Dependability through Redundancy

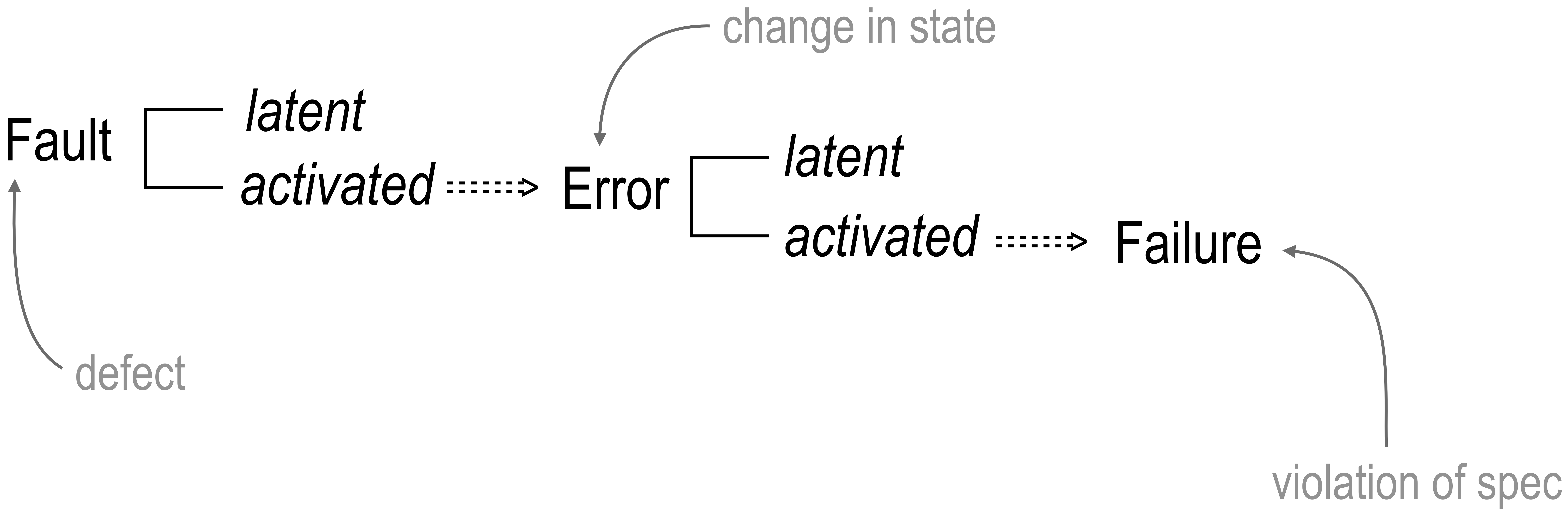
Prof. George Candea
School of Computer & Communication Sciences

How to achieve dependability?

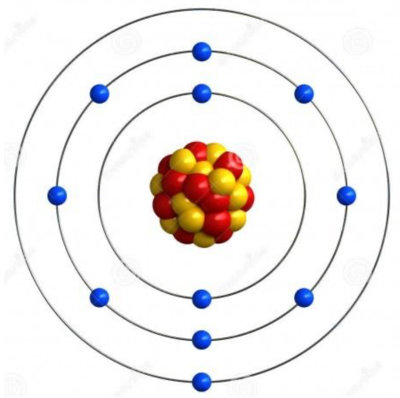
- Use modularity ...
- ... and REDUNDANCY for ...
  - *fault tolerance*
  - *high reliability*
  - *high availability*

Redundancy = duplication with the purpose of increasing dependability

Fault tolerance



Types of software faults / defects

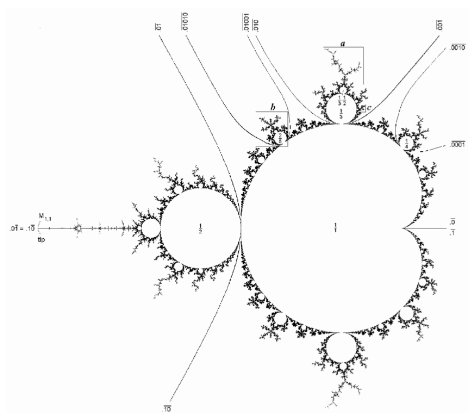


- Bohrbug
  - *clear + easy to reproduce => easy to fix*

$$\Delta\chi\Delta\rho \geq \frac{\hbar}{2}$$

- 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
  - *forward error correction*
  - *redundant copies/replicas (=coarse-grained ECC)*
  - *...*
- Using redundancy to cope with failures
  - *server/service failover*
  - *Internet routing*
  - *...*

Data/information redundancy

Geographic redundancy

Processing redundancy — Space / Time

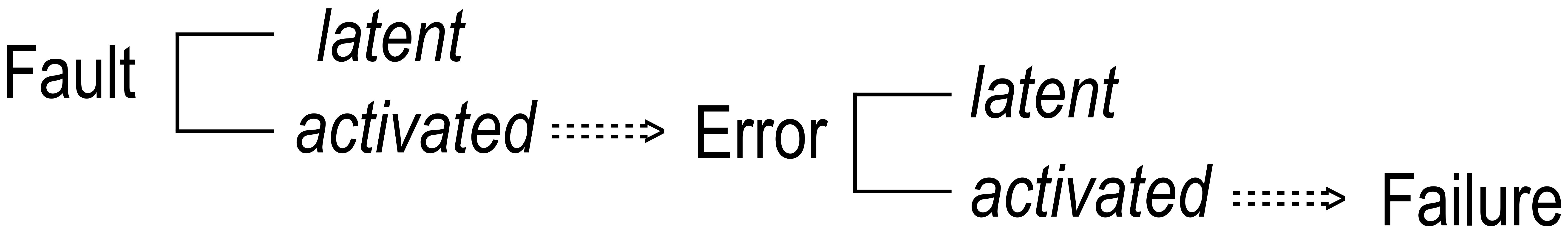
Functional redundancy


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

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



- Different types of software defects
  - *Bohrbug, Heisenbug, …*
- Redundancy helps tolerate errors and failures
  - *Data redundancy, processing redundancy, ...*
- Fault model = assumptions about what can vs. cannot go wrong

Safety-critical systems

- 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*

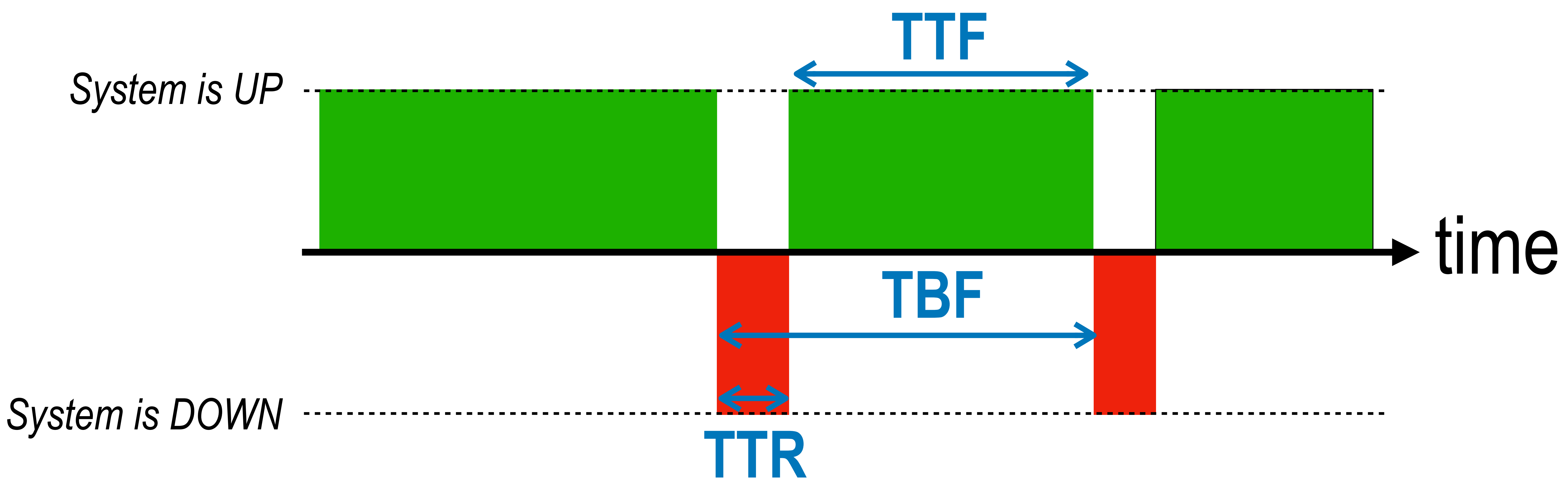
Dependable systems

- 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(t) = P(module operates correctly at time *t* | it was operating correctly at *t=0*)



MTBF = MTTF + MTTR

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*
  - *MTBF = 2M hours (228 years)*
    - assumes operation of 8 hrs/day
    - 2.5K SSDs => you'd experience 1 failure every ~100 days (2M / 8 / 2500)

Why different???

Recap: Reliability

- Dependability = Reliability + Availability + Safety + …
- Safety-critical vs. reliable
- MTBF = MTTF + MTTR

Availability

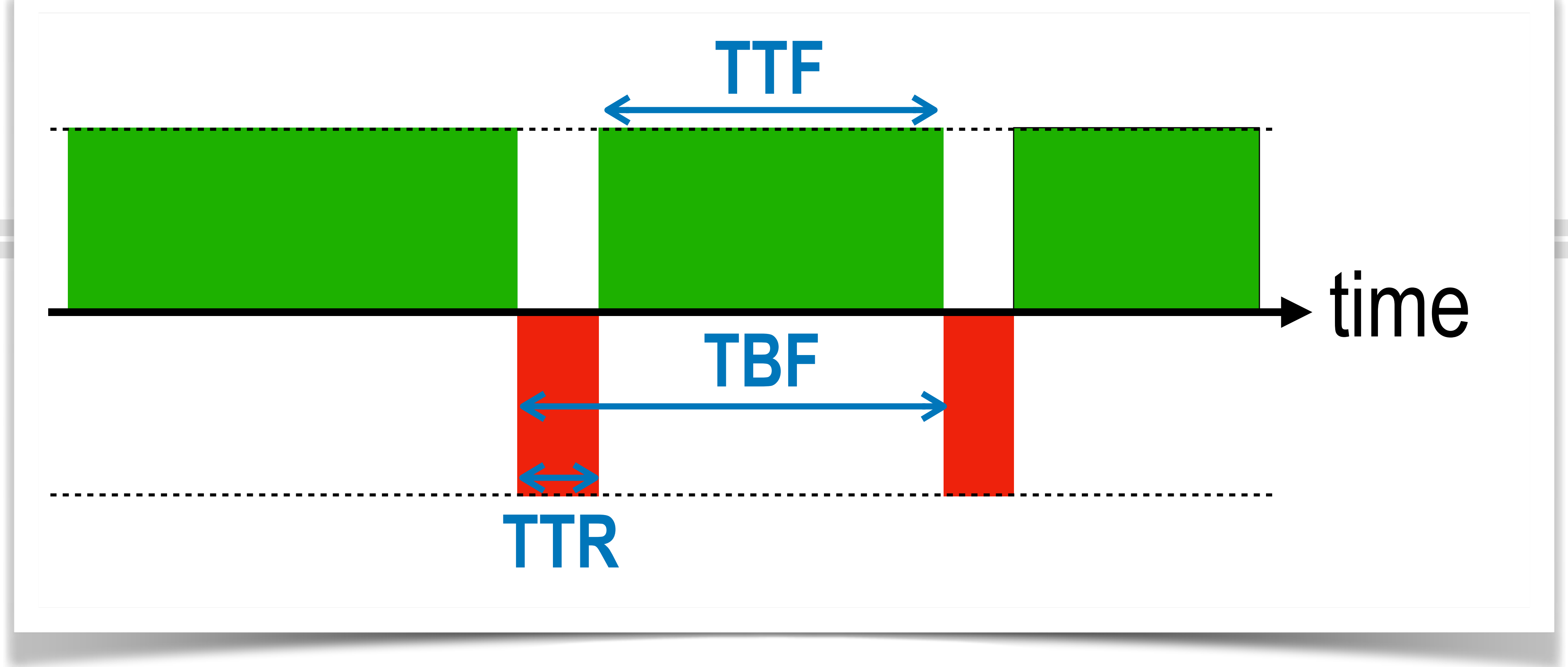
- Availability = probability of producing (correct) outputs in response to inputs

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. ↑×10	[illegible].6 sec.
5 Nines	99.999%	5.25 min.	.86 sec.
6 Nines	99.9999%	31.5 sec. ↓÷10	[illegible].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?*
- Uptime => availability but Availability ⇏ uptime
- Examples of …
  - *Highly available systems with poor reliability (and how is redundancy used) ...*
  - *Highly reliable systems with poor availability (and how is redundancy used) ...*

System availability



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

$$\text{Unavailability} = 1 - \text{Availability} = \frac{\text{MTTR}}{\text{MTBF}}$$

$$\text{MTBF} = \text{MTTF} + \text{MTTR} \cong \text{MTTF} \quad (\text{if MTTF} \gg \text{MTTR})$$

$$\text{Unavailability} \cong \frac{\text{MTTR}}{\text{MTTF}}$$

- Increase availability by
  - *increasing MTTF (higher reliability)*
  - *reducing MTTR (faster recovery)*

Failure modes

Failure modes

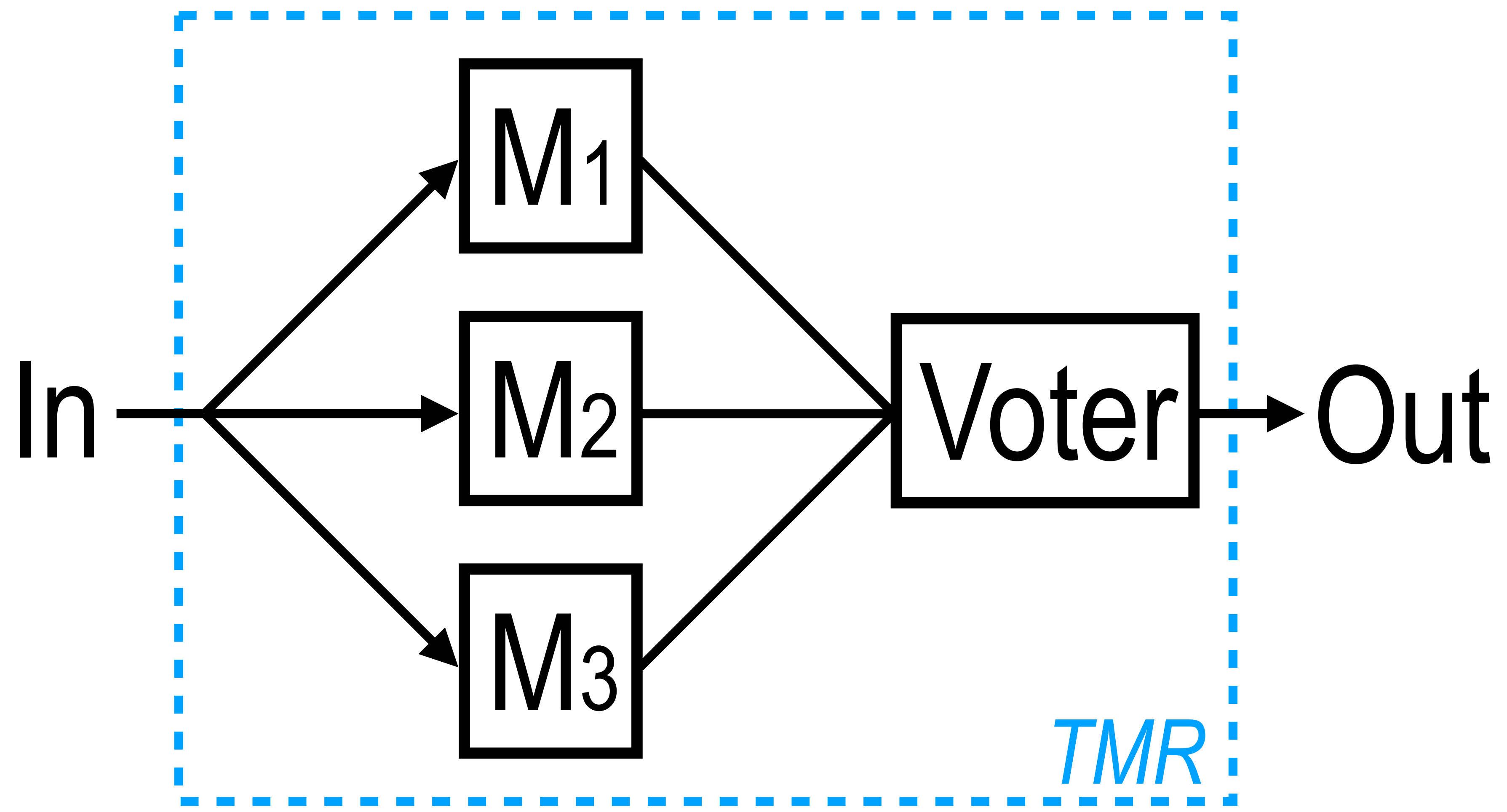
- Definition:

  When a system fails, how does that failure appear at the interface of a component?

- Four kinds
  - *fail-stop*
  - *fail-fast*
  - *fail-safe*
  - *fail-soft*

Failure mode 1: Fail-stop

- a.k.a. "crash failure" mode
- *Definition*: 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: voter is reliable*
  - *2f +1 independent modules to tolerate f failures*
  - *Achilles's heel: voter*



Failure mode 2: Fail-fast

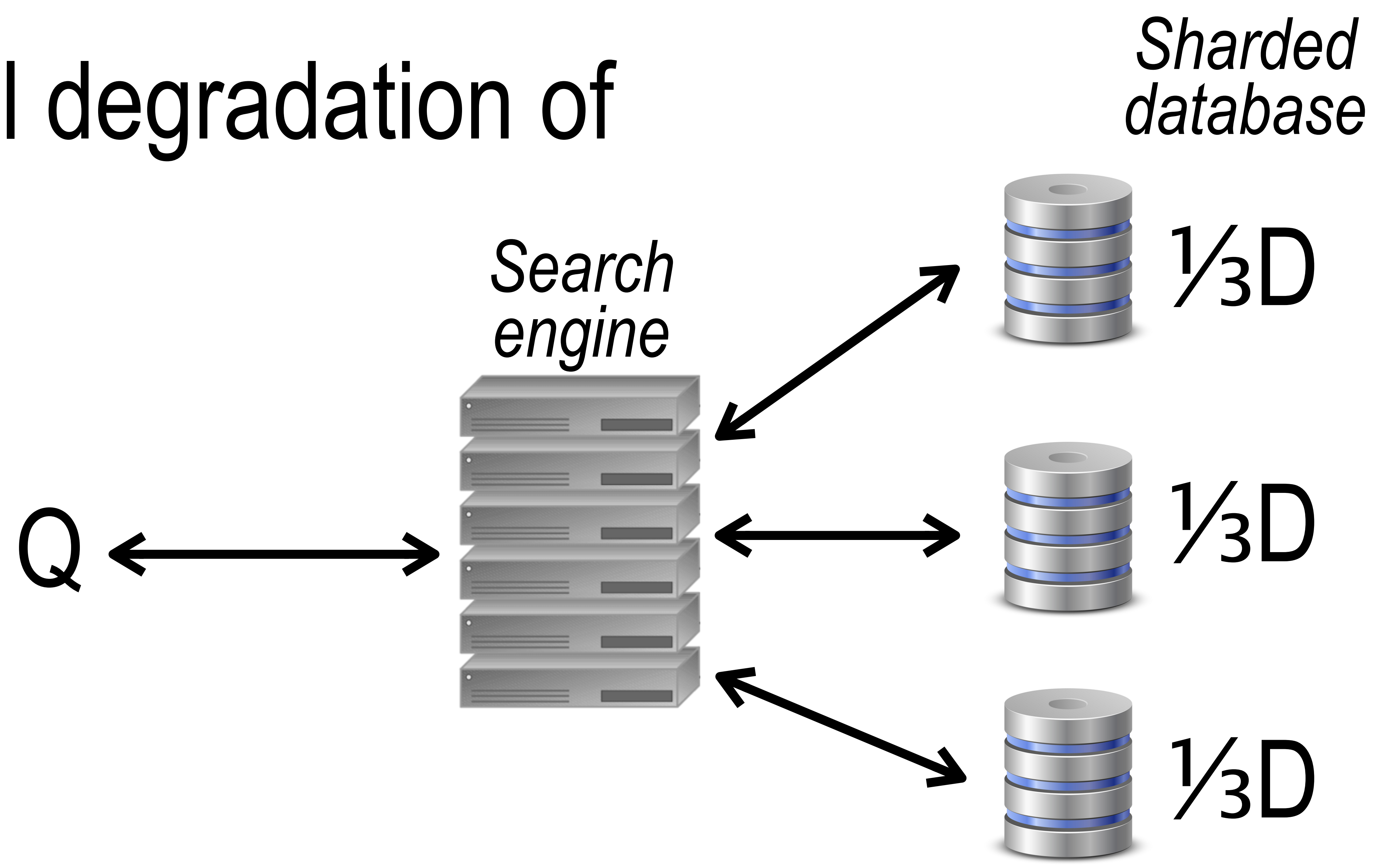
- *Definition*: immediately report at interface any situation that could lead to failure
  - *Can stop immediately after detection or delay (if expect recovery)*
  - *Must stop before failure manifests externally*
- Requires frequent checks of state invariants
- Get auditability of error propagation

Failure mode 3: Fail-safe

- *Definition*: 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

- Definition: internal failures lead to graceful degradation of functionality instead of outright failure
- Example: simple search engine
  - *system has redundancy at every level*
- Intuition
  - *Functionality is typically bottlenecked on data movement (disks, network switches)*
    - => Functionality tied to how much data can be moved per unit of time
  - ***Harvest*** *(completeness of responses) vs.* ***yield*** *(fraction of requests served)*



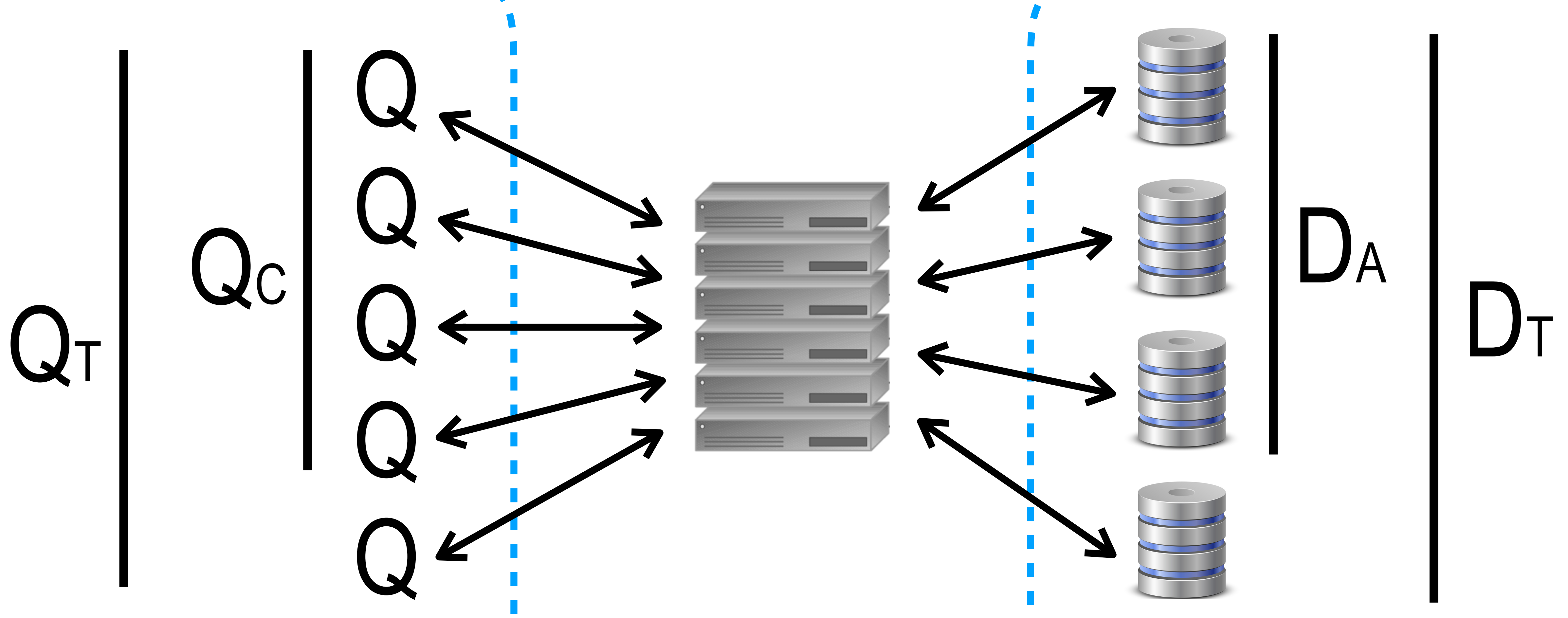
Failure mode 4: Fail-soft: DQ Principle

D = data/query
Q = queries/sec

DQ Principle: "D×Q is constant"
(DQ value ρ determined by system configuration)

Harvest $H = \frac{D_A}{D_T}$

Yield $Y = \frac{Q_C}{Q_T}$



DQ Principle: $H \times Y = \rho$ - - - - - ▸ $Y = Q_C / Q_T$ × $H = D_A / D_T$ = ρ

Recap: Failure modes

- Fail-stop (TMR)
- Fail-fast (Redundant invariant checks)
- Fail-safe
  - *OK to fail, as long as safety is not compromised*
- Fail-soft (Weaker spec)
  - *Redundant resources for top band of acceptable system behavior*
  - *Harvest/yield and the DQ principle in data-intensive parallel systems*

How to improve availability by 10× ?

How to improve availability by 10× ?

$$\text{Unavailability} \cong \frac{\text{MTTR} \;\downarrow \div 10}{\text{MTTF} \;\uparrow \times 10}$$

Components of recovery time

- $T_{recover} = T_{detect} + T_{diagnose} + T_{repair}$
- How to reduce T_{detect} ?
  - Automation
  - Prediction/anticipation
  - Trade-offs between FPs and FNs
- How to reduce $T_{diagnose}$?
  - Lots of instrumentation, ML, ...
  - Also a function of what recovery mechanism have available
- How to reduce T_{repair}?
  - Mostly app-specific
  - Reboot is universal

Detection/Prediction says...

Truth is...	Failure	No Failure
No Failure	FP	TN
Failure	TP	FN

How to improve availability by 10× ?

Reboot-based recovery

Reboot-based recovery

- Design system (components) that recover(s) solely via (micro)rebooting
  - *stop == crash   start == recover*
- Design for e.g. microservices
  - *short-running tasks, clusters of many nodes, ...*
- Crash-only components
  - *State segregation*
- Crash-only system of components
  - *Modularization + functional decoupling*
  - *Retryable interactions*
  - *Leased resources*

https://aknextphase.com/crossing-the-job-search-finish-line/better-things-are-coming/

Reboot-based recovery: State segregation

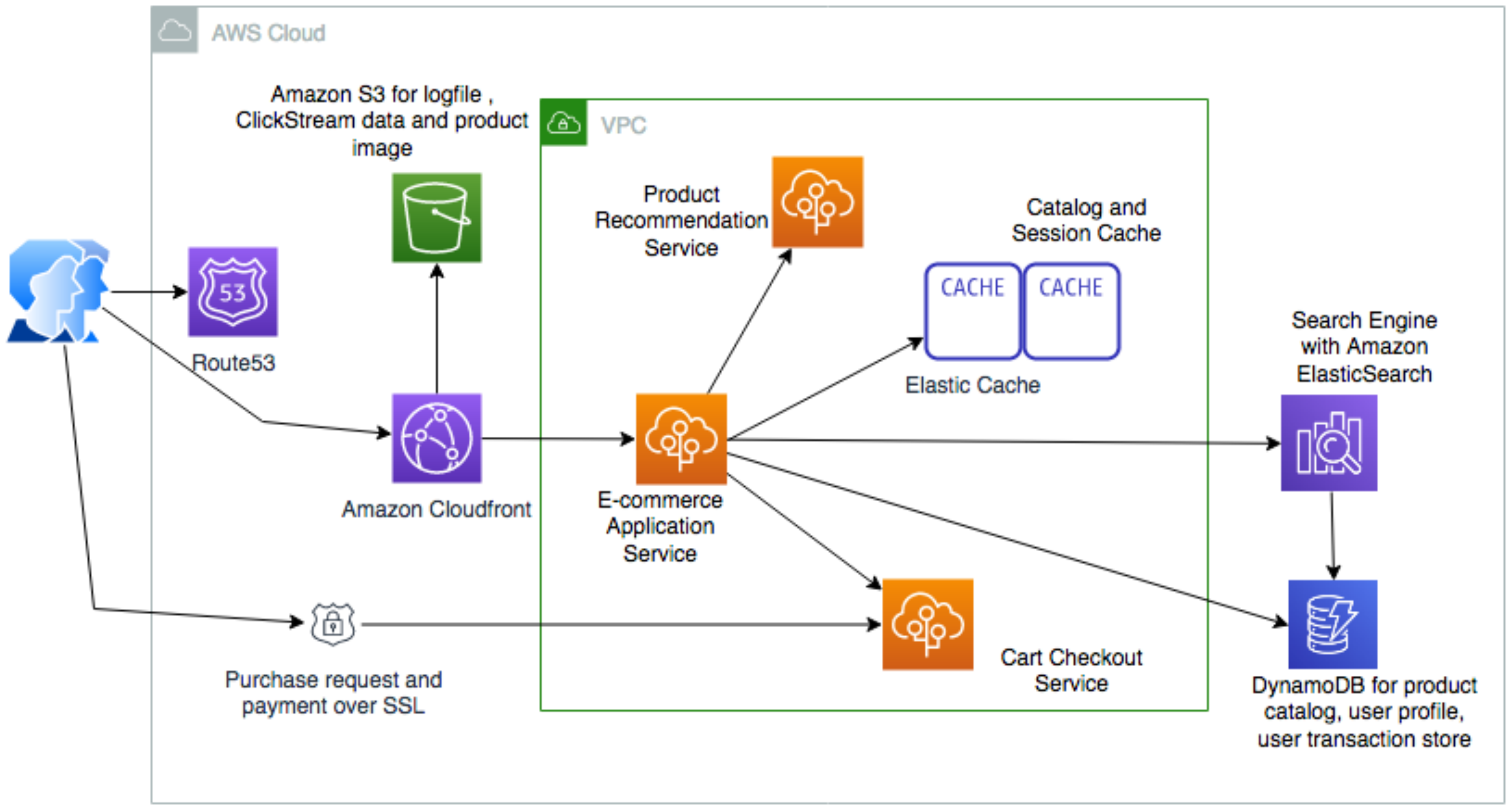
- Goal: prevent microreboot from inducing corruption or state inconsistency
  - *apply modularization idea to all state: session state vs. persistent state*
- Segment the state by lifetime
- Keep all state that should survive a reboot in dedicated state stores
  - *stores located outside the application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

State segregation
Modularization
Functional decoupling
Retryable interactions
Leased resources

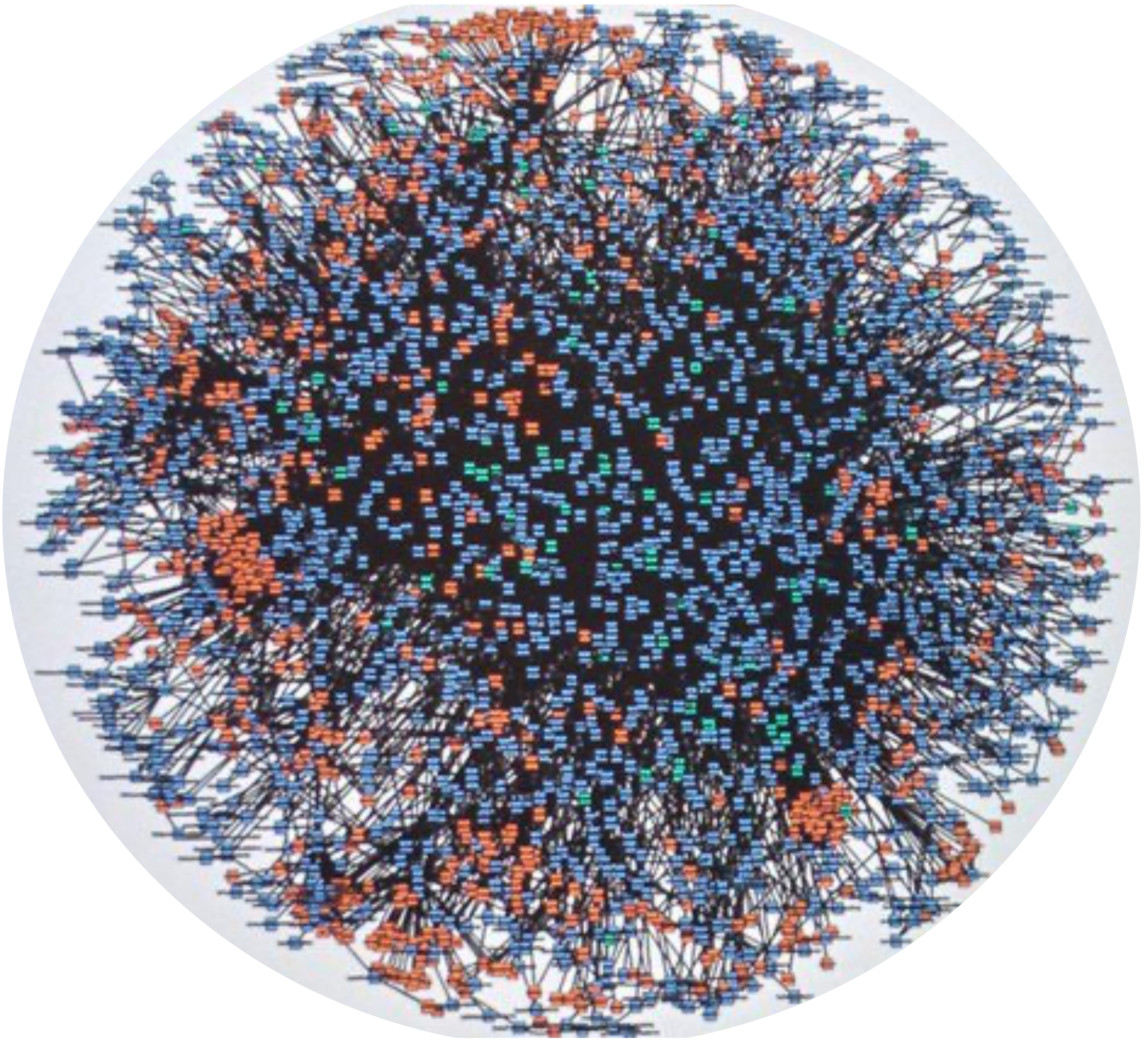
Reboot-based recovery: Strong modularization

State segregation
Modularization
Functional decoupling
Retryable interactions
Leased resources

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



https://subscription.packtpub.com/book/web-development/9781838645649/6/ch06lvl1sec28/building-an-soa-based-e-commerce-website-architecture

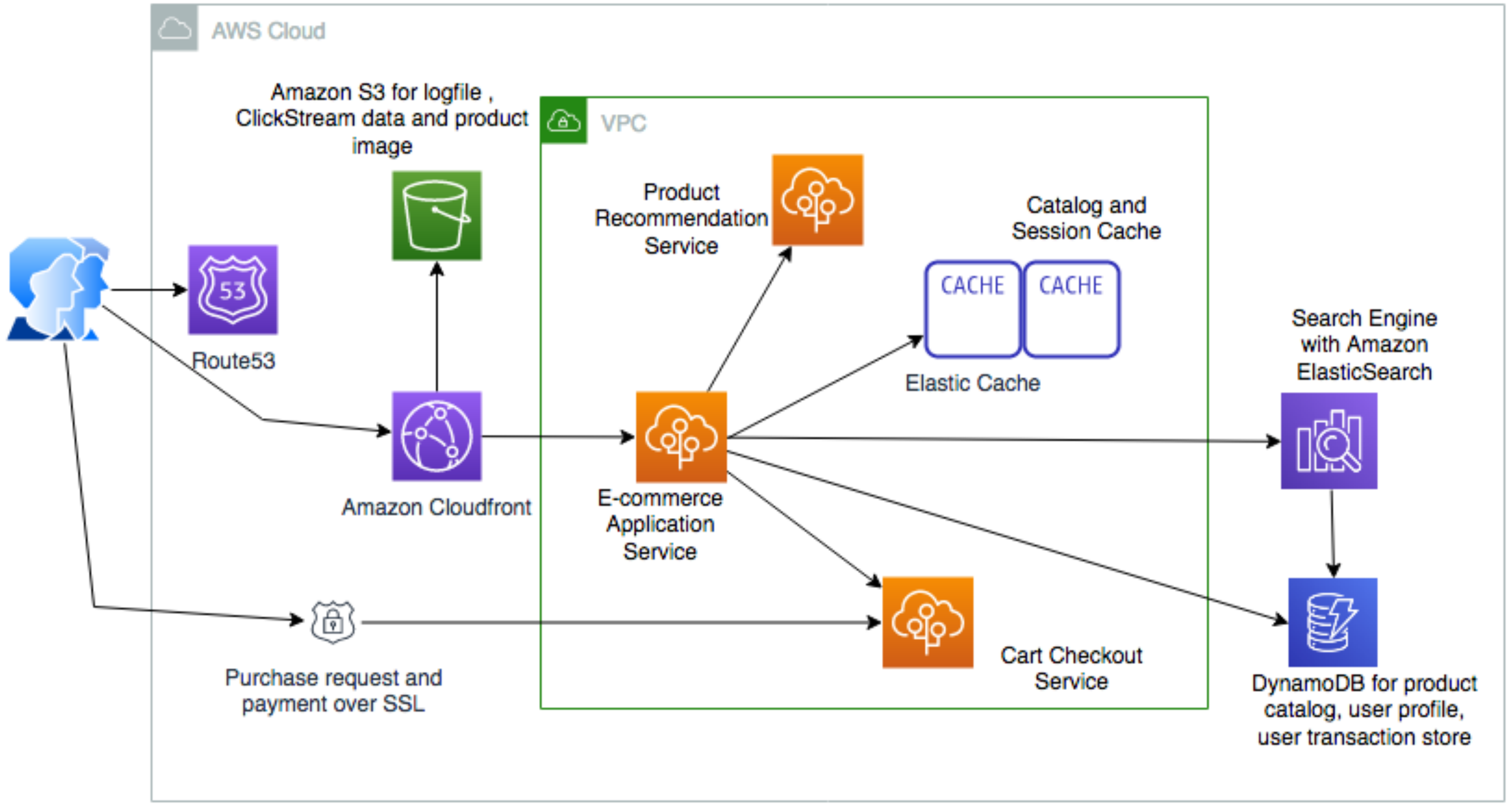


https://twitter.com/Werner/status/741673514567143424/photo/1

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
  - *Store cross-component references outside component*
    - Naming indirection through runtime
    - Marshall names into state store

State segregation
Modularization
Functional decoupling
Retryable interactions
Leased resources



https://subscription.packtpub.com/book/web-development/9781838645649/6/ch06lvl1sec28/building-an-soa-based-e-commerce-website-architecture

Reboot-based recovery: Retryable interactions

- Goal
  - *seamless reintegration of microrebooted component by recovering in-flight requests transparently*
- Interact via timed RPCs or equivalent
  - *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

State segregation
Modularization
Functional decoupling
Retryable interactions
Leased resources

Exercise: Reboot-based recovery: Leased resources

State segregation
Modularization
Functional decoupling
Retryable interactions
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

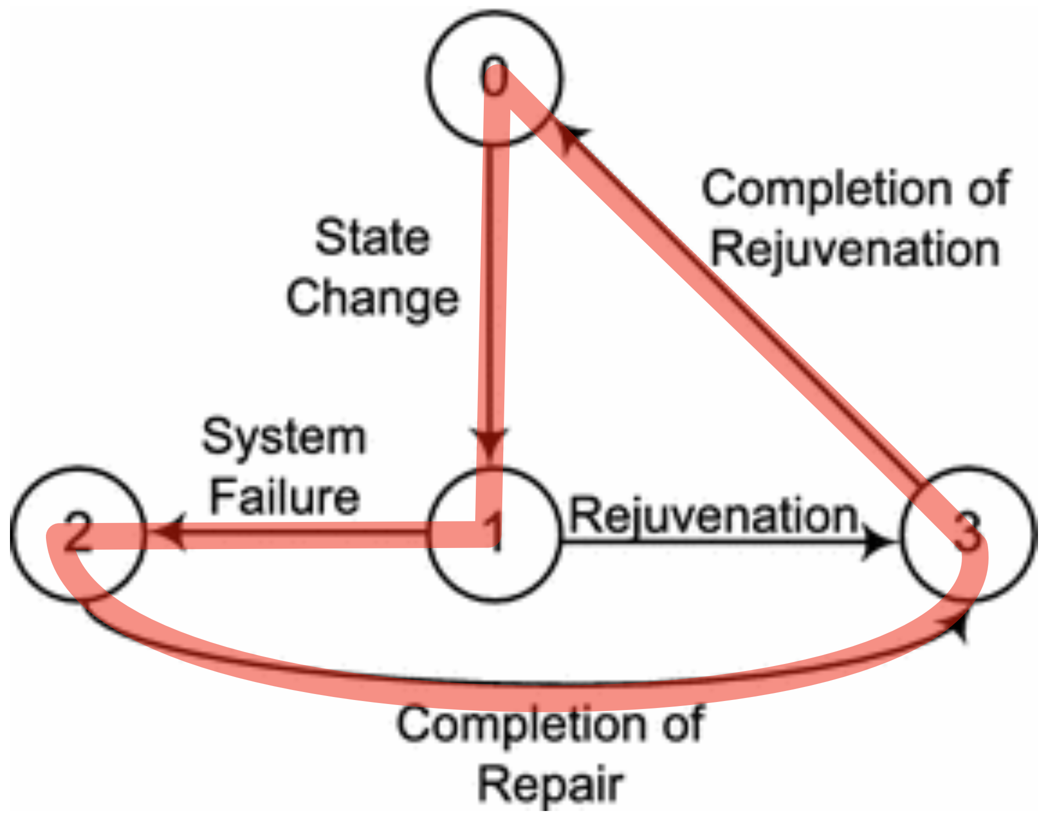
Recap

- $T_{recover} = T_{detect} + T_{diagnose} + T_{repair}$
  - *If recovery is cheap (i.e., T_{repair} is small), can tolerate FPs*
  - *Instead of trying to increase MTTF, consider reducing MTTR*
    - Availability goes up, reliability is not affected (in a well designed system)
- Reboot as a universal "hammer" for curing failures
    - Systematically employ rebooting to cure failures?
- Well suited for workloads consisting of fine-grained requests
    - Currently used in Internet services/microservices, analytics engine, satellite ground station
    - If a fine-grained microreboot doesn't make the problem go away, try coarser-grained

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

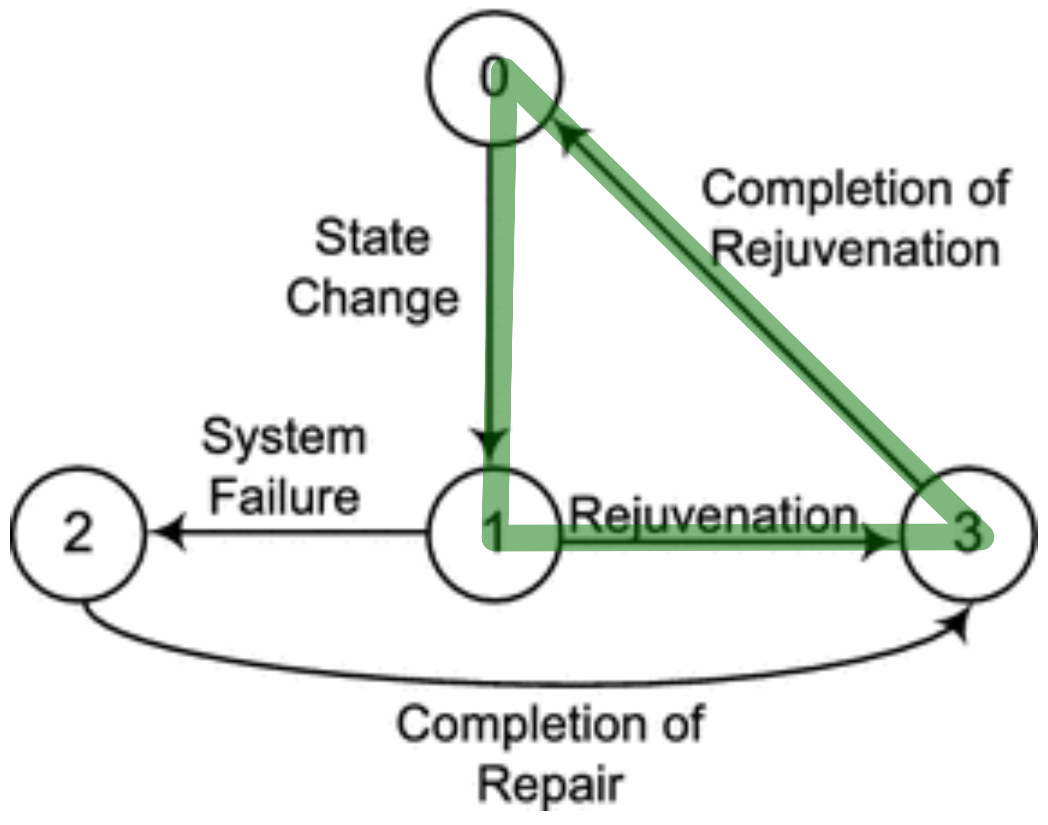
Software rejuvenation

- 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, ...*



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)*