ECE 455 – Module 4

Dependability

Spring 2023

Murray Dunne – mdunne@uwaterloo.ca

Slides Acknowledgement

Some material in these slides is based on slides from:

- Prof. Sebastian Fischmeister
- Prof. Carlos Moreno

Material in this module:

- Laprie, Jean-Claude. Dependability: Basic concepts and terminology. Dependability: Basic Concepts and Terminology. Springer, Vienna, 1992. 3-245.
- Avizienis, Algirdas, Jean-Claude Laprie, and Brian Randell. "Dependability and its threats: a taxonomy." Building the Information Society. Springer, Boston, MA, 2004. 91-120.

Other material used with citations

Faults, **Errors**, and **Failures**

- A **failure** occurs when the delivered service no longer complies with the specification
- An **error** is the system state that leads to a subsequent failure
- The cause of an error is a **fault**

Fault → Error → Failure → Fault → Error → Failure → Fault → …

Faults

- **Accidental faults** vs. **Intentional faults**
  - Accidental: mistakes, omissions, random chance
  - Intentional: malicious logic, intrusion

- **Permanent faults** vs. **Temporary faults**
  - Obvious distinction
  - Temporary breaks down into:
    - **Transient faults** originate from the physical environment
    - **Intermittent faults** originate internally

- **Latent** vs. **Active faults**
  - Latent faults are not yet detected

Faults

- **Accidental faults** vs. **Intentional faults**
  - Accidental: mistakes, omissions, random chance
  - Intentional: malicious logic, intrusion
- **Permanent faults** vs. **Temporary faults**
  - Obvious distinction
  - Temporary breaks down into:
    - **Transient faults** originate from the physical environment
    - **Intermittent faults** originate internally

The Fault Rabbit-hole

- A transistor fails to carry a charge when it should
  - Consequence of a fault at the electronic level
  - Consequence of a chemistry fault at the foundry
  - Consequence of a fault in the manufacturing process
  - Consequence of a limit in understanding semiconductor physics
  - Consequence of a limit in understanding physics
  - Consequence of a human fault in creating an imperfect standard model of particle physics

- At some point, you need to stop…

Errors

- Does a fault lead to a failure?
  - Only if the fault causes an error first
  - **Dependability** manifests in the design of the system
    - A more dependable system is better at not letting faults lead to failures by having less possible error states from those faults

Disaster of the Day – Los Angeles ARTCC (2004)

Failures

- A **value failure** occurs when delivered service deviates from specification
- A **timing failure** occurs when the timing of a service deviates from specification
- A **resource consumption failure** occurs when the system uses/doesn’t use the correct resources per specification
  - Too much power consumption

System failure types

- A **fail-stop** system stops running on failure
  - Often not feasible in real-world settings
- A **fail-silent** system does not do anything guaranteed on failure
- A **fail-soft** system gracefully degrades on failure
- A **fail-operational** system continues operating within specified given assumptions
  - Example: aircraft

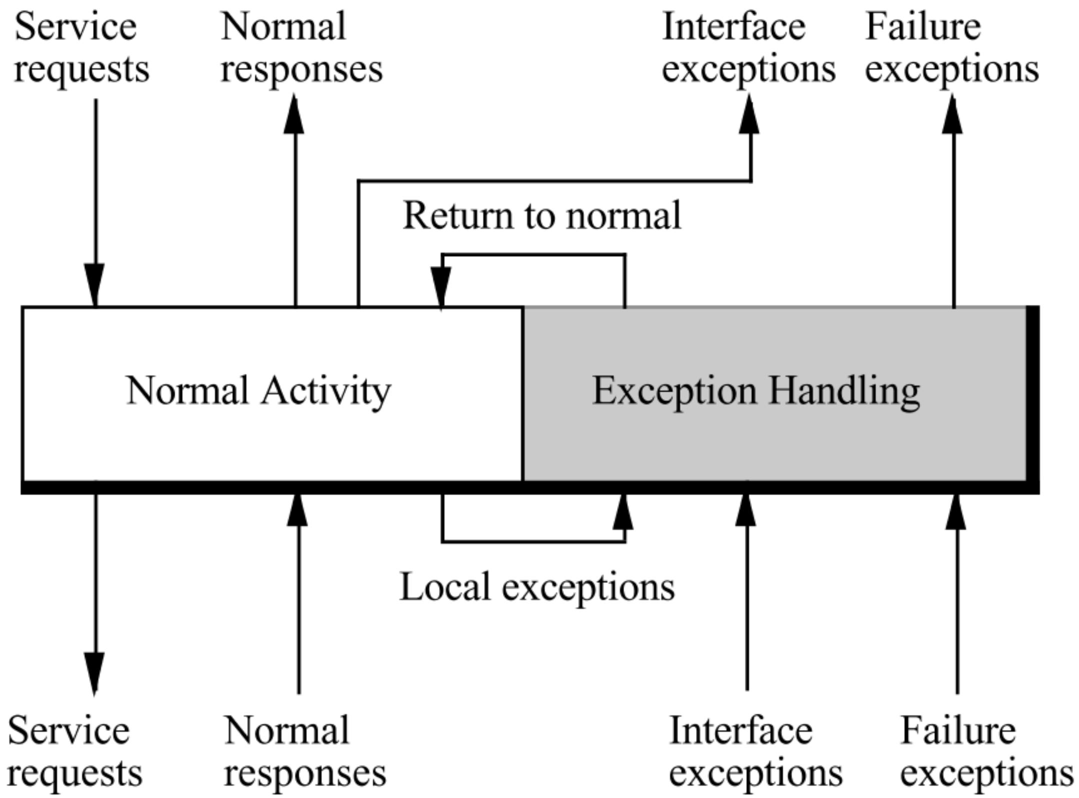
Fault Tolerance in Embedded Software

- How do we increase software dependability by not letting faults lead to failures?
  - Depends on where the fault comes from
    - If we get a failure from a lower tier component, that is an **interface exception**
    - If something goes wrong in our computation, that is a **local exception**
    - If we have an unrecoverable fault/error that must lead to a failure, that is a **failure exception**
  - Design our software component to handle interface and local exceptions

Recovery

- **Backward recovery** is when the system returns to a checkpoint to retry computation
  - Time permitting
- **Forward recovery** is when the system converts the error state into some acceptable default and continues
  - Often due to time constraints or environment state

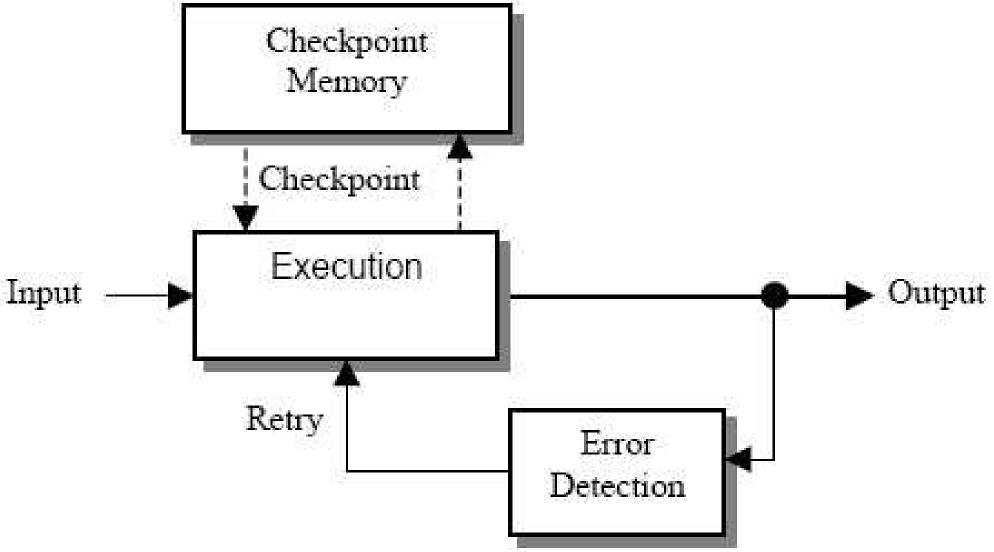
The Ideal Fault-Tolerant Component



- This is not a real fault tolerance approach
  - This is what some ideal fault tolerant component would look like in a perfect world

Approach 1: **Basic Checkpointing**

- Record a **checkpoint** before execution begins
  - A backup of algorithm/program state
- **Error detector** must be able to detect error states from the results of component execution
  - Also called an **acceptance test**
  - This might not be knowable
- If the error detector detects an error
  - Reload state from the checkpoint and try again
    - Might continue to fail

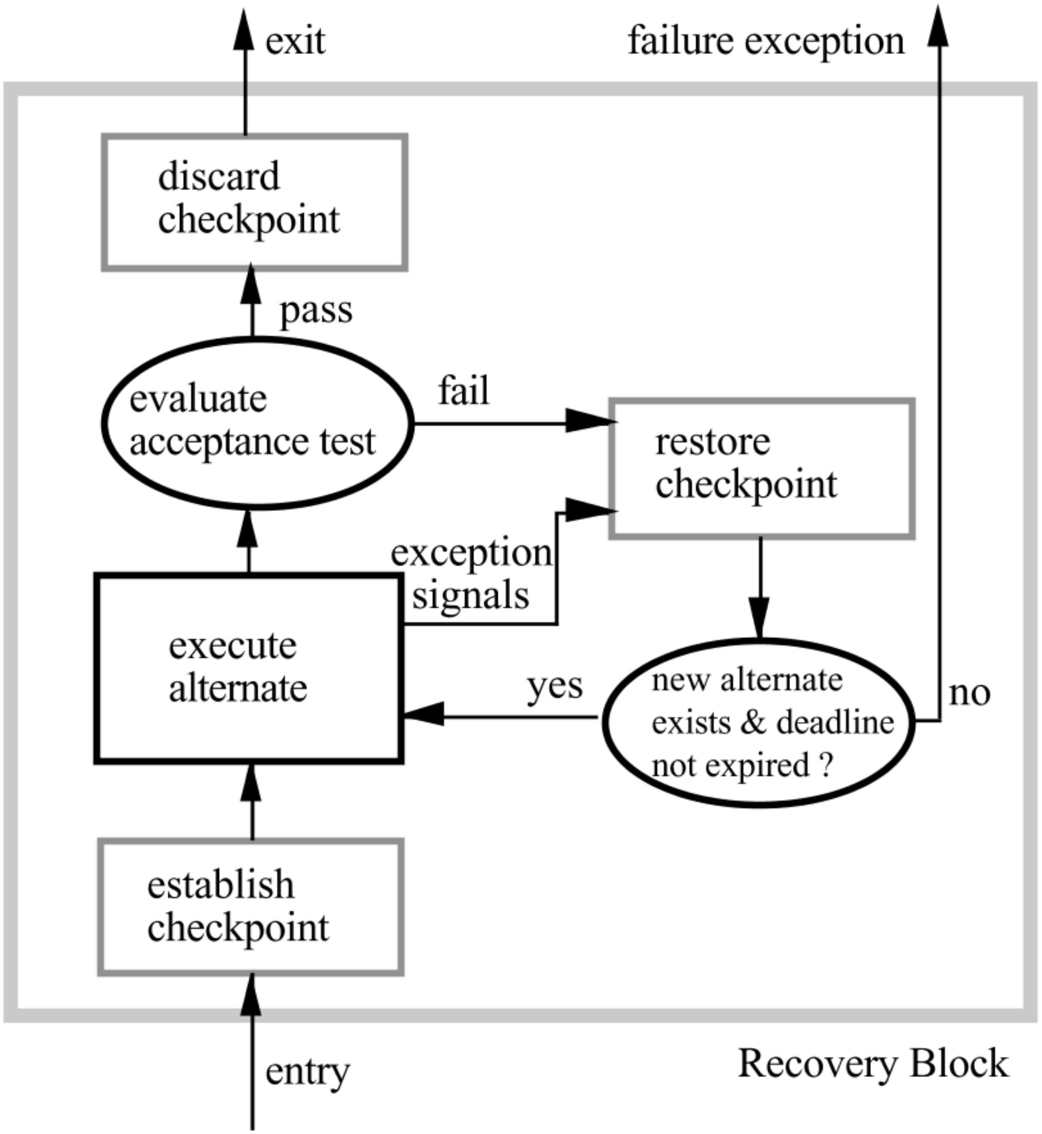


Basic Checkpointing Continued

- Backward recovery
- Very simple to implement
- Relies on existence of error detector
- Unbounded delay
  - Can have a limited number of retries for bounded delay
    - Fail on deadline
- Not actually that effective
  - Retrying the same computation is most likely just going to lead to the same error state

Approach 2: **Recovery Blocks**

- Record a checkpoint before execution begins
- Acceptance test must be able to detect error states from the results of component execution
  - This might not be knowable
- If the error detector detects an error
  - Reload checkpoint
  - Try another **variant**
    - Another implementation of the same algorithm that can start from the same state
    - No internal state!

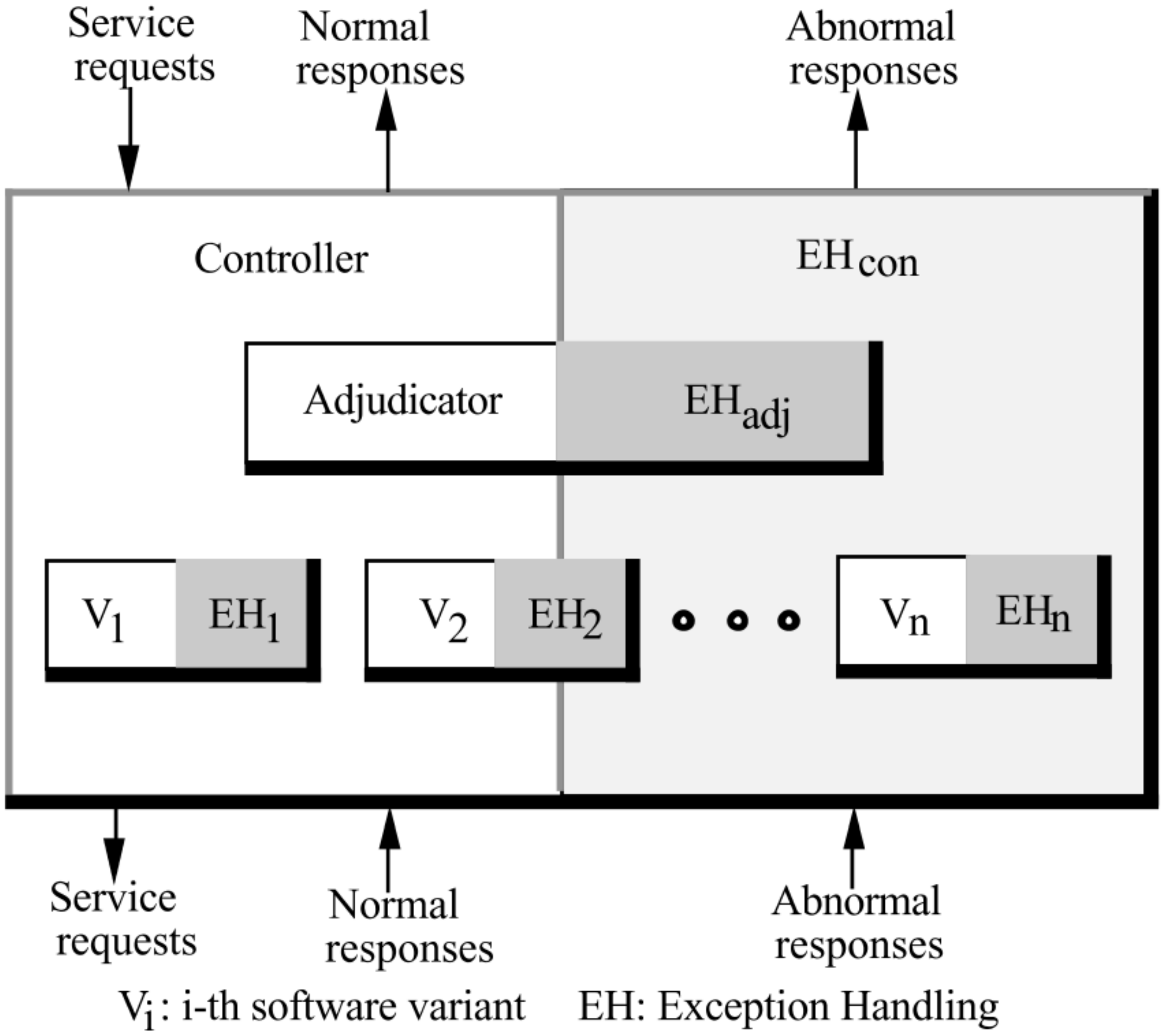


Recovery Blocks Continued

- Backward recovery
- Somewhat simple to implement
- Relies on existence of error detector
- Relies on existence of variants
  - Often there’s only one way to compute something
- Bounded delay
  - Fail after running out of variants, or reaching deadline
- More effective than basic checkpointing
  - Provided variants exist

Approach 3: **N-Version Programming**

- Run the variants concurrently
- An **adjudicator** selects the result between the completing variants
  - Sanity checks
  - Physical bounds
  - State model
- Variants pass state to their next iteration
  - Even if they're not selected



N-Version Programming Continued

“N-version programming is defined as the independent generation of functionally equivalent programs from the same initial specification."

- Algirdas A. Avizienis

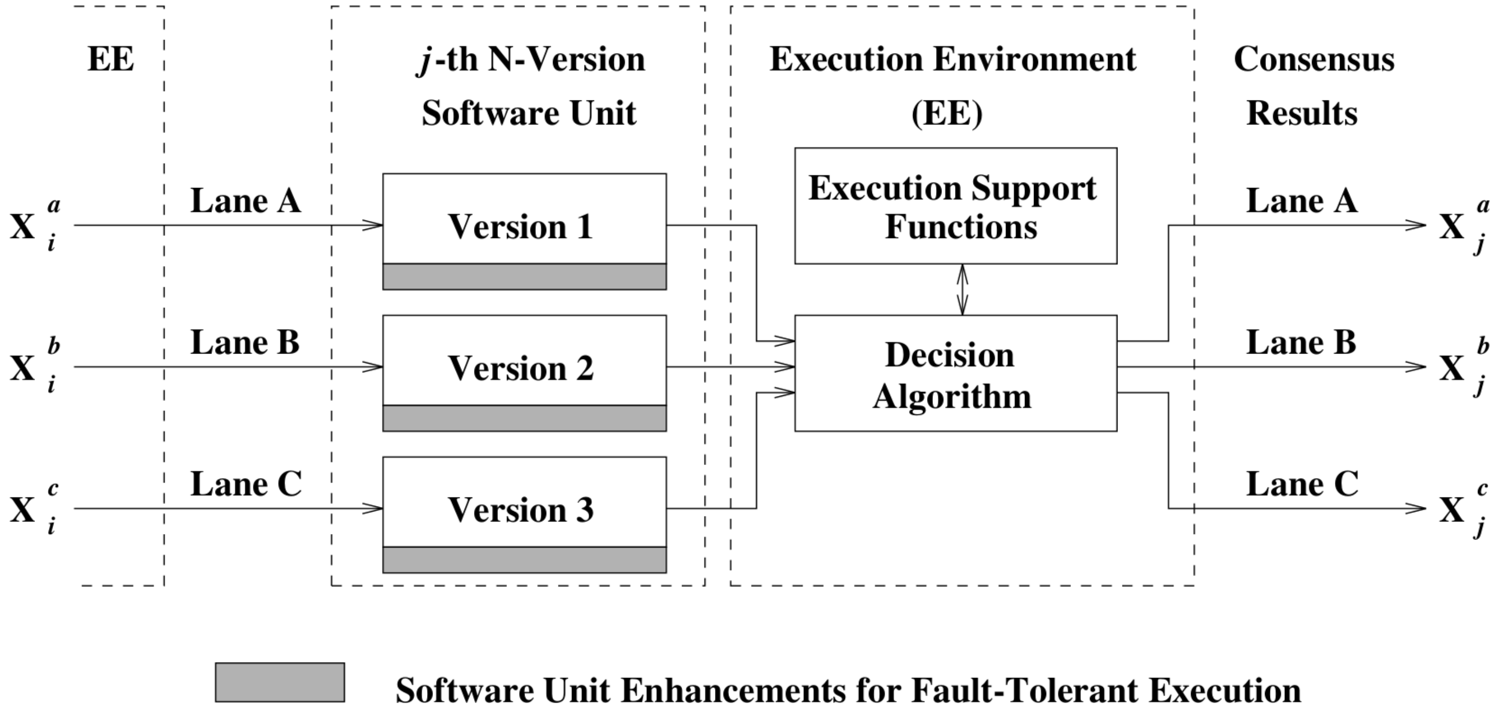
- More difficult to implement
- Relies on existence of adjudicator
  - Not too difficult
- Relies on existence of variants
  - Often there’s only one way to compute something
- Relies on concurrency
  - Or sufficient extra unused utilization to run additional variants in every period
- Variants can’t have common faults
  - Very easy to do by accident

N-Version Programming Continued 2

- Forward recovery
- Adjudicator decision making for
  - $n = 2$
    - Failure means pick other result
    - No decision between results
  - $n \geq 3$
    - Can vote among passing results
    - Simple majority or simple plurality
      - No independent member action

Implementing N-Version Programming

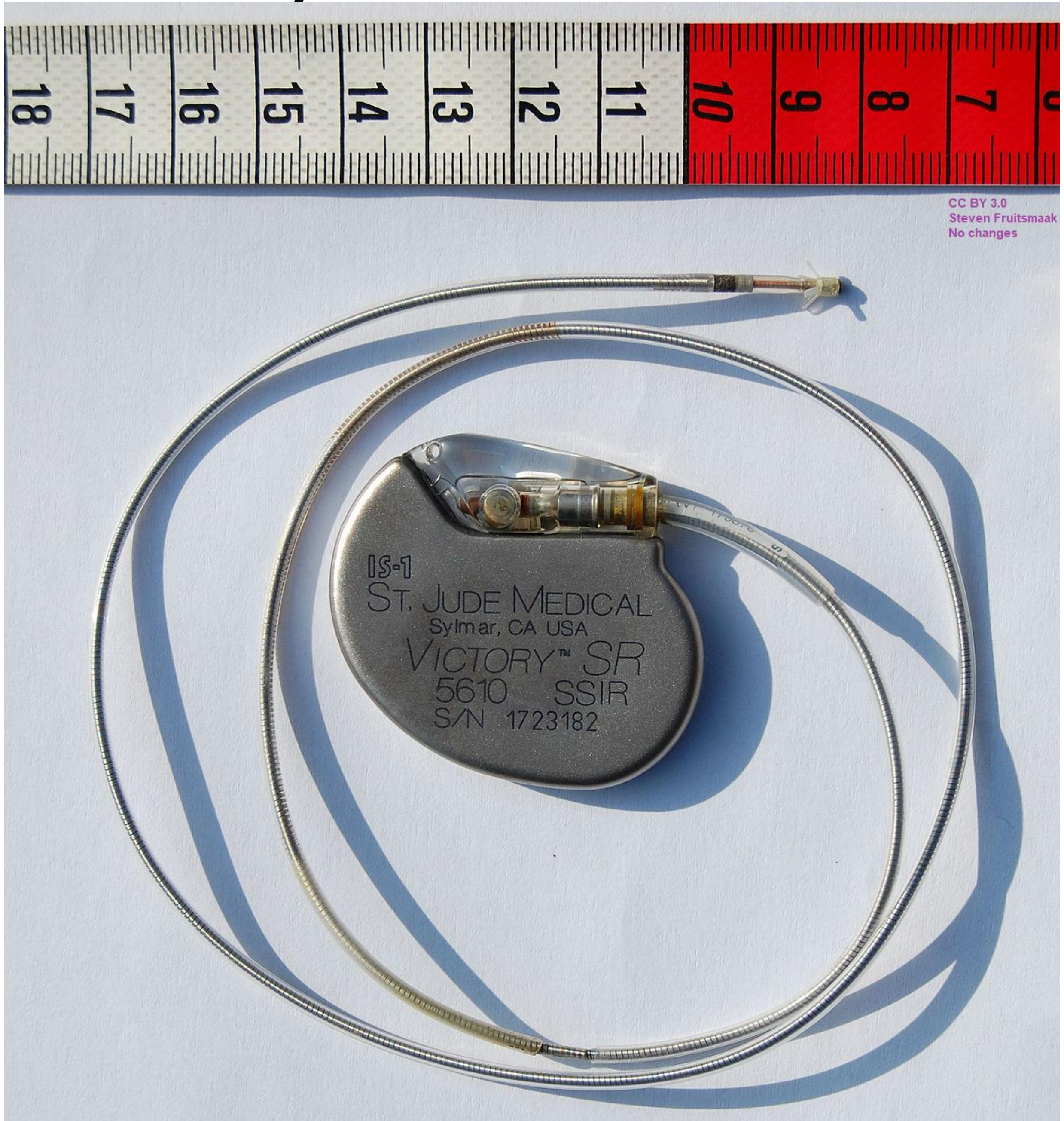
- Specify which units will be versioned
- Define execution environment for units
- Define state for next units
- Define decision algorithm for adjudicator



Implementing N-Version Programming

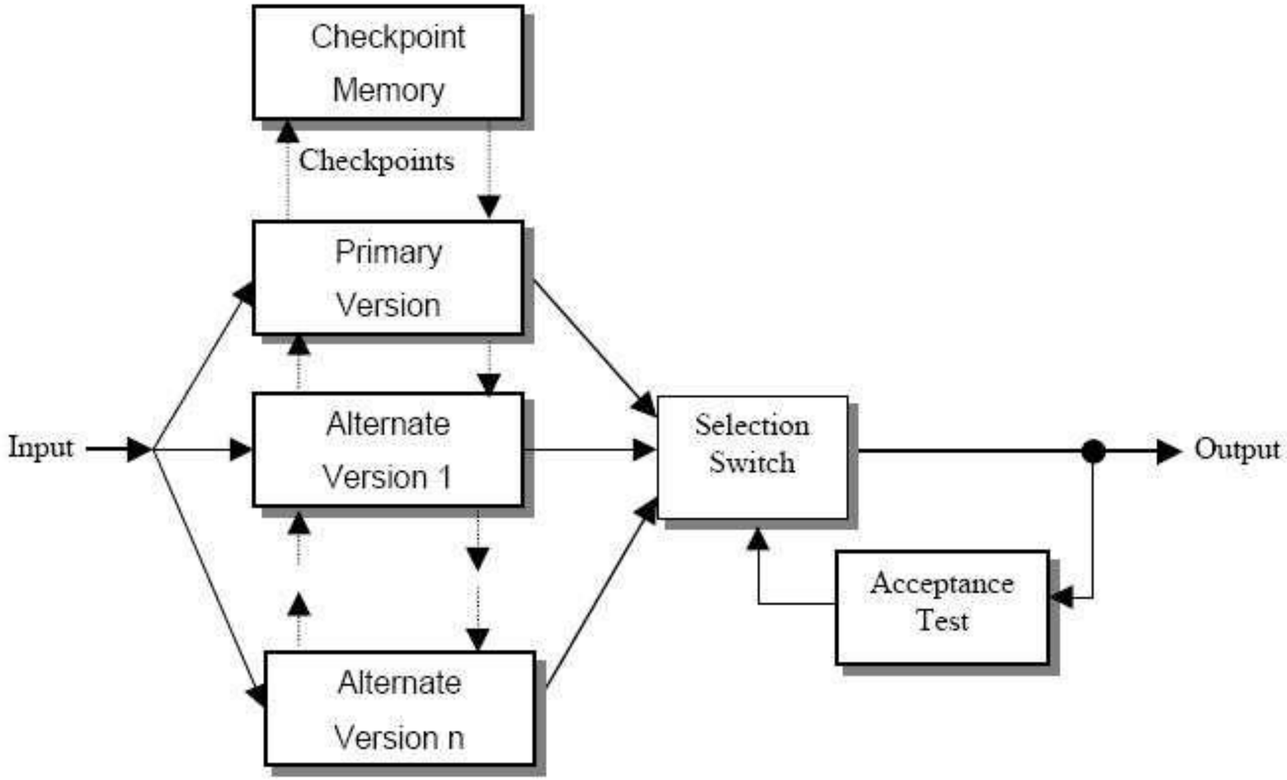
- Maximize **diversity**
  - Algorithm
  - Datatypes
  - Programming language
    - If possible
  - Toolchain
  - Programmer
  - Development process
- Gain security
  - Deliberate faults from compromised employees isolated
- Each variant can pass its own state to next iteration of itself

Disaster of the Day – Pacemaker Hack (2012)



Approach 4: **Consensus Recovery Blocks**

- Like NVP except selected variant state initializes all next iteration variant state
- Not to be confused with consensus algorithms
  - Which we will discuss in a few slides

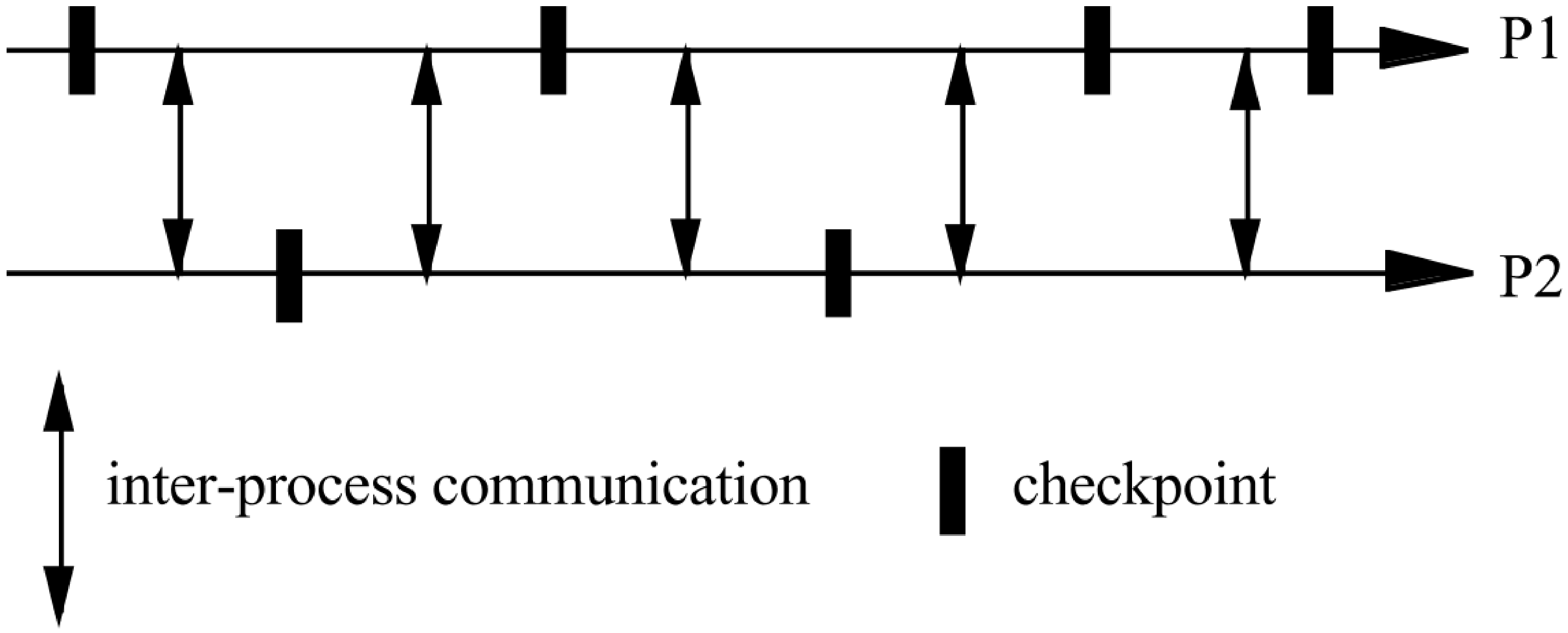


Consensus Recovery Blocks Continued

- Forward recovery
- Even more difficult to implement
  - Variants must initialize from common state
- Blocks reach “consensus” about system state
  - Less divergence between variants
  - More vulnerable to common fault
- Relies on concurrency, adjudicator, variants
  - Adjudicator has less role
    - Because less divergence between variants

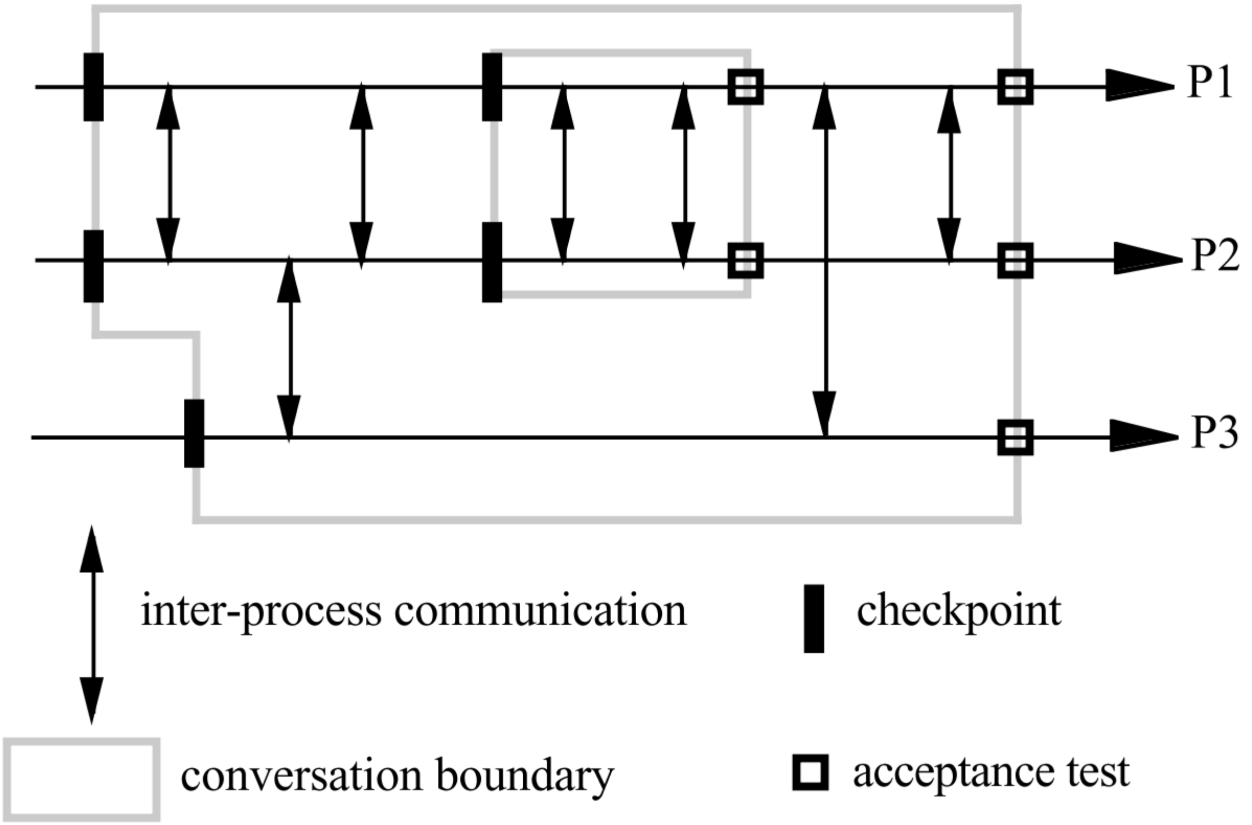
Distributed recovery blocks?

- What happens when we have multiple recovery blocks distributed across a distributed system?



Approach 5: **Conversations**

- All processes make a checkpoint before starting a conversation
- All processes must pass their acceptance test before advancing
  - If a single process fails, all processes revert to their checkpoints and try their next variant
  - Otherwise, all processes leave the conversation together



Conversations Continued

- **Deserters** are processes that are tardy in a conversation
  - Since everyone must wait for all acceptance tests
    - Now everyone misses their deadline
- **Information smuggling** is the process of information leaking from aborted conversations or recovery blocks
  - Resource consumption
  - Messages external to conversation
    - Not ideal, but sometimes unavoidable
- How do we make acceptance test fault tolerant?
  - Use one of our previous approaches
    - Communication issues?
  - Or…

Approach 6: **Consensus**

- Not to be confused with consensus recovery blocks

- **Two Generals Problem**
  - How do two Generals agree to attack a city together at the same time if they cannot guarantee their messengers will get through?
    - They can’t

- Try to do better with three or more generals
  - Still cannot in finite time
    - Not ideal for embedded systems, fail after some time

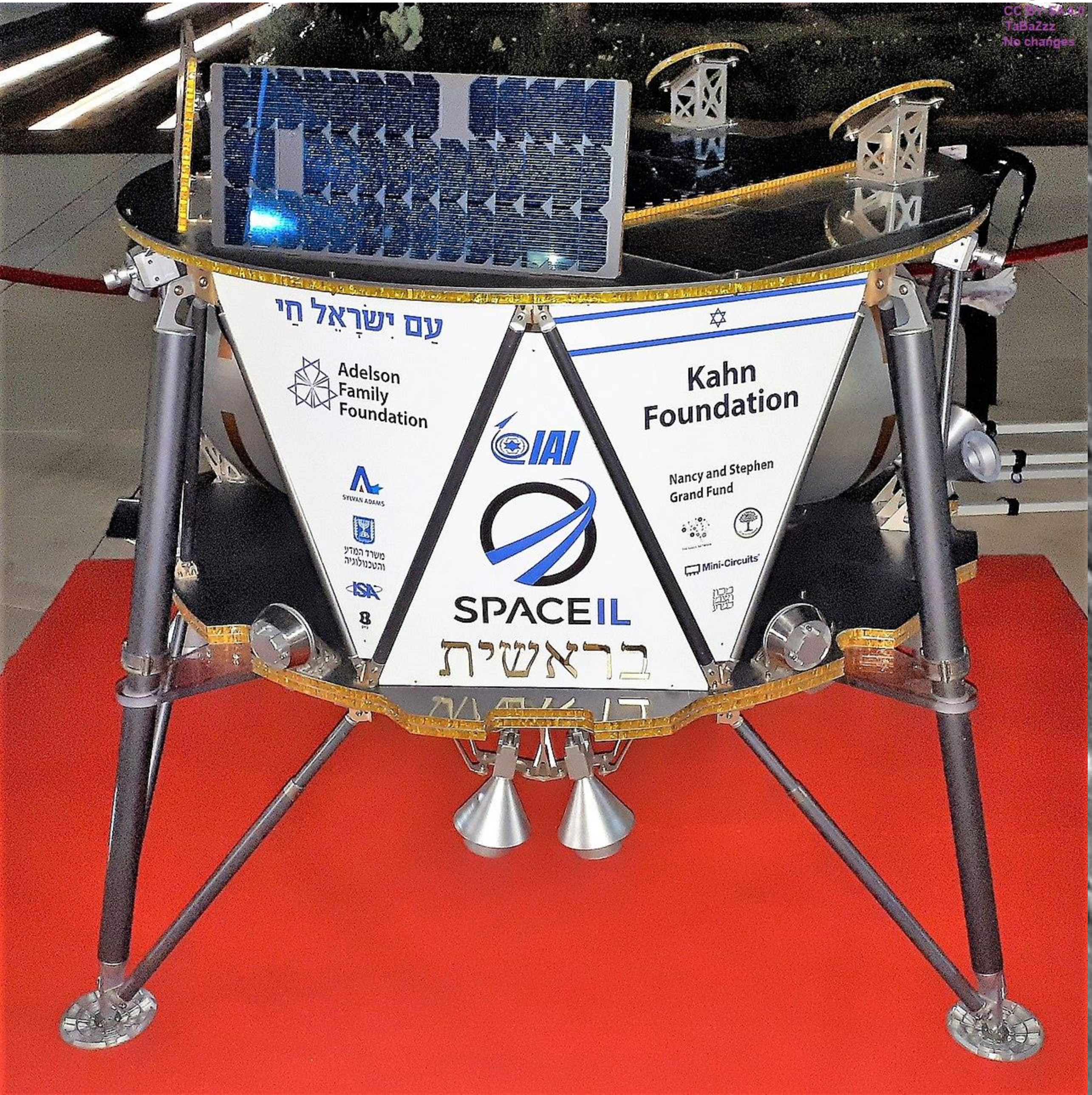
Consensus Continued

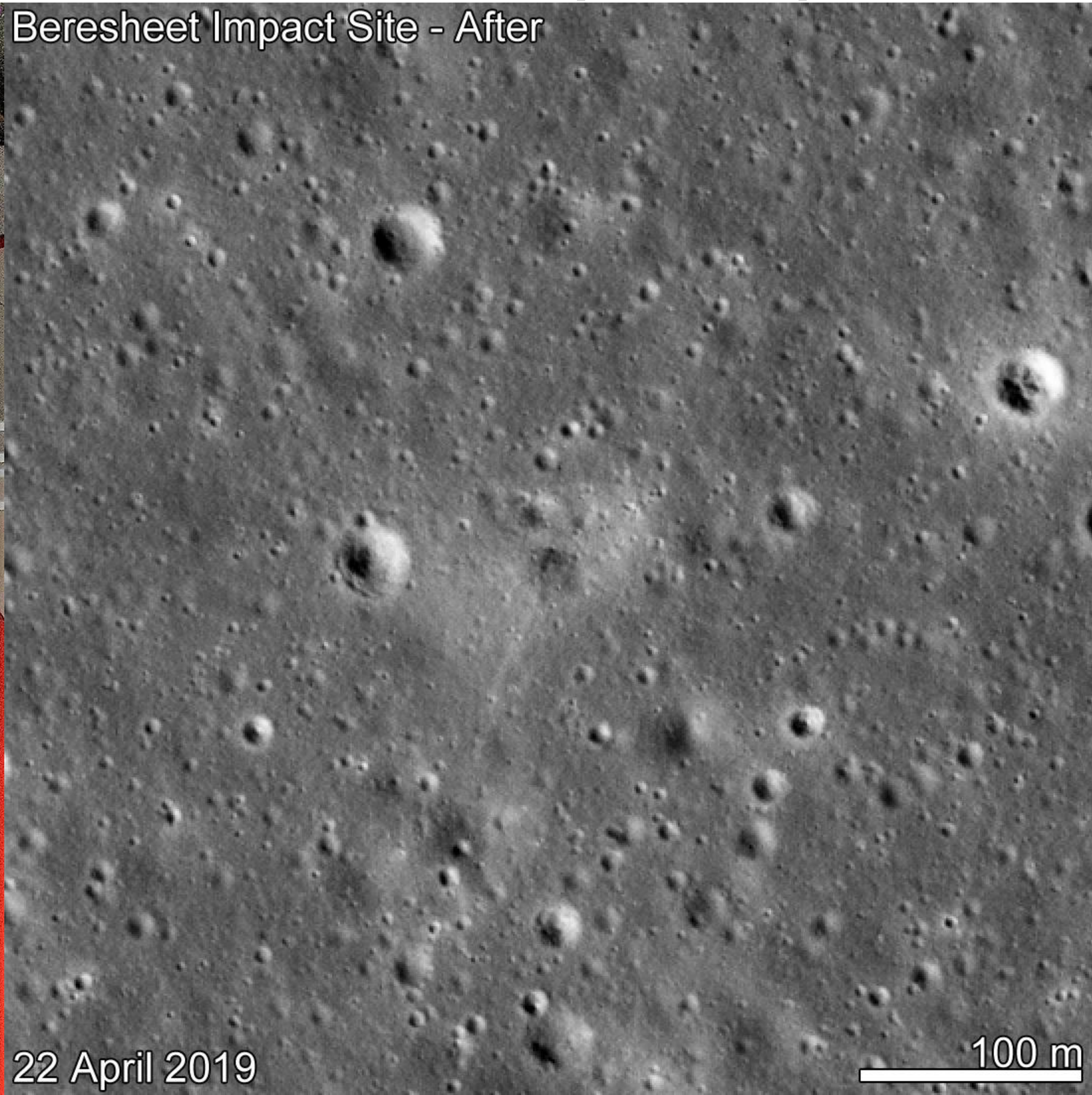
- Each member of a consensus protocol is called an **agent**
  - Each agent has some unique identifier (usually an integer)
- Properties of consensus
  - **Termination**
    - Every correct agent decides some value
  - **Agreement**
    - Every correct agent agrees on the same value
  - **Validity**
    - Only values suggested by some agent can be decided on
  - **Integrity**
    - If all the correct agents suggest the same value, each correct agent must decide on that value (often rolled into Agreement)

Traditional Consensus Algorithm: **Paxos**

- Voting based algorithm
  - Can tolerate f failures in a system of $2f + 1$ fail-stop agents
  - $f + 1$ is a **quorum** in a system of $2f + 1$ agents
- Two phases 1 and 2, each of two parts A and B
- Phase 1
  - Part 1a
    - Prepare for an agreement
  - Part 1b
    - Everyone promises this is the most recent agreement
- Phase 2
  - Part 2a
    - Propose value
  - Part 2b
    - Accept value

Disaster of the Day – Beresheet (2019)





Paxos Phase 1 Part 1

- Agent creates **prepare** message with a number n
  - This agent is the **proposer** for this paxos
- n must be greater than any number used in any previous prepare message from this agent
- Proposer sends the prepare n message to at least a quorum of agents
- Proposer should not initiate paxos if it cannot communicate with at least a quorum of agents

Paxos Phase 1 Part 2

- Each agent waits for a prepare message
  - This agent is an **acceptor** for this paxos
- Look at n
  - If n is higher than every previous proposal received from any proposer by this acceptor
    - Send a **promise** message back to the proposer with n
      - This is a promise that this acceptor will ignore all future proposals with lower n
    - Inform the proposer of any value v it previously accepted with lower n in phase 2 part 2
  - If this acceptor has already promised not to accept n
    - Don’t have to do anything, can just ignore the proposal
    - But optimization: tell the proposer no, and the current highest proposal

Paxos Phase 2 Part 1

- If the proposer receives a quorum of promises
  - If any of the acceptors indicated a previous value v
    - The proposer must choose the v with the highest n reported by the acceptors
  - Otherwise, the proposer is free to chose its own value of v
- The proposer sends an **accept** message to a quorum of acceptors with the same n as the propose message and the value v
  - This is a “please accept this proposal”

Paxos Phase 2 Part 2

- If an acceptor receives an accept message
  - If it has promised not to accept that n
    - Ignore the message
  - If it has not promised not to accept that n
    - It registers v as the value for this paxos
    - Send an **accepted** message with n and v to all agents
      - Telling them I have accepted this
- Consensus is reached when a quorum of acceptors accept a value for the same n

Disaster of the Day – Mars Global Surveyor (2006)

Bonus Phase

- Acceptors tell all agents the have accepted a value at n
  - Agents receiving this are **learners**
  - If all agents are acceptors, there is no need for learners
- A learner has learned a value when it gets a quorum of reports from acceptors
- Learning is the “output” of the consensus algorithm

Caveats and Counterintuitive Scenarios

- Acceptors can accept multiple values
- A value may achieve a majority across acceptors (but with different n) only to later be changed
- Acceptors may continue to accept proposals after n has achieved a quorum
- The counting of n resets for the next consensus the system wants to reach
  - n is a part of reaching one agreement
  - the next agreement is a whole new counting of n

Leader Elections

- Who gets to propose?
  - Kinda painful if everyone’s just proposing madly all the time
  - Lots of failed proposes
  - Lots of rejected accepts

- Vote for a leader!
  - Only the leader gets to propose
  - If the leader is unreachable for a while, propose yourself as a new leader
    - Do Paxos to agree on a leader

Byzantine Generals Problem

- Recall the Two Generals Problem
  - How do two Generals agree to attack a city together at the same time if they cannot guarantee their messengers will get through?
    - They can’t
- What if there’s more Generals but they can *lie*?
  - **Byzantine Faults** are faults where the failed subcomponent presents as failed or not failed differently to different observers
    - Often this is because the failed subcomponent is malicious

Byzantine Generals Problem Continued

- There is a group of generals
  - They must either decide to attack a city or retreat
  - A half-hearted attack is a disaster, have to commit one way or the other

- There is some number of impostor generals
  - They can lie
  - They can lie *selectively*
    - Send a message saying “I will attack” to some generals and “I will retreat” to others
    - Not send any message to some generals
  - **Byzantine agents**
    - When they fail as a subcomponent, it could be a **byzantine fault**
  - In some literature it’s the messengers that forge fake messages
    - This is the same result in the end

Byzantine Faults and Paxos

- Normal Paxos cannot handle Byzantine faults
  - But there is a **Byzantine Paxos** algorithm that can
    - Too complex for this class
    - Only tolerates f failures in a system of $\mathbf{3}f+1$ fail-stop agents

- There is another, very famous, modern algorithm that solves this problem for f failures in $2f+1$ byzantine agents…

Nakamoto Consensus

- Also known as proof of work
- For an agent to propose a new value, they must first solve a puzzle based on the last value
  - This is a traditionally a cryptographic challenge
    - Let v be the last value the proposer trusts
    - Consider some hash function $\mathrm{y} = h(x)$ that transforms an input x into a fixed-length bit sequence y
      - This function is computationally infeasible to reverse
      - Small changes in x lead to large changes in y
    - Solving the puzzle is finding some nonce n such that $h(v.n) = \mathrm{y}$ where y has some number of leading zeros
      - More leading zeros gives a harder puzzle
      - Solved by raw computational power
    - It is computationally easy for a node to verify $h(v.n) = \mathrm{y}$

Nakamoto Consensus Continued

- Once an agent finds n to solve the puzzle for v_n they can suggest a value v_{n+1} for the next consensus
- They are given some kind of reward to incentivize honesty
- All honest nodes now try to solve v_{n+1}
- If two agents both find some n at the same time
  - Could even be the same n, but likely different
  - May split the honest agents
    - Eventually one of the “chains” of values will be longer due to random chance solving the puzzle
    - All honest nodes work on the longest chain

Nakamoto Consensus Continued 2

- A dishonest agent may find some n and propose a new, bad value for v_{n+1}
  - Honest agents will see the bad value, and continue to try and find a solution for v_n
  - So long as there are more honest agents than dishonest agents the longest chain will remain honest
    - f failures in $2f+1$ byzantine agents