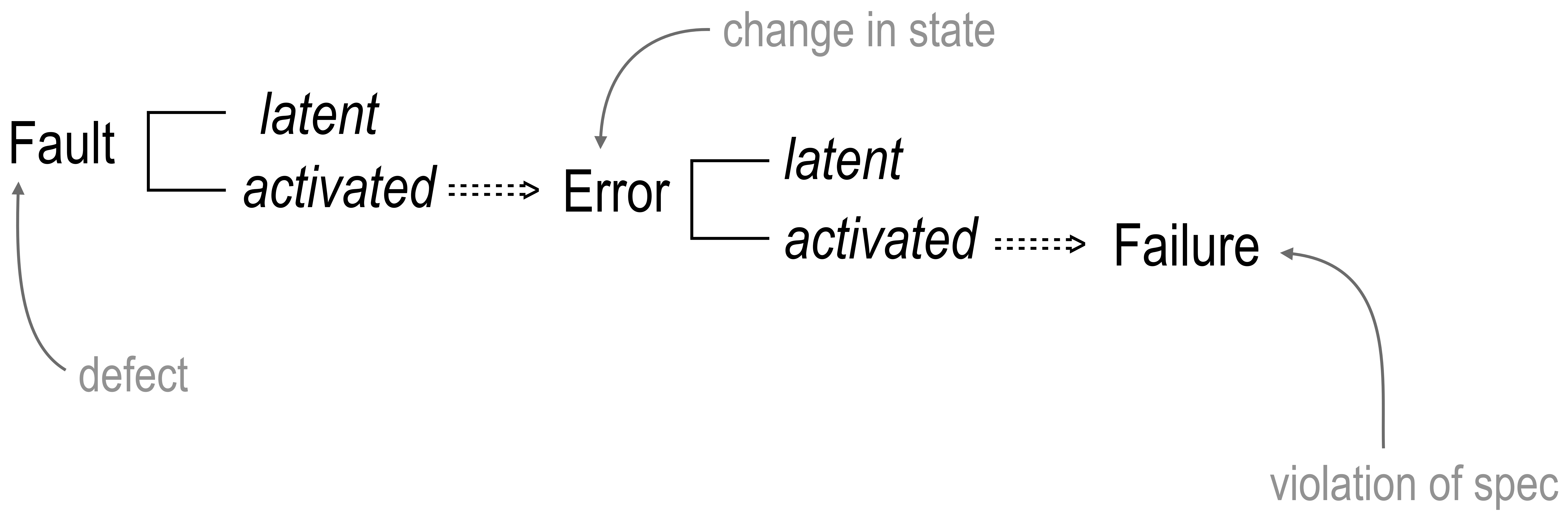
Dependability

Prof. George Candea
School of Computer & Communication Sciences

How to achieve dependability?

- Use modularity ...
- ... and REDUNDANCY for ...
  - *fault tolerance*
  - *high reliability*
  - *high availability*

Fault tolerance



Types of software faults/defects/bugs

- Bohrbug
- Heisenbug
- Schrödingbug
- Mandelbug

Types of software faults/defects/bugs

- Bohrbug
  - *clear + easy to reproduce => easy to fix*
- Heisenbug
  - *disappears when you attach with debugger*
- Schrödingbug
  - *starts causing failure once you realize it should*
- Mandelbug
  - *complex, obscure, chaotic, seemingly non-deterministic*

Using redundancy to tolerate faults

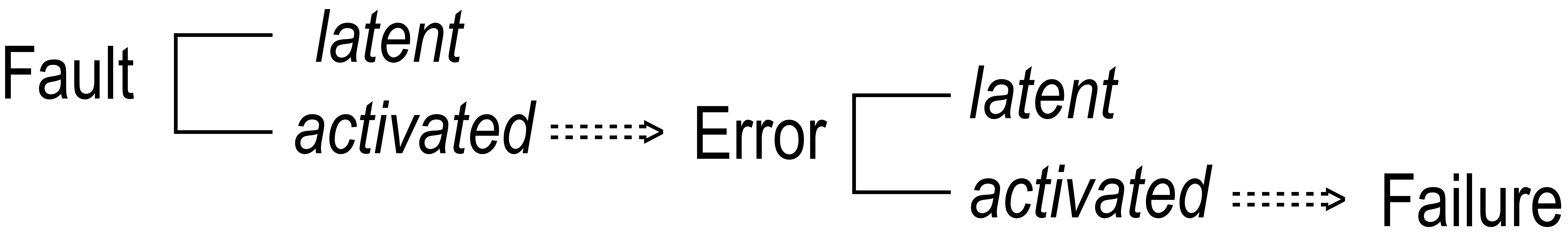
- "tolerate" faults = cope with errors or the resulting failures
  - *the actual goal is to tolerate the consequences of faults*
- Using redundancy to cope with errors
  - *error-correction codes*
  - *redundant copies/replicas (=coarse-grained ECC)*
  - *...*
- Using redundancy to cope with failures
  - *server/service failover*
  - *Internet routing*
  - *…*


https://www.submarinecablemap.com
SPOF
Arctic Ocean
Greenland Sea
Baffin Bay
Barents Sea
Kara Sea
East Siberian Sea
Norwegian Sea
Northwestern Passages
Greenland
Iceland
Sweden
Finland
Norway
Russia
Canada
Hudson Bay
Labrador Sea
Poland
Belarus
Germany
Ukraine
Kazakhstan
Mongolia
Bering Sea
Sea of Okhotsk
France
Romania
Uzbekistan
Kyrgyzstan
Spain
Italy
Turkey
Turkmenistan
United States
North Atlantic Ocean
Sea of Japan
South Korea
China
Syria
Iraq
Iran
Afghanistan
Pakistan
Tunisia
Morocco
Algeria
Libya
Egypt
Saudi Arabia
Nepal
India
Mexico
Gulf of Mexico
Western Sahara
Mauritania
Mali
Niger
Chad
Sudan
Oman
Yemen
Myanmar (Burma)
Thailand
Philippine Sea
Arabian Sea
Cuba
Burkina Faso
Guinea
Nigeria
Ethiopia
South Sudan
Venezuela
Colombia
Suriname
Somalia
Kenya
DRC
Tanzania
Brazil
Peru
Angola
Zambia
Bolivia
Namibia
Zimbabwe
Madagascar
Botswana
Paraguay
Indian Ocean
Australia
Coral Sea
South Pacific Ocean
South Atlantic Ocean
Chile
Uruguay
Argentina
South Africa
Great Australian Bight
Tasman Sea

Fault model

- Specification of what could go wrong and what cannot go wrong
  - *Used to predict consequences of failures*
  - *Should also specify what can / cannot happen during recovery*
  - *Remember the single points of failure (SPOFs)*
- Example: N-version programming
  - *use redundancy to tolerate software faults*

Recap: Fault tolerance



- Types of software defects (Bohrbug, Heisenbug, …)
- Using redundancy for tolerating errors and failures
- Fault model

Dependable = Safety-critical ???

- Safety critical = system whose failure may result in "bad" outcomes
  - *SCADA, aviation, space, automotive, healthcare, …*
- Fail-safe = failure does not have "bad" consequences
  - *safety-critical* $\not\Rightarrow$ *fail-safe*

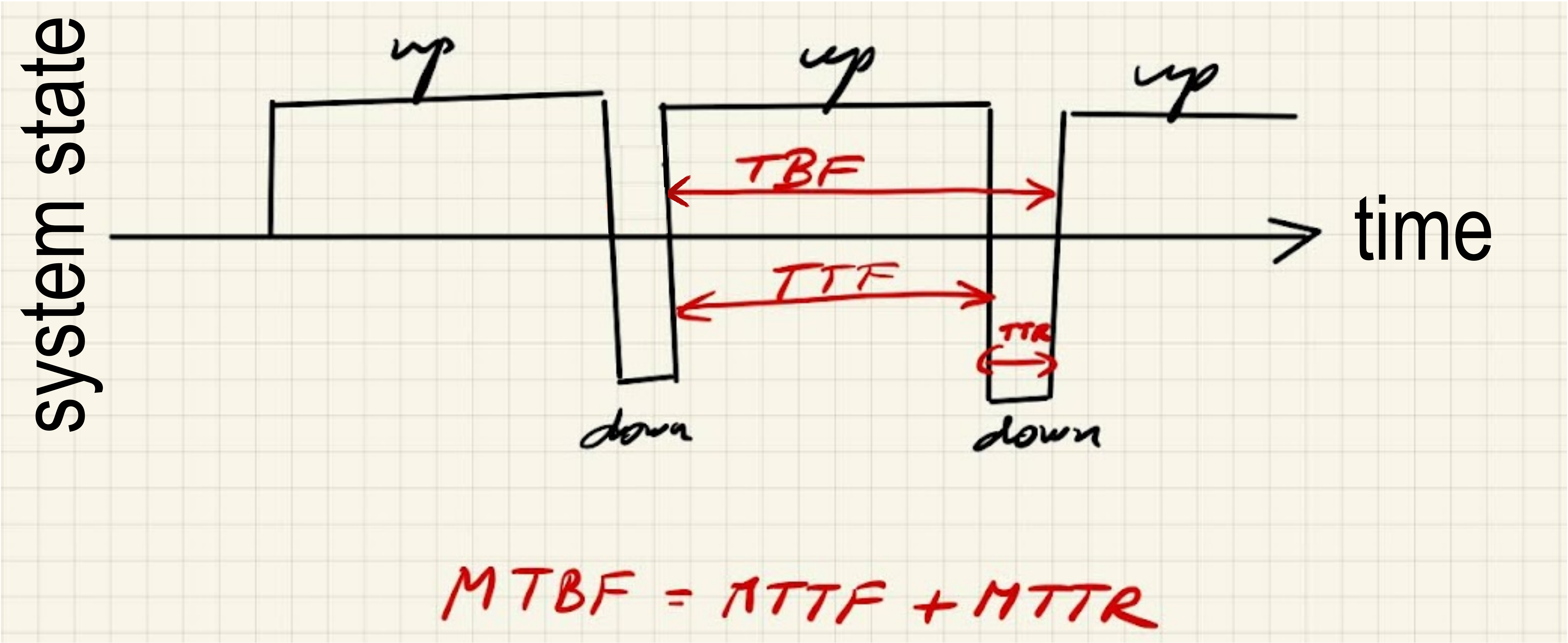
A dependable system …

- Availability = readiness for correct service
- Reliability = continuity of correct service
- Safety = absence of catastrophic consequences
- Confidentiality = absence of unauthorized disclosure of information
- Integrity = absence of improper system state alterations
- Maintainability = ability to undergo repairs and modifications

Reliability

- Reliability = probability of continuous operation
  - *continuous operation = (correctly) producing outputs in response to inputs*

$R_m(t)$ = P(module *m* operates correctly at time *t* |
m was operating correctly at *t=0*)

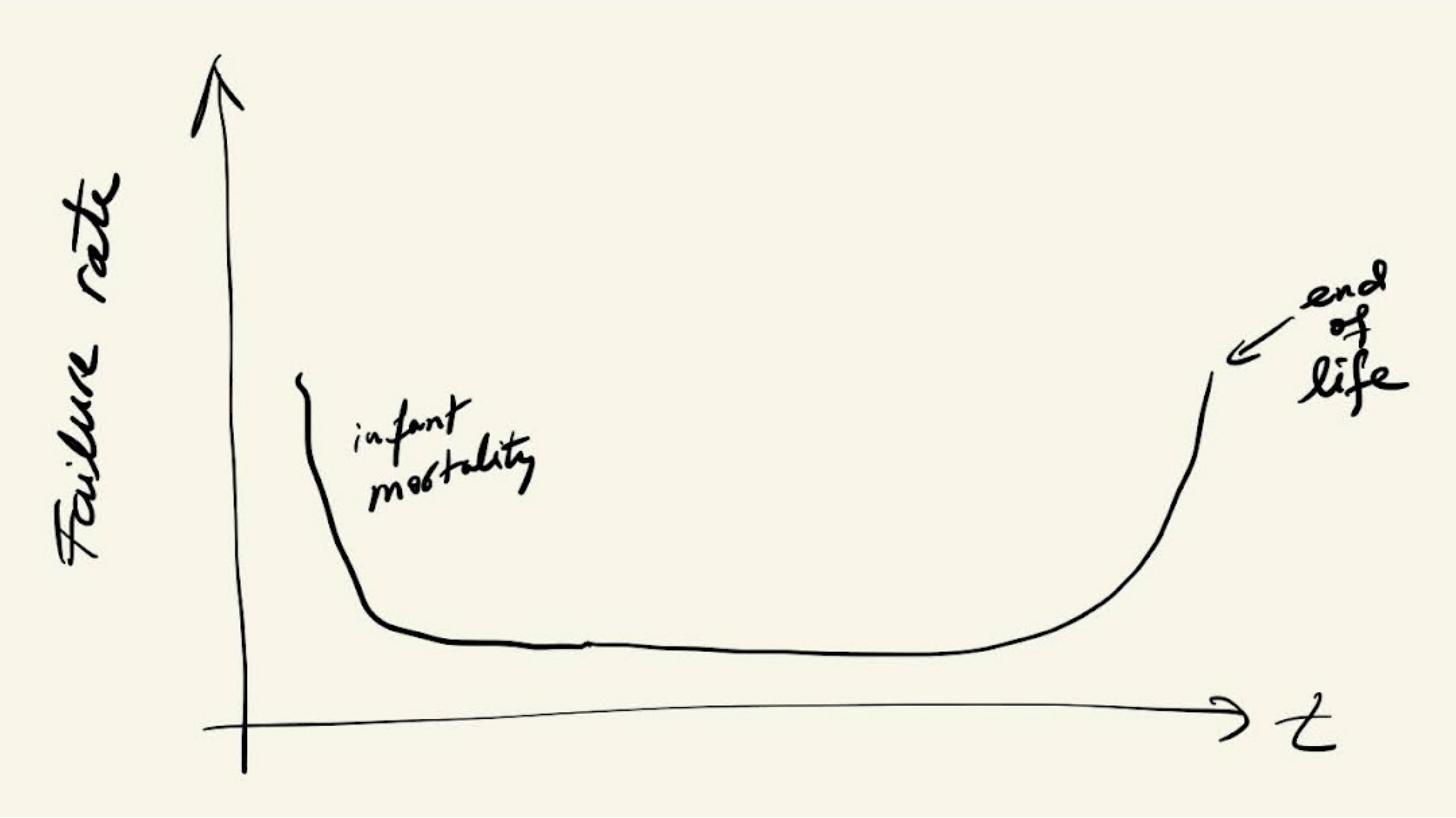


Measuring reliability

- In general MTBF or MTTF (MTBF = MTTF + MTTR)
  - *Specifics: Example from SSD spec sheet: P/E cycles, TBW, GB/day, DWPD, MTBF ...*
- Example: Samsung SSD 850 Pro SATA
  - *Warranty period = 10 years*
  - *TBW=150 => over warranty period can read/write 40 GB each day*
  - *MTBF = 2M hours (228 years)*
    - assume operation of 8 hrs/day
    - 1K SSDs => you'd experience 1 failure every ~250 days (2M / 8 / 1000)

Is systems failure ergodic ?

- Ergodicity => statistical properties of the entire process can be deduced from a single, sufficiently long, random sample of the process
  - *A system has memory => conditional failure rate of a component is not independent of how long the component has been operating*



Recap: Reliability

- Dependability = Reliability + Availability + Safety + …
- Safety-critical vs. reliable
- MTBF = MTTF + MTTR
- Failure is rarely an ergodic process

Availability

- Availability = probability of producing (correct) outputs in response to inputs

Table 1 - Levels of Availability			
Level of Availability	Percent of Uptime	Downtime per Year	Downtime per Day
1 Nine	90%	36.5 days	2.4 hrs.
2 Nines	99%	3.65 days	14 min.
3 Nines	99.9%	8.76 hrs.	86 sec.
4 Nines	99.99%	52.6 min.	8.6 sec.
5 Nines	99.999%	5.25 min.	.86 sec.
6 Nines	99.9999%	31.5 sec.	8.6 msec

Availability vs. Reliability

- Continuity of service does not matter (unlike reliability)
  - *In theory: uptime is too strict a measure of availability*
  - *In practice: what's the difference?*
- Examples of …
  - *Highly available systems with poor reliability (and how is redundancy used)*

    ...
  - *Highly reliable systems with poor availability (and how is redundancy used)*

    ...
  - *Uptime => availability but Availability ⇏ uptime*

Increasing system availability

$$\text{Avail} = \frac{MTTF}{MTBF}$$

$$\text{Unavail} = 1 - \text{Avail} = \frac{MTTR}{MTBF}$$

$$MTBF = MTTF + MTTR \cong MTTF$$

$$\boxed{\text{Unavail} = \frac{MTTR}{MTTF}}$$

- Two levers to increase availability: MTTF and MTTR
  - *i.e., increase reliability o reduce recovery time*

Intermission

Failure modes

- To increase availability or reliability, must understand failure modes
- Def: When a system fails, how does that failure appear at the interface of a component?
  - *Not the same as fault models !*

Failure mode 1: Fail-stop

- a.k.a. "crash failure" mode
- Def: halt in response to any internal error that threatens to turn into a failure, before the failure becomes visible
  - => *never expose arbitrary behavior*
- Any system can be made fail-stop with triple-modular redundancy (TMR)
  - *Strict fault model*
  - *2f + 1 independent modules to tolerate f failures*
  - *Achilles heel: voter*

Failure mode 2: Fail-fast

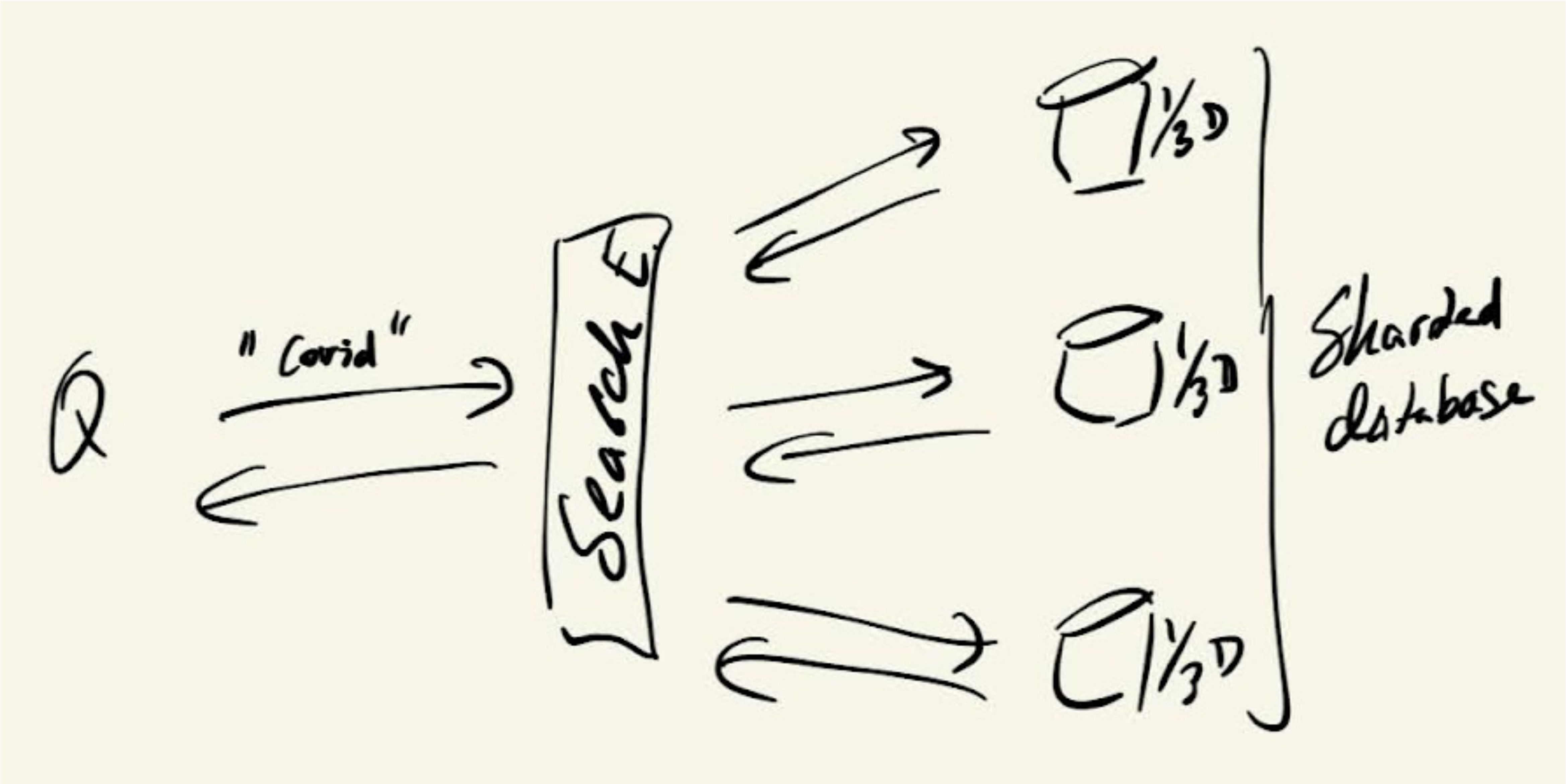
- Def: immediately report at interface any situation that could lead to failure
  - *Can stop immediately after detection or delay (if expect recovery)*
  - *Must stop before failure manifests externally*
- Requires frequent checks of state invariants
- Get auditability of error propagation

Failure mode 3: Fail-safe

- Def: the component remains safe in the face of failure (but possibly degraded functionality or performance)
- "Safety" is context-dependent
- "Controlled" failure

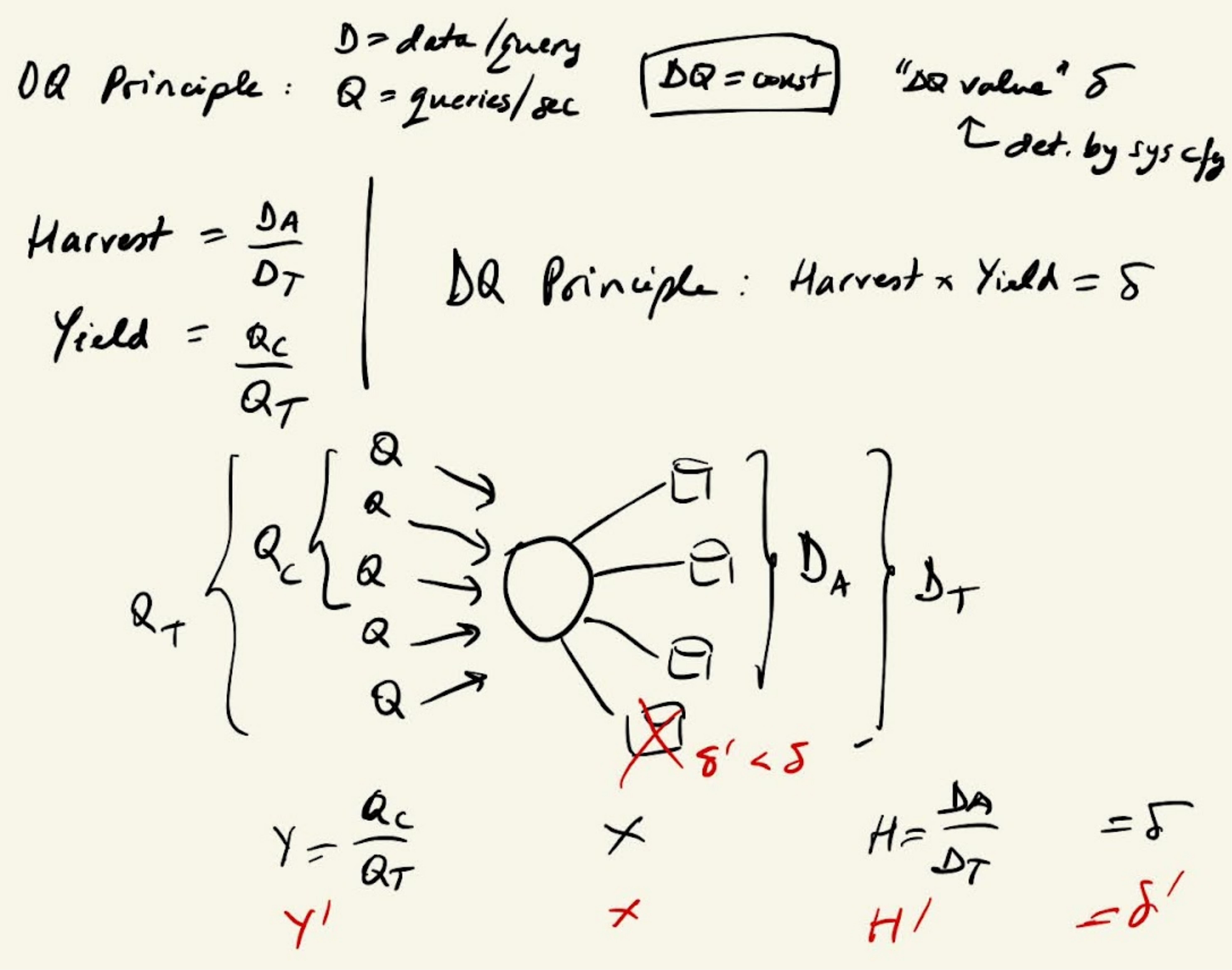
Failure mode 4: Fail-soft

- Def: internal failures lead to graceful degradation of functionality instead of outright failure



- Example: search engine
  - *system has redundancy at every level*
    - what is the fault model?
- Intuition
  - *Functionality is always bottlenecked by I/O bandwidth of disks => data movement*
    - Thus not the network, not synchronization, …
    - => Functionality tied to how much data can be moved per unit of time
  - *Harvest vs. yield*

Failure mode 4: Fail-soft: DQ Principle



Recap: Increasing availability

- Failure modes
  - *Fail-stop, fail-fast, fail-safe, fail-soft*
  - *harvest/yield, DQ principle*
- Availability equations
  - *how can we reduce unavailability by 10x?*
- Example: Internet search engine
  - *how to recover 10x faster?*

Components of recovery time

- $T_{recover} = T_{detect} + T_{diagnose} + T_{repair}$
- How to reduce T_{detect} ?
  - Automation
  - Prediction/anticipation
  - Trade-offs between FN and FPs
- How to reduce $T_{diagnose}$?
  - Lots of instrumentation, ML, ...
  - Also a function of what recovery mechanism have available
    - E.g., if only 1 way to recover, diagnosis takes zero time
- How to reduce T_{repair}?
  - Mostly app-specific
  - Reboot is universal

Exercise: Reboot-based recovery

- Design system (components) that recover(s) solely via (micro)rebooting
  - *Microreboot = surgical reboot of one or more components without affecting the rest*
- Five design principles
  - *Modularization*
  - *State segregation*
  - *Functional decoupling*
  - *Retryable interactions*
  - *Leased resources*
- Design encountered in, e.g., microservices

Exercise: Reboot-based recovery: Strong modularization

- Components with individual loci of control
  - *Well defined interfaces*
  - *Small in terms of program logic and startup time*
- $T_{reboot} = T_{restart} + T_{initialization}$

Exercise: Reboot-based recovery: State segregation

- Goal: prevent microreboot from inducing corruption or state inconsistency
  - *apply modularization idea to all state*
- Keep all important state in dedicated state stores
  - *stores located outside the application ...*
  - *... behind strongly-enforced high-level APIs (e.g., DBs, KV stores)*
- Separate data recovery from app recovery => do each one better
- Segment the state by lifetime

Exercise: Reboot-based recovery: Functional decoupling

- Goal
  - *reduced disruption of system during restart*
  - *easy reintegration of component after reinit*
- No direct references (e.g., no pointers) across component boundaries
  - *Cross-component references stores outside component*
    - Naming indirection through runtime
    - Marshall names into state store

Exercise: Reboot-based recovery: Retryable interactions

- Goal: make reintegration of component seamless by recovering in-flight requests transparently
- Interact via timed RPCs – if no response, caller can gracefully recover
  - *timeouts help turn non-Byzantine failures into fail-stop events*
  - *RPC to a microrebooting module throws RetryAfter(t) exception*
- Action depends on whether RPC is idempotent or not

Exercise: Reboot-based recovery: Leased resources

- Goal: avoid resource leakage without fancy resource tracking
- Lease = timed ownership
  - *File descriptors, memory, …*
  - *Persistent long-term state*
  - *CPU execution time*
- Requests carry TTL => automatically purged when TTL runs out

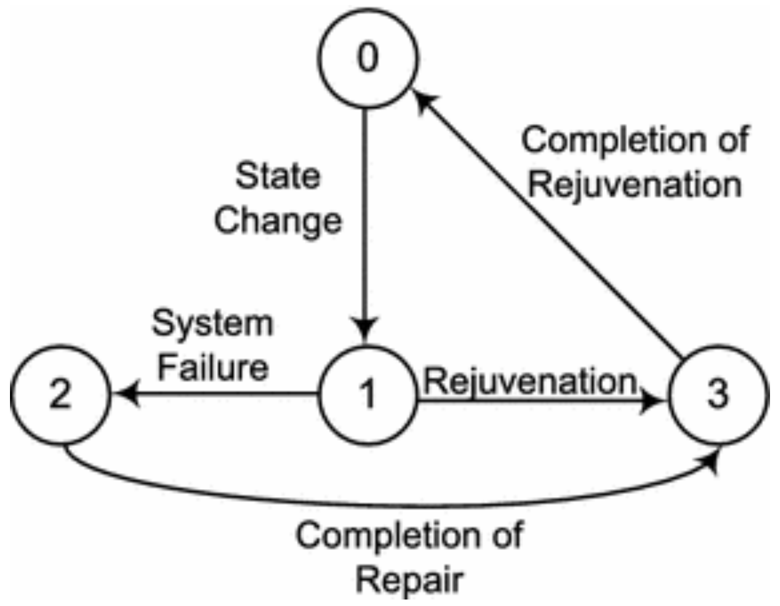
Recap: Reboot-based recovery

- Insight: Reboot as a universal "hammer" in curing failures
  - Can we systematically employ rebooting to cure failures?
  - While everyone is trying to increase MTTF, why not try to reduce MTTR?
- Five design principles
  - Modularization, State segregation, Functional decoupling, Retryable interactions, Leased resources
- Well suited for workloads consisting of fine-grained requests
  - Used in Internet services/microservices, analytics engine, satellite ground station
- Recursive microrebooting
  - Let MTTF and MTTR indicate boundaries of restart

`Google "crash-only software" for more info...`

Software rejuvenation

- Goal: clean up state to prevent accumulation of errors
  - *Insight: Reboot as a prophylactic*
  - *Does nothing about defects, but reduces probability of turning errors into failures*
- Turns unplanned downtime into planned downtime
  - *Dynamic version of "preventive maintenance"*
  - *Release leaked resources, wipe out data corruption, ...*



- Microrejuvenation: turn unplanned downtime into planned partial downtime (or none at all)

Recap

- $T_{recover} = T_{detect} + T_{diagnose} + T_{repair}$
- With reboot-based recovery...
  - $T_{recover} = T_{detect} + T_{reboot}$
- If recovery is cheap (i.e., T_{repair} is small), can offer imperfect detection
- By reducing $T_{recover}$ we reduce MTTR => availability goes up
  - *reliability is not affected in a well designed system*