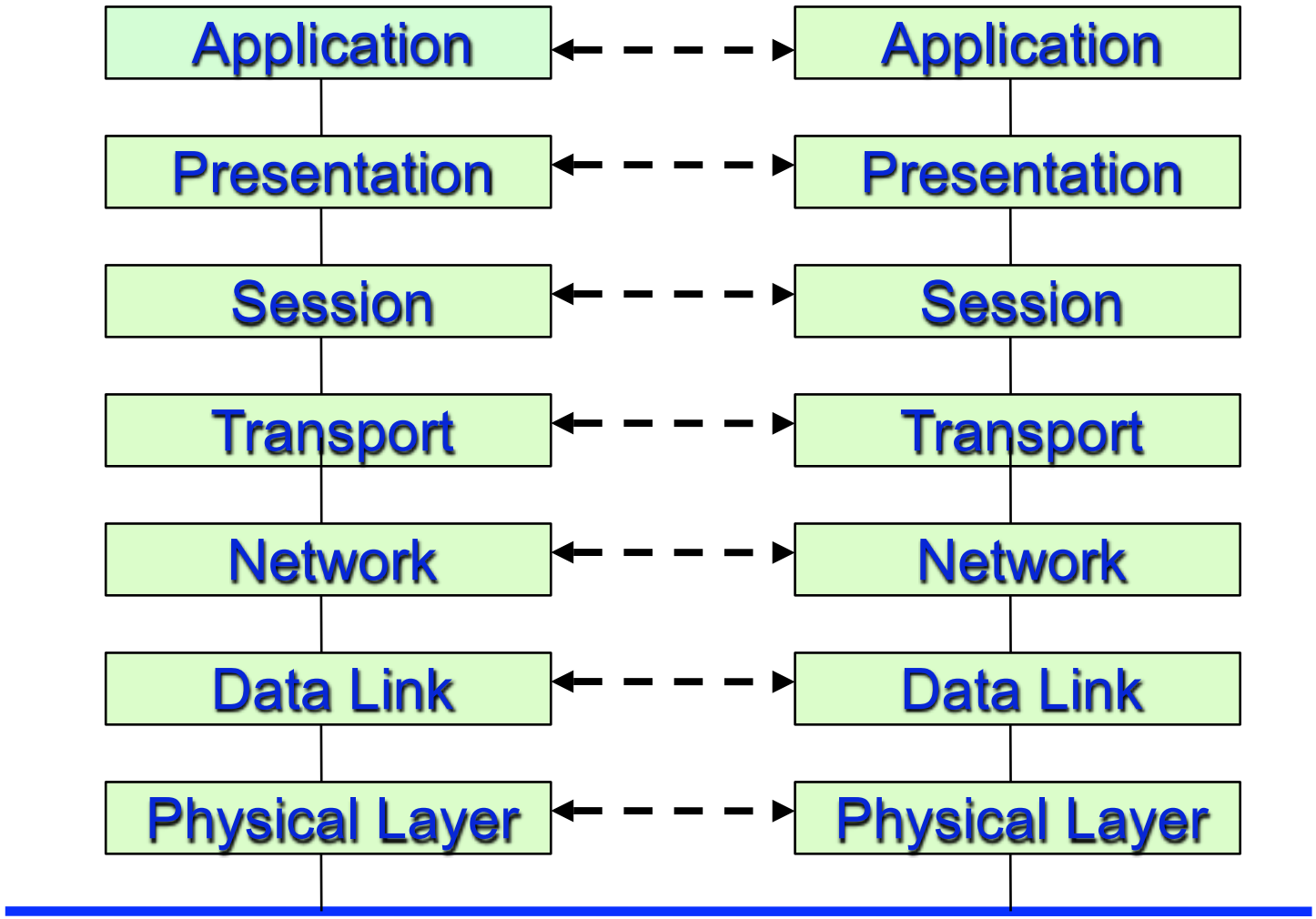
# Overview

- System models and failures
- One-to-one communication

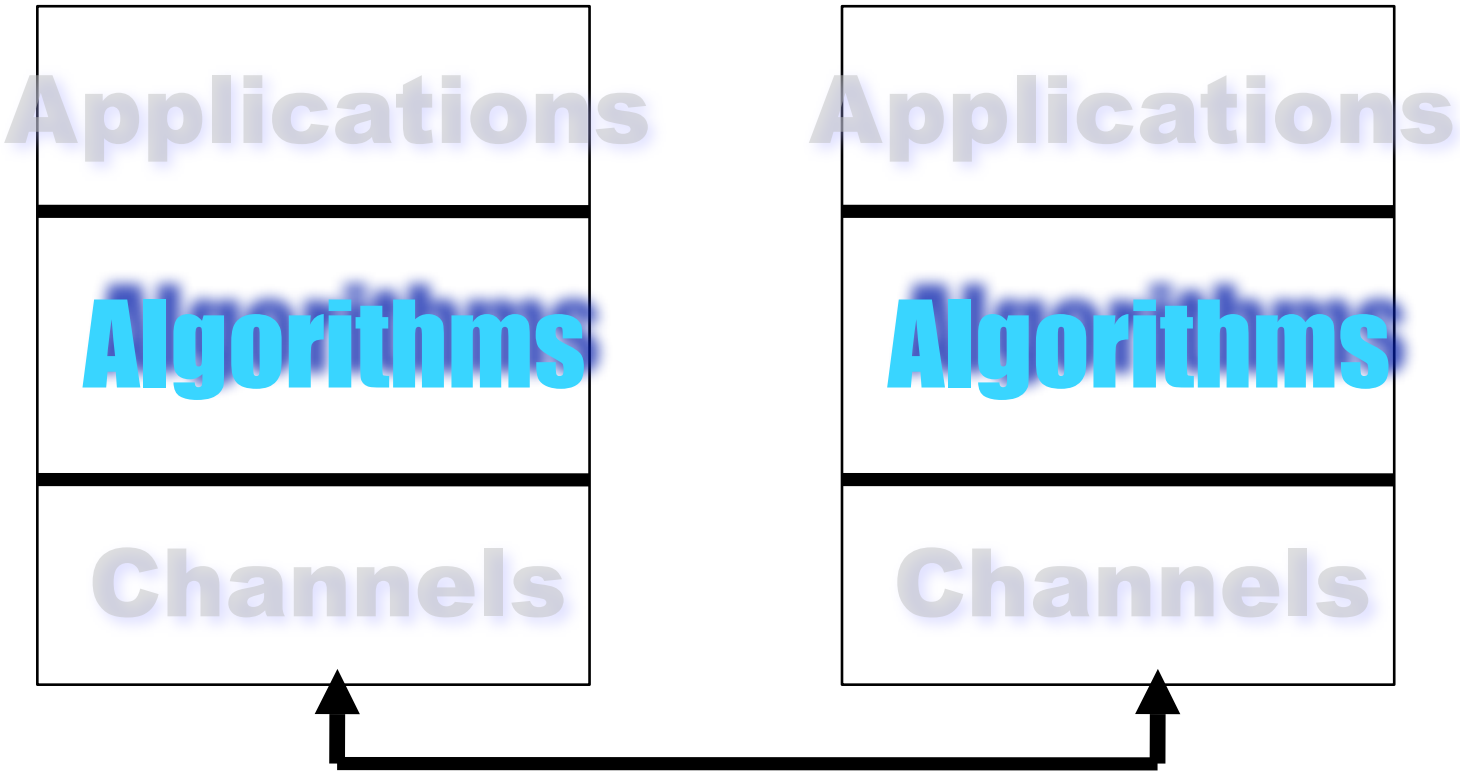
# Distributed Systems

- The network is not enough
  - Reliability guarantees (e.g., TCP) are only offered for communication among pairs of processes, i.e., one-to-one communication (client-server)
  - Reliability guarantees are only informal
- The application needs underlying services for reliable one-to-many and many-to-many interaction

# OSI/ISO Model



# Distributed Systems



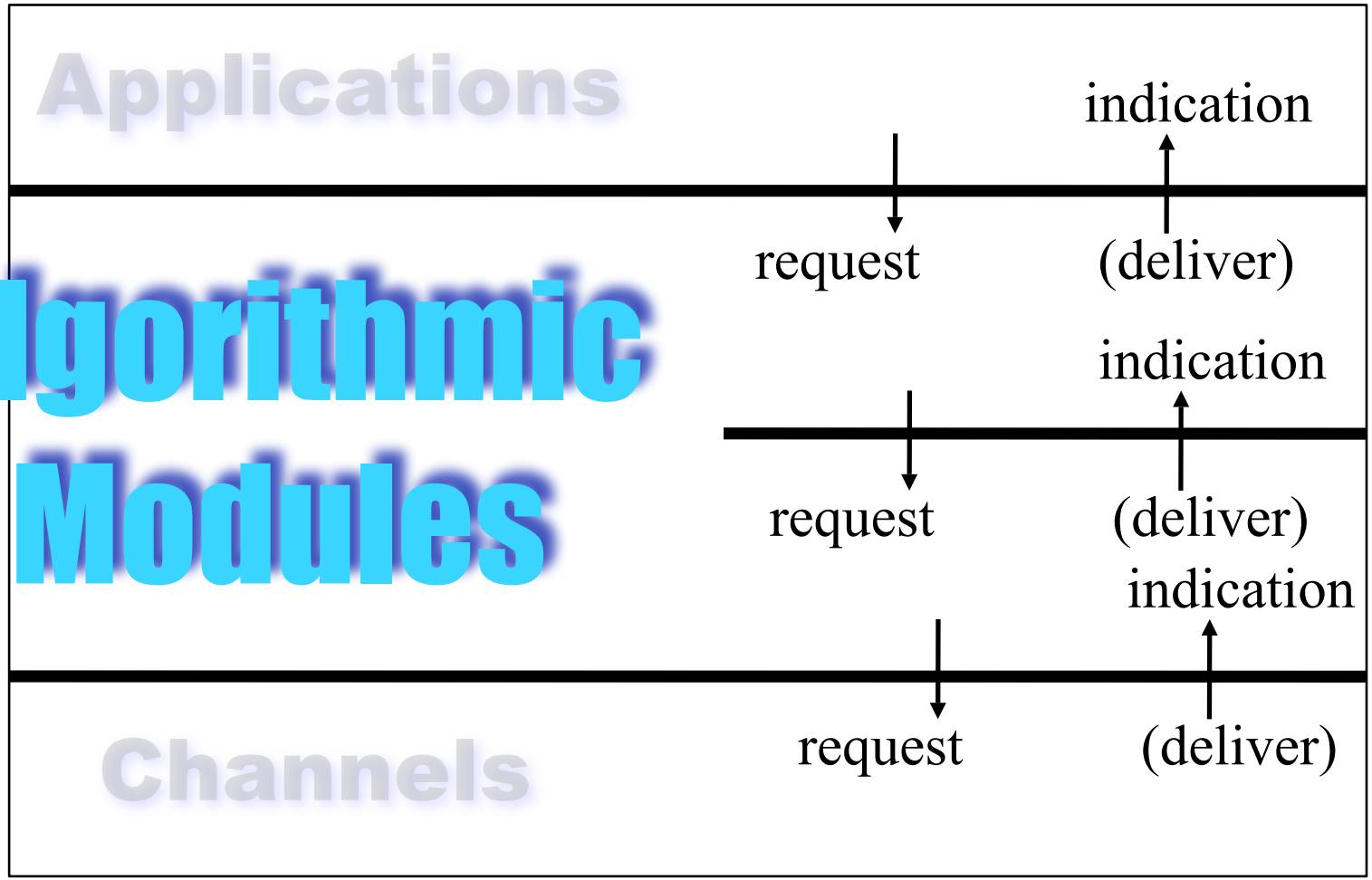
# Layered View

- **Channel layer**
  - Corresponds to transport layer
- **Algorithm layer**
  - Corresponds to anything above, below application layer
- **Application layer**
  - Corresponds to OSI application layer

# Planar View

- Distributed system consists of a set of processes $\Pi = \{p_1, p_2, \ldots\}$
- Processes execute state automata (state machine)
- Communicate by some means
  - Message passing (network): pair-wise communication channels
  - Shared memory: registers

# Modules


Applications
indication
request
(deliver)
Algorithmic Modules
indication
request
(deliver)
indication
request
(deliver)
Channels

# Specifications

- Reasoning about fault tolerance requires
  - Model of system
  - Model of faults
- Every component (module/layer/state machine) is specified in terms of properties
  - Separated as far as possible
  - Primary classification
    - *Safety*
    - *Liveness*

# Liveness and Safety

- Safety is a property which states that nothing bad should happen
- Liveness is a property which states that something good should happen

- Any specification can be expressed in terms of liveness and safety properties [Lamport and Schneider'84]

# Examples

- **Telling the truth**
  - Not lying is safety
  - Having to say something is liveness
- **Following the class**
  - Not listening to anything/-one else than the lecturer is safety
  - Not looking at anything/-one else than the lecturer is safety
  - Being awake and alert is liveness

# Asynchronous System Model

- Typical of Internet
- No bounds on relative speeds of processes
  - Interruptions, multi-tasking, diverging architectures
- No bounds on message transmission delays
  - Cf. exponential back-off in Ethernet, multiplicative decrease in TCP/IP
- No synchronized clocks (no bounds on clock drifts)
  - Synchronization protocols exist, usually not relevant for correctness but more for efficiency

# Further

- Usually some “hidden” assumptions, e.g.,
  - Unique identifiers for processes
    - unique Ethernet addresses; how about wireless settings?
    - NAT, DNS abstracted
  - A priori, existence of some (independent) path between any two processes
    - Network topology, routers, abstracted
  - Unique identifiers for messages

# Failures

- Failure model can be seen as “adversary”
- Halting failures
  - Crash-stop
  - Fail-stop
  - Crash-recovery
- Omission failures
  - Send
  - Receive
- Byzantine failures
  - With message authentication
  - W/o: sender can be faked

# Example

- Crash-recovery: specific case of omissions
  - A process loses its *volatile* storage state when it crashes (amnesia)
  - The process has a *stable* storage that it does not lose
  - NB. The challenge here is to devise algorithms that use few accesses to stable storage, and little communication (storage can be traded against communication)

# Examples

- **Crash-stop: a more specific case of omissions**
  - **A process that omits a message to a process, omits all subsequent messages to all processes (permanent distraction): it crashes**
  - **No need for stable storage**
  - A ***correct*** process is a process that does not fail (that does not crash)
- **Fail-stop**
  - **Like crash-stop**
  - **Failure is however (accurately) detectable**

# Duration

- **Transient failures**
  - Overcome eventually (within the considered time/algorithm run)
- **Intermittent failures**
  - Repeated
- **Permanent failures**
  - No recovery

# Causes

1. Fault (e.g., dead memory cell)
2. Error (e.g., wrong program state; 0 instead of 1)
3. Failure (program misbehaves/crashes)

- "Software failures"
  - Caused by fault/bug in software
- "Hardware failures"
  - Caused by power outages, terrorist attacks, …
- Byzantine failures
  - Hackers, …

# Consolidated Model [Gaertner'98]

- A program is viewed as a state machine
  - The state is a configuration
  - An execution is an infinite sequence of configurations
  - Transition by execution of single guarded statement
  - A program property is (described by) a set of executions
  - A specific property holds if contained in/ensured by the sequences

# Properties and Correctness

- Safety property is described by legal set of configurations (*invariant*)
- Liveness property is one for which every partial execution is *live*
  - A partial execution is live for a property *p* if it can be extended to remain in *p*
- A problem specification is defined by safety and liveness properties
- An algorithm *A* is correct regarding a problem specification if properties hold for *A*

# Fault Tolerance

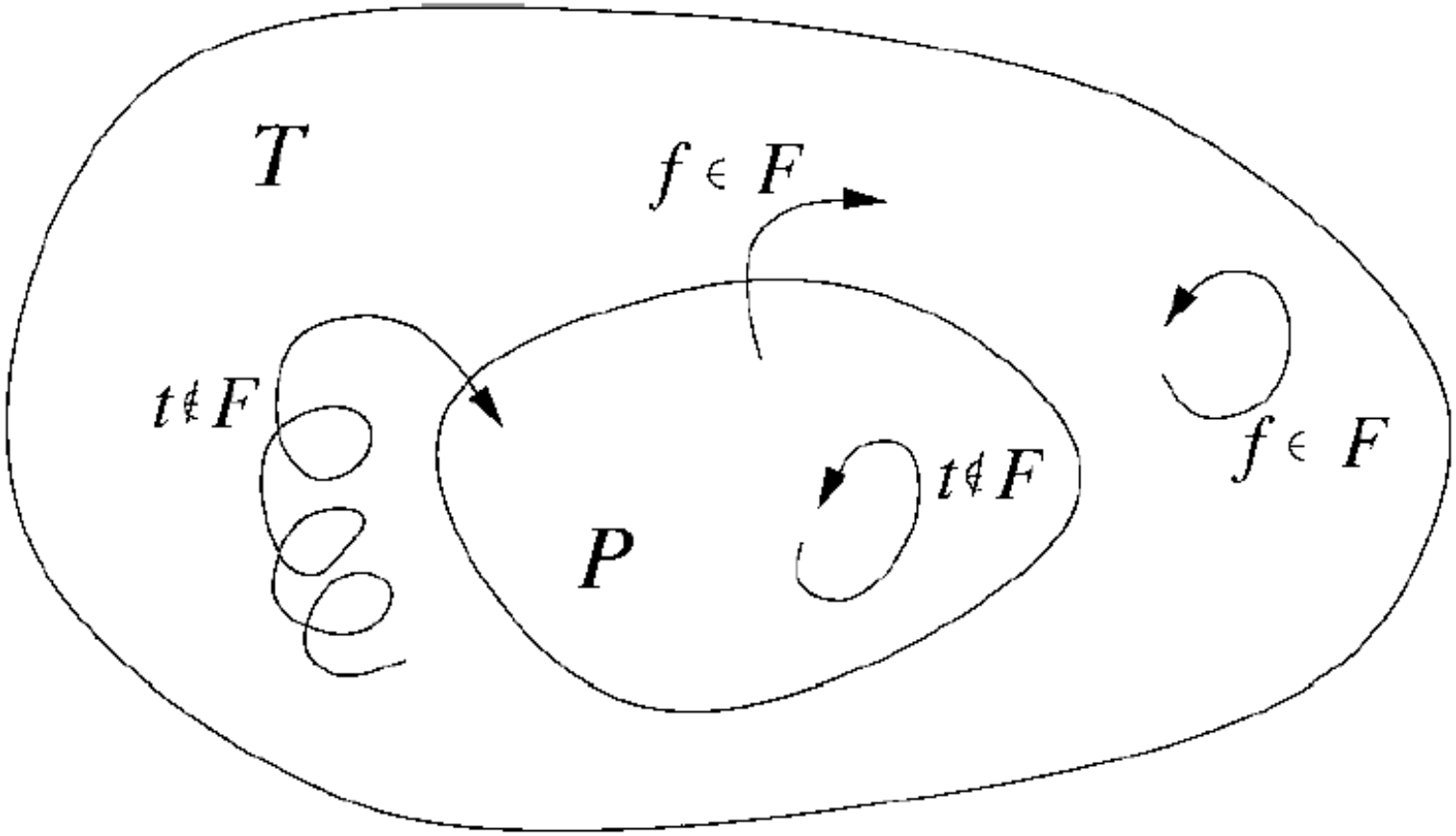
A distributed program *A* tolerates faults from class *F* for an invariant *P* ⇔ exists predicate *T* s.t.

1. At any configuration where *P* holds, *T* also holds (*P*=> *T*)

- Starting from any state where *T* holds, if any actions of *A* or *F* are executed, the resulting state will always be one in which *T* holds (i.e., *T* is closed in *A* and *T* is closed in *F*).
- Starting from any state where *T* holds, every computation that executes actions from *A* alone eventually reaches a state where *P* holds.

If a program *A* tolerates faults from a fault class *F* for invariant *P*, we say that *A* is *F*-tolerant for *P*.

# Graphic Representation

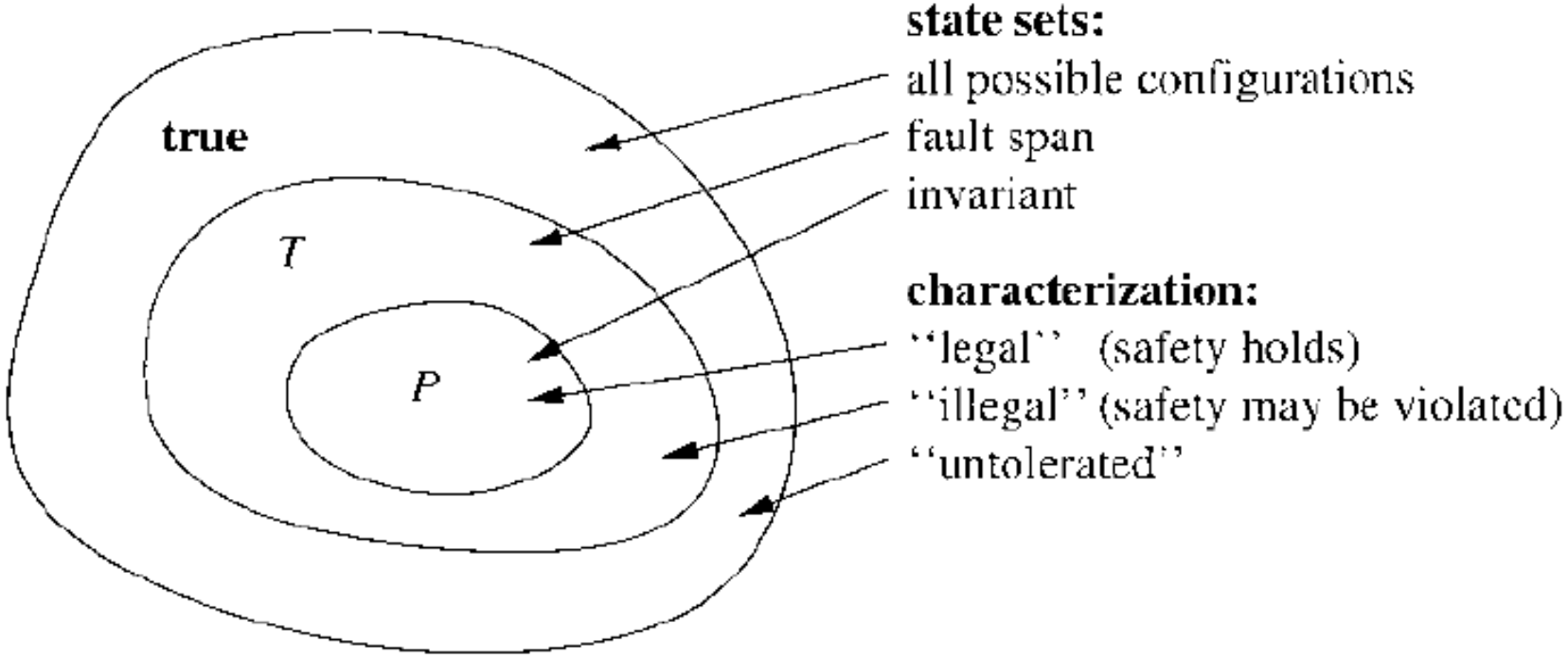


# Intuitively

- Faults: unwanted ("illegal") but possible state transitions (considered in fault model) leading to configurations in $T \setminus P$
- Failure: deviation from specification
  - $T \setminus P$: safety violated
- Modular decomposition/layers leads to
  1. Fault
  2. Failure
  3. Fault
  4. …

# Forms of Fault Tolerance

| | Live | Not live |
|---|---|---|
| Safe (*T*=*P*) | *Masking* | *Fail safe* |
| Not safe | *Non masking* | *None* |


true
T
P
state sets:
all possible configurations
fault span
invariant
characterization:
"legal" (safety holds)
"illegal" (safety may be violated)
"untolerated"

# Redundancy

- Primary means for fault tolerance
- Two categories
  - *Time*: program defines “additional actions” w.r.t. fault-free executions
  - *Space*: program A defines “additional configurations” w.r.t. fault-free executions
- Redundancy necessary but not sufficient for fault tolerance
  - Every useful program contains redundancy in space

# Example

- Consider a program switching between `x=1` and `x=2`
- Variable(s): `x` ∈ `{0,1,2,3}`
- Possible state transitions
  - `x = 1 -> x := 2`
  - `x = 2 -> x := 1`
  - `true -> x := 0` models fault class F
  - `x = 0 -> x:= 1`

- *P*: `x ∈ {1,2}`
  - Liveness?
- *T*: `x ∈ {0,1,2}`
- Redundancy?

# Concrete Programs

- How does this translate to “real” programs?
  - E.g., types, classes, …
- What about exceptions?

# Communication

- No bounds on communication latency
  - What does this imply?
- How about reliability?
- Common guarantees
  - Best effort
  - At-most-once
  - At-least-once
  - Exactly-once

# Typically

- TCP/IP
  - FIFO, at-most-once
  - Exactly-once *eventually*, if processes correct
    - When primitive send returns normally we know message has been received
    - But no time limits
- General assumption
  - Channels bidirectional
  - Same guarantees in both directions

# Basic Channel

Defined by primitives `send` and `receive`

I. No duplication
   - A message is `received` at most once

II. No creation
   - No message is `received` unless some process did `send` it

- Assumptions/requirements?

# Reliable Channel

I., II., and

III. Validity

- If a process $p_i$ `sends` a message `m` to a process $p_j$ and both are correct then $p_j$ *eventually* `receives` `m`

- Usually paired with FIFO order
- Assumptions/requirements/consequences?
  - Buffering?

# How About

I., II., and

IV. Validity

- If a process $p_i$ `sends` a message `m` to a *correct* process $p_j$, then $p_j$ *eventually* `receives` `m`

- ?

# Fair-lossy Channel

I., II., and

V. Fair losses

- If a process $p_i$ `sends` a message `m` to a *correct* process $p_j$ infinitely many times, then $p_j$ *eventually* `receives` `m` infinitely many times

- Assumptions/requirements/consequences?
- Vs reliable channel?

# Best Effort

I., II., and

VI. Validity

- If a process $p_i$ `sends` a message `m` to a process $p_j$ and neither suspects the other then $p_j$ *eventually* `receives` `m`

- Captures notion of timeout

# Quiescence

- With reliable channels
  - Processes can end up resending infinitely
  - Thus buffer infinitely, in particular in case of failures
  - Need point to define when can stop
- In asynchronous systems no possibility to avoid
- Much ado about nothing?
  - FIFO with a notion of message obsolescence

# Tradeoffs

- *Some* assumptions are necessary for useful systems
  - E.g., unique identifiers
  - Synchrony, failures
    - Can be reduced also by choices in architecture (hardware, "software")
- Failures/synchrony can be traded in assumptions
  - E.g., timing failures: one assumes *by default* bounds, but introduces failures to reflect violation of these bounds (timed asynchronous system model [Cristian&Fetzer'99])
  - E.g., crash recovery: processes which recover eventually are not considered "faulty" in the classic sense

# Example

"Instead of assuming reliable channels with finite transmission delays, we assume stubborn channels with a finite average response time (if neither the sender nor the receiver crashes), and we assume that there exists some unknown physical bound on how fast an integer can be incremented. Note that there is no limit on how slow a program can be executed or how fast other statements can be executed" [Fetzer et al.'05]

# Partially Synchronous Systems

- **Partially synchronous communication**
  - There exists an upper bound on communication
    - Even if unknown
  - There is a time GST after which there exists… (eventual synchrony)
    - The system goes through “stable” and “unstable” phases
    - Usually safety ensured throughout all phases, but progress only in stable ones
- **Partially synchronous computation**
  - There exists an upper bound on differences in processing speeds
    - Even if unknown…
  - …

# Tolerating *t* Failures in Consensus

| | Synchronous | Asynch. | Partially synch. comm., synch. proc. | Partially synch. comm. and proc. | Partially synch. proc., synch. comm. |
|---|---|---|---|---|---|
| Crash-stop | *t* | - | *2t + 1* | *2t + 1* | *t* |
| Omission | *t* | - | *2t + 1* | *2t + 1* | [*2t, 2t + 1*] |
| Byzantine w. authent. | *t* | - | *3t + 1* | *3t + 1* | *2t + 1* |
| Byzantine | *3t + 1* | - | *3t + 1* (exp) [*3t + 1, 4t + 1*] | *3t + 1* (exp) [*3t + 1, 4t + 1*] | *3t + 1* (exp) [*3t + 1, 4t + 1*] |

# Shared Memory Systems

- Better fit for parallel computing (clusters, LANs, …)
- Basic interaction read/write registers
  - `read` returns value
  - `write` returns (ok/ack)
  - Operation occurs somewhere ``in between''
  - Variations in semantics/guarantees
- Usually stronger assumptions (weaker failure models)
  - Requires failure detector to be implemented in asynchronous distributed system

# References


- *Simulating Reliable Links with Unreliable Links in the Presence of Process Crashes*. A. Basu, B. Charron-Bost, S. Toueg. WDAG'96, October 1996.
- *Fundamentals of Fault-tolerant Distributed Computing in Asynchronous Environments*. Felix C. Gartner, ACM Computing Surveys 31(1):1—26, 1999.
- *Consensus in the Presence of Partial Synchrony. C. Dwork, N. Lynch, L. Stockmeyer*, JACM 35(2): 288--323, 1988.

- *On the Possibility of Consensus in Asynchronous Systems with Finite Average Response Times*. C. Fetzer, U. Schmid, M. Susskraut, ICDCS'05 271-280, 2005.
- The Timed Asynchronous Distributed System Model. F. Cristian and C. Fetzer, TPDS, 10(6): 642-657 (1999)
- *Formal Foundation for Specification and Verification*. L. Lamport and F. Schneider, Advanced Course: Distributed Systems 203-285, 1984.

# Next Time

- *Time, Clocks, and The Ordering of Events in Distributed Systems*. L. Lamport, CACM 27(7): 558-565, 1978.
- *Virtual Time and Global States of Distributed Systems*, F. Mattern, WDPA'89.