Verlässliche Echtzeitsysteme

Redundanz und Fehlertoleranz

Peter Wägemann

Lehrstuhl für Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität Erlangen-Nürnberg

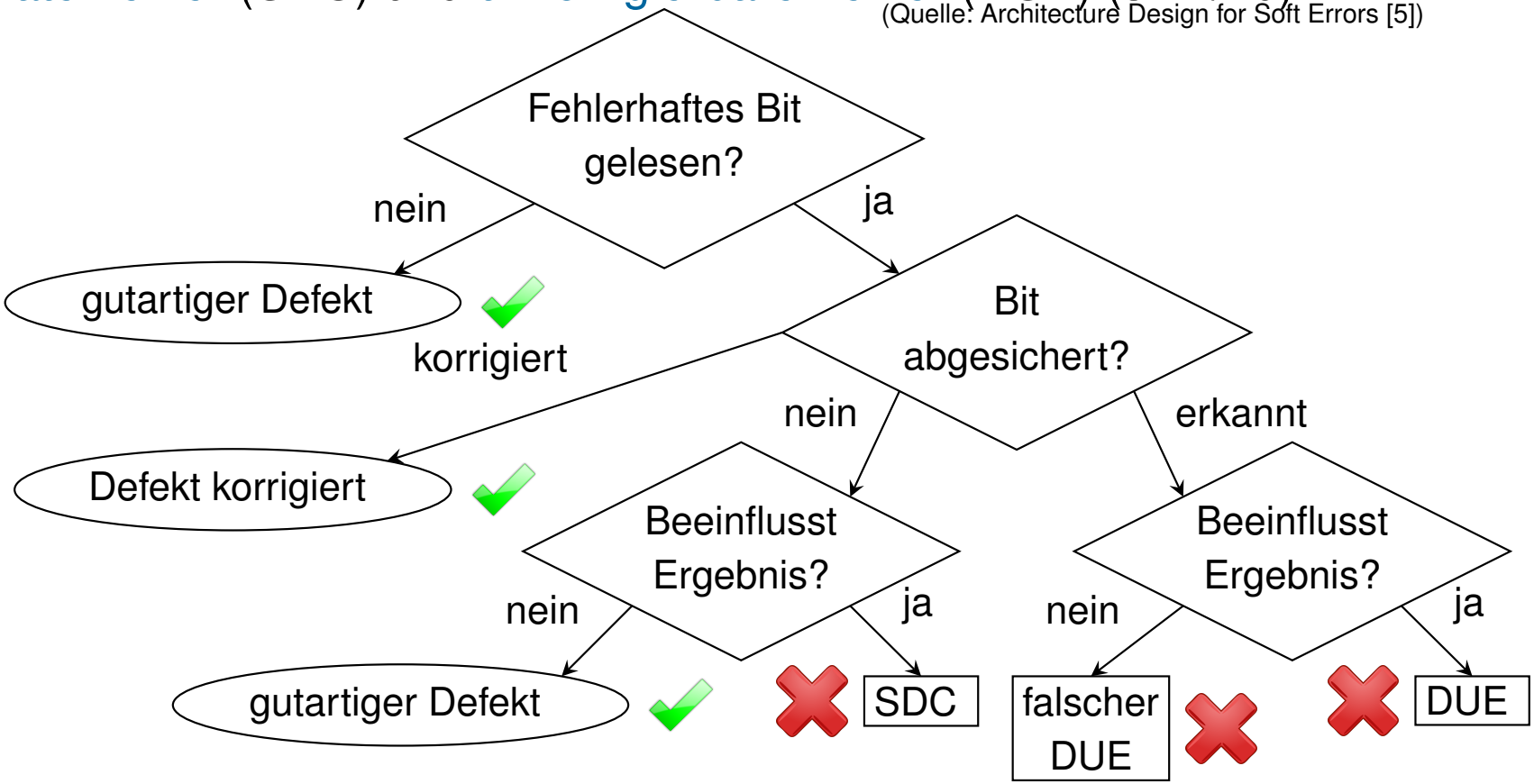
`https://www4.cs.fau.de`

26. April 2018

Wiederholung – Fehlerarten

- Erinnerung: transiente Fehler (engl. *soft errors*) (s. III/31)
  - Treten wie sporadische Fehler unregelmäßig auf . . .
  - Bewirken kurzzeitige Fehlfunktion $\rightsquigarrow$ Defekt, Fehler, Fehlverhalten
- Datenfehler (SDC) und unkorrigierbare Fehler (DUE) (s. III/10)

(Quelle: Architecture Design for Soft Errors [5])



Das Problem

Ungehemmte Fehlerfortpflanzung führt zum Systemversagen

- Unerkannte Datenfehler (engl. *silent data corruption*) (vgl. III/10)
  - Bedingen beispielsweise fehlerhafte Stellwerte für Aktoren
  - Ihre Folgen treten häufig räumlich und zeitlich unkorreliert auf
- Erkannte, unkorrigierbare Fehler (engl. *detected unrecoverable errors*)
  - Führen zu einem unmittelbaren, erkennbaren Systemversagen

Vermeidung dieser Fehler ist erforderlich

- Problematik: Robuste Auslegung aller Komponenten ist häufig nicht möglich
  - Diese müssten frei von konzeptionellen Fehler sein
    (↦ keinerlei Hardware- oder Softwaredefekte)
  - Sie müsste widrigen äußeren Umständen trotzen
- Lösung: Ein System, welches Fehler tolerieren kann
  - Einzelne Komponenten (HW/SW) können (dürfen) ausfallen
  - Dies wird durch andere redundante Komponenten aufgefangen
  - Die gewünschte Funktionalität an der Schnittstelle bleibt erhalten
  - Der Anwender bekommt davon möglichst nichts mit (⇝ Transparenz)

Übersicht – Fehlertoleranz durch Redundanz

Was tun gegen unzuverlässige Komponenten?

- Redundanz als Grundlage von Fehlertoleranz
  - Welche Arten von Redundanz existieren?
  - Welche Eigenschaften verknüpfen sich hiermit?
  - Auf welcher Ebene wird Redundanz angewandt?
- Hardwarebasierte Replikation
  - Klassische Lösung für die Auslegung fehlertoleranter Systeme
  - Replikation auf Ebene des Knotens bzw. der Hardware
  - Fokussierung auf Triple Modular Redundancy
- Softwarebasierte Replikation
  - Process-Level Redundancy: Zuhilfenahme von Mehrkernprozessoren
  - Replikation auf Ebene von Prozessen bzw. Software

  → Maskierung transienter Hardwarefehler durch redundante Ausführung
- Vermeidung von Gleichtaktfehlern durch Diversität
  - „Replizierte Entwicklung“ der einzelnen Redundanzen

Gliederung

Ziele von Fehlertoleranz

Was man mit dem Mehraufwand eigentlich bezweckt!

- Fehlererkennung (engl. *fault detection*)
  - Erkennen von Fehlern z. B. mithilfe von Prüfsummen
- Fehlerdiagnose (engl. *fault diagnosis*)
  - Identifikation der fehlerhaften (redundanten) Einheit
- Fehlereindämmung (engl. *fault containment*)
  - Verhindern, dass sich ein Fehler über gewisse Grenzen ausbreitet
- Fehlermaskierung (engl. *fault masking*)
  - Dynamische Korrektur von Fehlern z. B. durch Mehrheitsentscheid
- Wiederaufsetzen (engl. *recovery*)
  - Wiederherstellen eines funktionsfähigen Zustands nach Fehlern
    - Reparatur (engl. *repair*) bzw. Rekonfiguration (engl. *reconfiguration*)

☞ Fokus der Vorlesung: Fehlererkennung und Fehlermaskierung

Arten von Redundanz

Redundanz ist eine Grundvoraussetzung für Fehlertoleranz

Strukturelle Redundanz (dieses Kapitel)

→ Bereitstellung mehrerer gleichartiger Komponenten

- Replikation ⇝ (typisch) hardwarebasierte Fehlertoleranzlösungen
- Mehrfache Auslegung: Prozessoren, Speicher, Sensoren, Aktoren, ...

Funktionelle Redundanz

→ Bereitstellung mehrerer verschiedenartiger Komponenten

- Mehrfache Herleitung desselben Sachverhalt auf verschiedenen Wegen
- Ventilstellung ⇝ Stellungsgeber bzw. Durchflussmengenmesser
- Funktionswächter (engl. *watchdog*) für bestimmte Parameter

Informationsredundanz (vgl. Kapitel 5)

→ Einbringung zusätzlicher Informationen/Daten (nicht zwingend erforderlich)

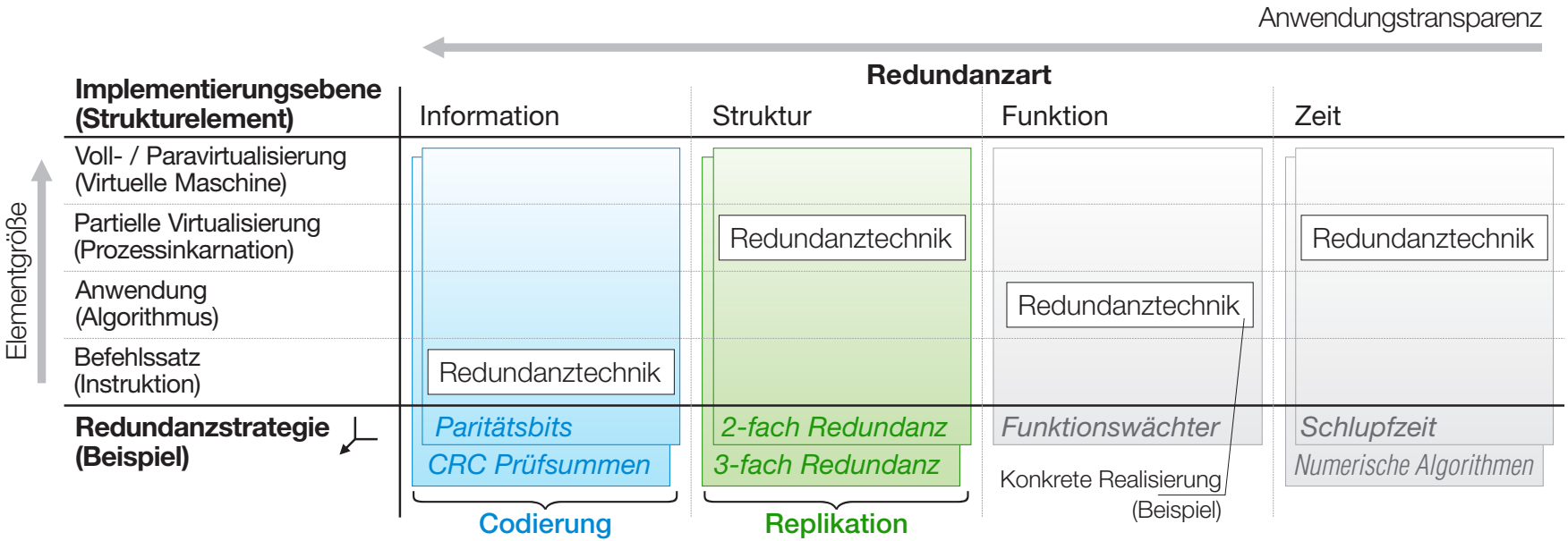
- Speicherung von Brutto- und Nettobetrag
- Typischerweise in Form von Codierung (Prüfsummen, CRC, ...)

Zeitliche Redundanz

→ Bereitstellung von über den Normalbetrieb hinausgehender Zeit

- Z.B. Numerische Algorithmen, Schlupf in einem EZS, ...

Koordinierter Einsatz von Redundanz



Erst der koordinierte Einsatz von Redundanz ermöglicht Fehlertoleranz

In VEZS: Klassifizierung nach Fehlererkennung ($\neq$ Literatur)

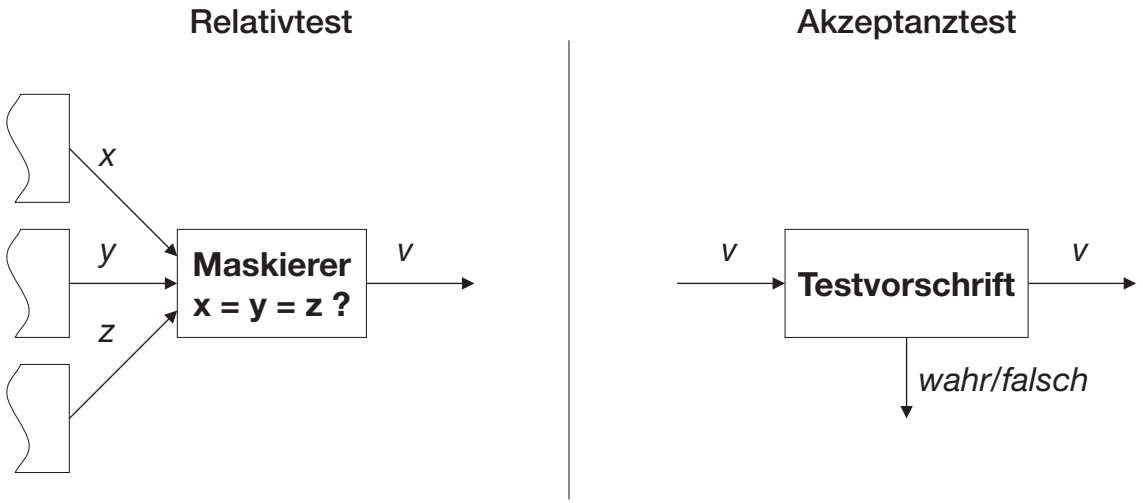
Es existieren viele Implementierungsalternativen

- Implementierungsebene (vgl. Sichtbarkeit Folien III/7 ff)
- Art der Redundanz und Erkennungsstrategie
- Anwendungstransparenz[1]

→ Konkrete Redundanztechnik

[1] Funktionelle und zeitliche Redundanz lassen sich in der Praxis nur fallspezifisch implementieren und stehen daher außerhalb des Veranstaltungskontextes.

Fehlererkennung – Grundlagen



⚠ Zwei Testverfahren zur Fehlererkennung (vgl. [3, S.78 ff])

1 Relativtest (engl. *comparison test*) (auch Vergleichstest)

- Ist-Ist-Vergleich auf Übereinstimmung ⇝ anwendungsunabhängig
- Erfordert mehrere Vergleichsobjekte

→ Mehrheitsentscheider (engl. *voter*) (auch Maskierer)

⚠ Ausschließlich bei struktureller Redundanz anwendbar

2 Akzeptanztest (engl. *acceptance test*) (auch Absoluttest)

- Soll-Ist-Vergleich auf Konsistenzbedingung ⇝ anwendungsabhängig
- Ein Testobjekt genügt

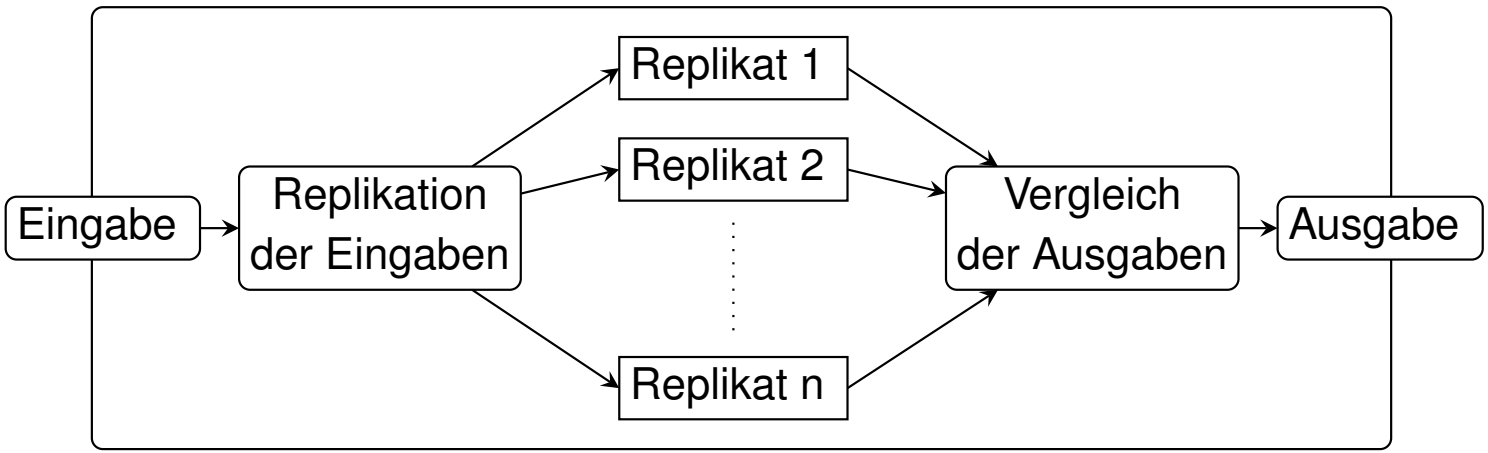
⚠ Vollständigkeit der Testbedingung ist das Problem

Gliederung

Replikation

Replikation ist der koordinierte Einsatz struktureller Redundanz

- Aufbau eines Replikationsbereichs (engl. *Sphere of Replication, SoR*) [6]
  - Sie maskiert transparent Fehler in einzelnen Replikaten



  - Eingaben werden repliziert und auf die Replikate verteilt
  - In einem Ausgangsvergleich werden die Ausgaben abgestimmt

Offene Fragestellungen:

- Wie viele Replikate benötigt man, um das zuverlässig tun zu können?
- Welche Voraussetzungen müssen für eine erfolgreiche Replikation gelten?

Wie viele Replikate braucht man?

- Zahl benötigter Replikate hängt von der Art des Fehlverhaltens ab [4]
  - Annahme: von n Replikaten sind in folgender Weise f fehlerhaft

„fail-silent“ $\mapsto$ Anzahl der Replikate $n = f + 1$

  - Ein Replikat erzeugt korrekt oder gar keine Antworten
  - Das Fehlerverhalten führt zum Stillstand

  → Einfachster Fehlermodus

  - Der Ausfall wird von den anderen Replikaten als solcher erkannt

„fail-consistent“ $\mapsto$ Anzahl der Replikate $n = 2f + 1$

  - Ein Replikat kann auch fehlerhafte Antworten erzeugen
  - Alle anderen Replikate sehen konsistent dasselbe Fehlverhalten

„malicious“ $\mapsto$ Anzahl der Replikate $n = 3f + 1$

  - „bösartige“, fehlerhafte Replikate erzeugen verschiedene Antworten
    - Keine konsistente Sicht auf das Fehlverhalten
    - Typischerweise verursacht durch Komm’system (Nachrichtenausfall!)
  - Synonym: byzantinische Fehler (engl. *byzantine failures*)

Fehlerhypothese (engl. *fault hypothesis*)

Annahmen über das Verhalten einzelner Replikate im Fehlerfall

In der Praxis betrachtet man für Echtzeitsysteme Replikate, die:

- Einen Fehler tolerieren können
- Sich „fail-silent“ oder zumindest „fail-consistent“ verhalten
- Unabhängig voneinander ausfallen (vgl. Folie 14 f)
- Sich Replikdeterministisch verhalten (vgl. Folie 16 ff)

Byzantinische Fehlertoleranz wird üblicherweise nicht angestrebt

- Grund ist der enorme Aufwand, der damit verbunden ist
- $3f + 1$ Replikate um f Fehler zu tolerieren
- Getrennte Kommunikationswege zwischen allen Replikaten
  - Hoher Hardwareaufwand für Replikate und Verkablung

  → hohe Kosten, Gewicht, Energieverbrauch
- Erkennung fehlerhafter Replikate erfordert aufwendige Kommunikation
  - $f + 1$ Kommununikationsrunden für $3f + 1$ Replikate und f Fehler
  - Je Runde schickt jedes Replikat eine Nachricht an alle anderen Replikate

  → Für Echtzeitsysteme ein nicht tolerierbarer zeitlicher Aufwand

Fehlerisolation

☞ Replikate müssen unabhängig voneinander ausfallen

- Gleichtaktfehler (engl. *common mode failures*) sind zu vermeiden
  - Sie führen zum gleichzeitigen Ausfall mehrerer Replikate
  - ⚠ Eine Fehlermaskierung ist in diesem Fall nicht mehr möglich
- Quellen für Gleichtaktfehler
  - Permanente Fehler in den Komponenten (vgl. Ariane 5 Kapitel II/16 ff)
  - Übergreifen eines Fehlers auf andere Replikate (Fehlerausbreitung)

⚠ Einzelne Replikate sind gegeneinander abzuschotten

- Räumliche Isolation des internen Zustands
  - Dieser darf nicht durch andere Replikate korrumpiert werden
  - Ein verfälschter Zeiger hat großes Schadenspotential
- Zeitliche Isolation anderer Aktivitätsträger
  - Eine Monopolisierung der Ausführungseinheiten ist zu verhindern
  - Ein Amok laufender Faden könnte in einer Schleife „festhängen“
  - Selbiges gilt für alle gemeinsamen Betriebsmittel

Lose Kopplung unterstützt Isolation

☞ Ziel sind lose gekoppelte Replikate

- Minimierung des Koordinations- und Kommunikationsaufwands
  - Je weniger sich einzelne Replikate abstimmen müssen, umso besser

→ Fehlerausbreitung wird auf diese Weise effektiv vermieden

- Unterstützung durch eine statische, zyklische Ablaufstruktur

1. Eingaben lesen
   - Der Zustand des kontrollierten Objekts wird erfasst
2. Berechnungen durchführen
   - Der neue Zustand wird aus dem alten Zustand und den Eingaben berechnet
3. Ausgaben schreiben
   - Die Stellwerte werden an die Aktoren ausgegeben

- Lediglich die Schritte 1 und 3 erfordern eine Abstimmung der Replikate
  - Austausch von Nachrichten zwischen den Replikaten, um durch ein Einigungsprotokoll einen Konsens über die Eingaben/Ausgaben zu erzielen
- Die Berechnung wird von jedem Replikat in „Eigenregie“ durchgeführt
  - Ermöglicht einen unterbrechungsfreien Durchlauf (engl. *run-to-completion*)

Replikdeterminismus

Korrekt arbeitende Replikate müssen identische Ergebnisse liefern.

☞ Replikate sind replikdeterministisch (engl. *replica determinate*), wenn:

- Ihr von außen beobachtbarer Zustand identisch ist, und . . .
- Sie zum ungefähr gleichen Zeitpunkt identische Ausgaben erzeugen
  - Sie müssen innerhalb eines Zeitintervalls der Länge d erzeugt werden
  - Im Bezug auf einen gemeinsamen Referenzzeitgeber

- Warum ist Replikdeterminismus wichtig?
  - Replikdeterminismus ist eine Grundvoraussetzung für aktive Redundanz!
  - Korrekte Replikate könnten sonst unterschiedliche Ergebnisse liefern
    - Ein Mehrheitsentscheid ist in diesem Fall nicht mehr möglich
  - In den Replikaten kann der interne Zustand divergieren
    - Unterschiedliche Ergebnisse sind die logische Folge
    - Ein im Hintergrund laufendes Replikat kann im Fehlerfall nicht übernehmen
  - Außerdem wird die Testbarkeit verbessert
    - Schließlich kann man präzise Aussagen treffen, wann welche Ergebnisse von den einzelnen Replikaten geliefert werden müssten

Phänomene, die Replikdeterminismus verhindern

- Abweichende Eingaben bei verschiedenen Replikaten
  - Digitalisierungsfehler, z. B. bei der Analog-Digital-Wandlung
    - Temperatur- oder Drucksensoren liefern zunächst eine Spannung
    - Diese Spannungen werden in einen diskreten Zahlenwert überführt
    - Abbildungen kontinuierlicher auf diskrete Werte sind fehlerbehaftet
  - Dies betrifft auch die Diskretisierung der physikalischen Zeit

    → unterschiedliche Reihenfolge beobachteter Ereignisse
- Unterschiedlicher zeitlicher Fortschritt der einzelnen Replikate
  - Oszillatoren verschiedener Replikate sind nie exakt gleich

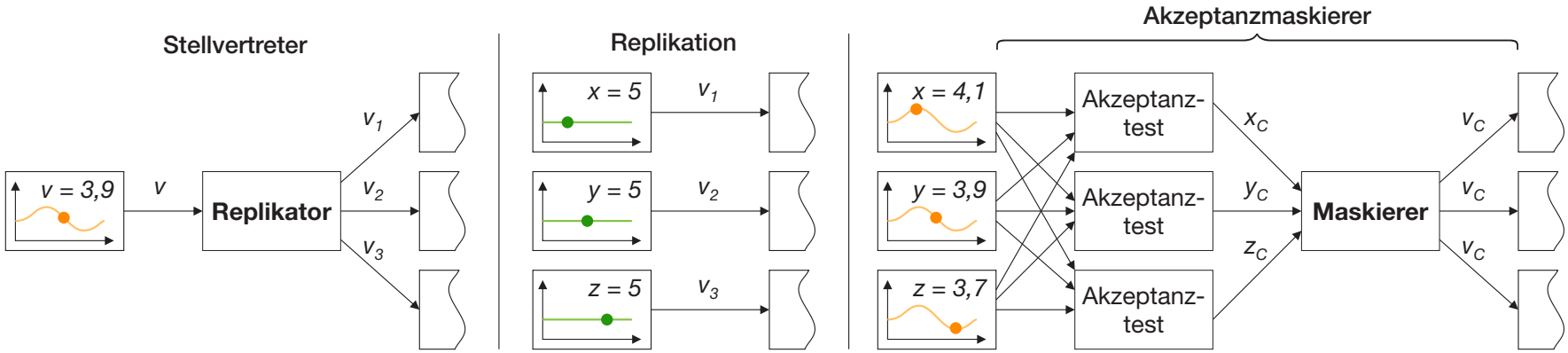
    → Vor allem der Zugriff auf die lokale Uhr ist problematisch
    - u. U. werden lokale Auszeiten (engl. *time-outs*) deshalb gerissen
- Präemptive Ablaufplanung ereignisgesteuerter Arbeitsaufträge
  - Diese bearbeiten u. U. unterschiedliche interne Zustände
    - Die evtl. aus Wettlaufsituation (engl. *data races*) erwachsen sind
- Nicht-deterministische Konstrukte der Programmiersprache
  - z. B. die SELECT-Anweisung der Programmiersprache Ada

Wie stellt man Replikdeterminismus sicher?

- Globale diskrete Zeitbasis
  - Ermöglicht eine globale zeitliche Ordnung relevanter Ereignisse
    - Ohne dass sich die Replikate hierfür explizit einigen müssen
  - Es dürfen keine lokale Auszeiten verwendet werden
    - Betrifft die Anwendung, Kommunikations- und Betriebssystem
- Einigung über die Eingabewerte
  - Die Replikate führen hierzu ein Einigungsprotokoll durch
    - Konsistente Sicht bzgl. Wert und Zeitpunkt der Eingabe

    → Grundlage für die globale zeitliche Ordnung aller Ereignisse
- Statische Kontrollstruktur
  - Kontrollentscheidungen sind unabhängig von Eingabedaten
    - Ermöglicht außerdem eine statische Analyse dieser Entscheidungen
  - Programmunterbrechungen sind mit größter Vorsicht einzusetzen
- Deterministische Algorithmen
  - Keine randomisierten Verfahren, nur stabile Sortierverfahren, . . .

Replikation der Eingänge



- Deterministische, seiteneffektfreie Eingänge
  - Mehrfaches Auslesen (Replikation) des Wertes möglich

  → Zustandswerte, welche im Betrachtungszeitraum ihre Gültigkeit bewahren
- Indeterministische Eingänge

  ⚠ Mehrfaches Auslesen führt zu unterschiedlichen Werten
  - Einsatz eines Stellvertreters
    - Liest den Wert einmal und dupliziert diesen ↦ Keine Redundanz!
  - Alternative: Akzeptanzmaskierer
    - Kombination aus Akzeptanz- und Relativtest
    - Anwendungsspezifische Übereinstimmungsbedingung

Zuverlässigkeitsgewinn durch Replikation

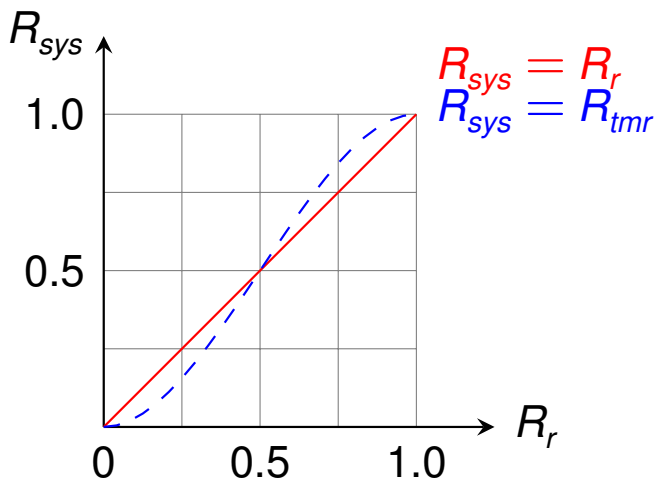
☞ Erhöht sich durch Replikation in jedem Fall die Zuverlässigkeit?

- Anders formuliert: $R_{tmr} > R_r$?
  - R_{tmr} – Zuverlässigkeit des TMR-Verbunds, R_r des einzelnen Replikats

Die Replikation arbeitet korrekt, solange ...

- Der Mehrheitsentscheid korrekt funktioniert $\rightsquigarrow R_v$
- Zwei Replikate korrekt funktionieren $\rightsquigarrow R_{2/3} = R_r^3 + 3R_r^2(1-R_r)$
  - Alle drei Replikate arbeiten korrekt oder ...
  - Ein Replikat fällt aus, hierfür gibt es drei Möglichkeiten

→ Insgesamt $R_{tmr} = R_v(R_r^3 + 3R_r^2(1-R_r))$



☞ Annahme: Perfekter Voter $R_v = 1$

⚠ TMR ist nur sinnvoll falls $R_r > 0.5$

⚠ Praxis: Voter sollte zuverlässig sein

- Größenordnung $R_v > 0.9$

Verbliebene kritische Bruchstellen

Kritische Bruchstellen (engl. *single points of failure*)

- Führen zu einem beobachtbaren Fehlerfall innerhalb der Fehlerhypothese
- Kompromittieren die fehlertolerierende Eigenschaft des Redundanzbereichs

→ Im Beispiel auf Folie 11 sind dies Eingabe und Ausgabe

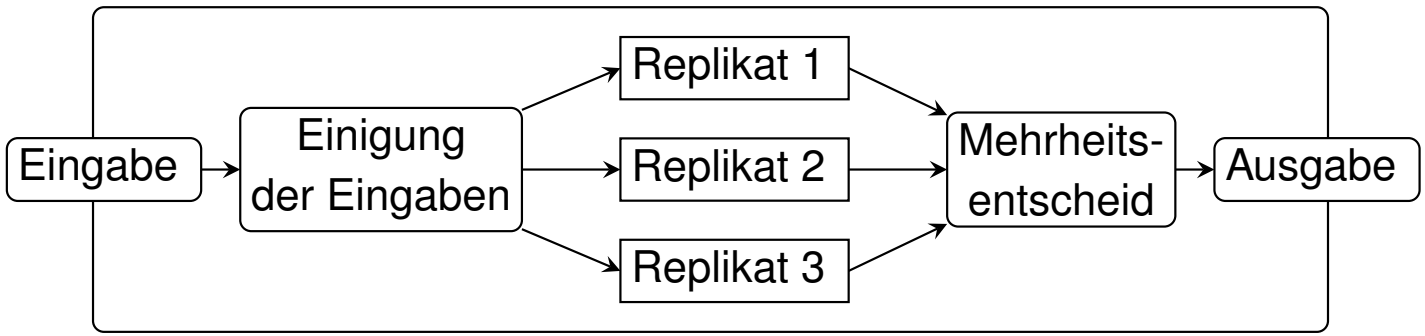
Lösungsmöglichkeiten

- Bestimme Eingabedaten aus mehreren Sensoren
  - Dies erfordert eine Einigung der Replikate über den Eingabewert, allen muss exakt derselbe Wert zugestellt werden
  - Anwendung funktionaler Redundanz ⇝ Sensorfusion (engl. *sensor fusion*)
- Repliziere den Ausgangsvergleich
  - Erneuter Mehrheitsentscheid über die Ergebnisse des replizierten Vergleichs

  → Das ist wieder eine kritische Bruchstelle, aber die Fehlerwahrscheinlichkeit sind insgesamt geringer, verschwinden tut sie nie . . .
- Robuste Implementierung des Ausgangsvergleichs
  - Zusätzliche Absicherung des Ergebnisses
  - Durchführung des Mehrheitsentscheids durch den Aktor

Gliederung

Hardware Triple Modular Redundancy (TMR)



- Üblicherweise dreifache Replikation kompletter Rechenknoten
  - Räumlich redundante Systeme

  → Weitgehende räumliche und zeitliche Isolation
- Abstimmung der Eingabewerte zwischen den Replikaten
  - Die Replikate verfügen über eine gemeinsame globale Zeitbasis
  - Das Kommunikationssystem verhindert die Steuerfehlerausbreitung

  → Vollständige zeitliche Isolation [7, Kapitel 8] und Replikdeterminismus
- Mehrheitsentscheid stimmt Ausgabewerte ab
  - Vereinigung von Fehlermaskierung und -erkennung

Hardwarebasierte Replikation

Hot standby: Rechensysteme arbeiten simultan

- Sie verarbeiten gleichzeitig dieselben Eingaben
- Ihr Zustand ist jederzeit konsistent

  → nahtloser Ersatz für ausgefallene Replikate

Warm standby: Unterscheidung von Primär- und Sekundärsystem

- Sekundärsystem läuft im Hintergrund
  - Regelmäßige Zustandssicherung (engl. *checkpoint*) des Primärsystems
  - Rückkehr zur letzten Sicherung im Fehlerfall (engl. *recovery*)
- Primär- und Sekundärsystem sind zeitweise inkonsistent

  → Höherer Aufwand im Falle der Fehlererholung

Cold standby: Sekundärsystem startet im Fehlerfall

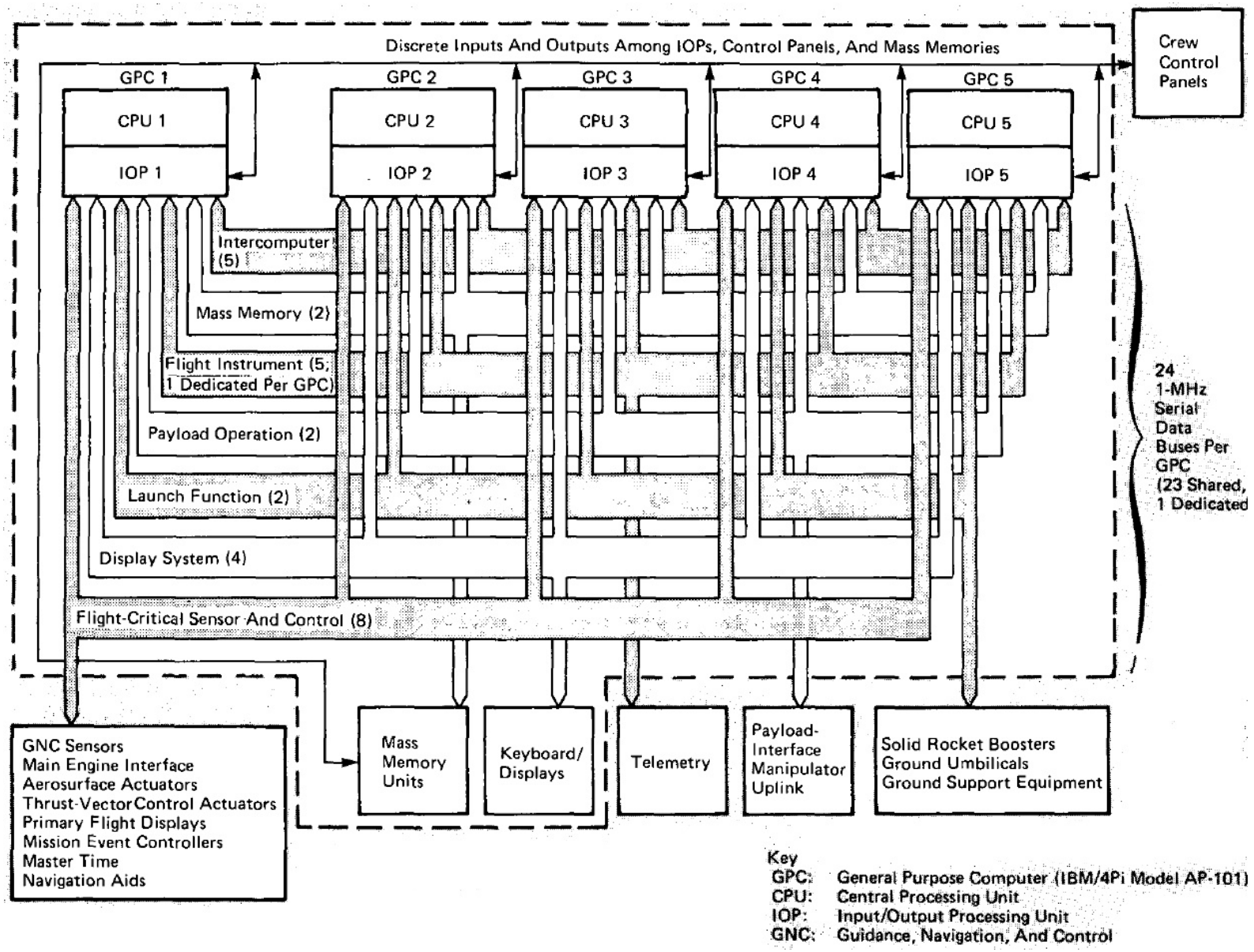
- Unregelmäßige und eher seltene Zustandsicherung

  → Potentiell großer Abstand der Redundanzen

  → Potentiell langwierige Fehlererholung

☞ Fokus: Replizierte Systeme im „hot standby“-Betrieb

Beispiel: Steuerung des Space Shuttle [2]

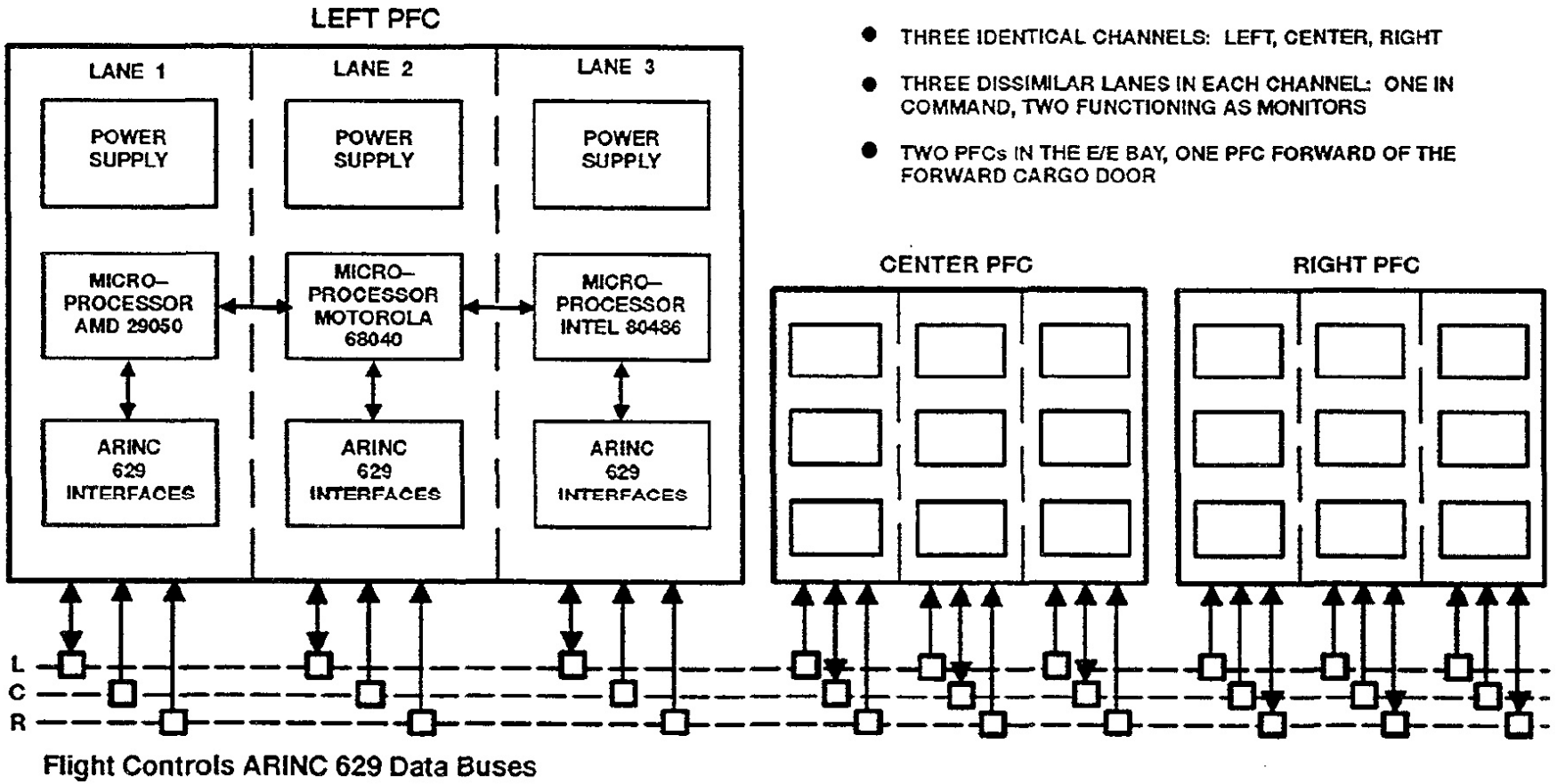


Beispiel: Steuerung des Space Shuttle (Forts.)

- Insgesamt fünf redundante Rechensysteme [1, Kapitel 4.4]
  - Ursprünglich gewünschte: fail-operational/fail-operational/fail-safe
    - Verlust eines Kontrollrechners ändert nichts an der Funktionsfähigkeit
    - Das Gesamtsystem behält immer noch die Eigenschaft fail-operational
  - Das war jedoch zu teuer $\rightsquigarrow$ Reduktion auf vier Systeme
    - Dies bedeutet fail-operational/fail-safe
    - Das fünfte System war aber bereits überall eingeplant

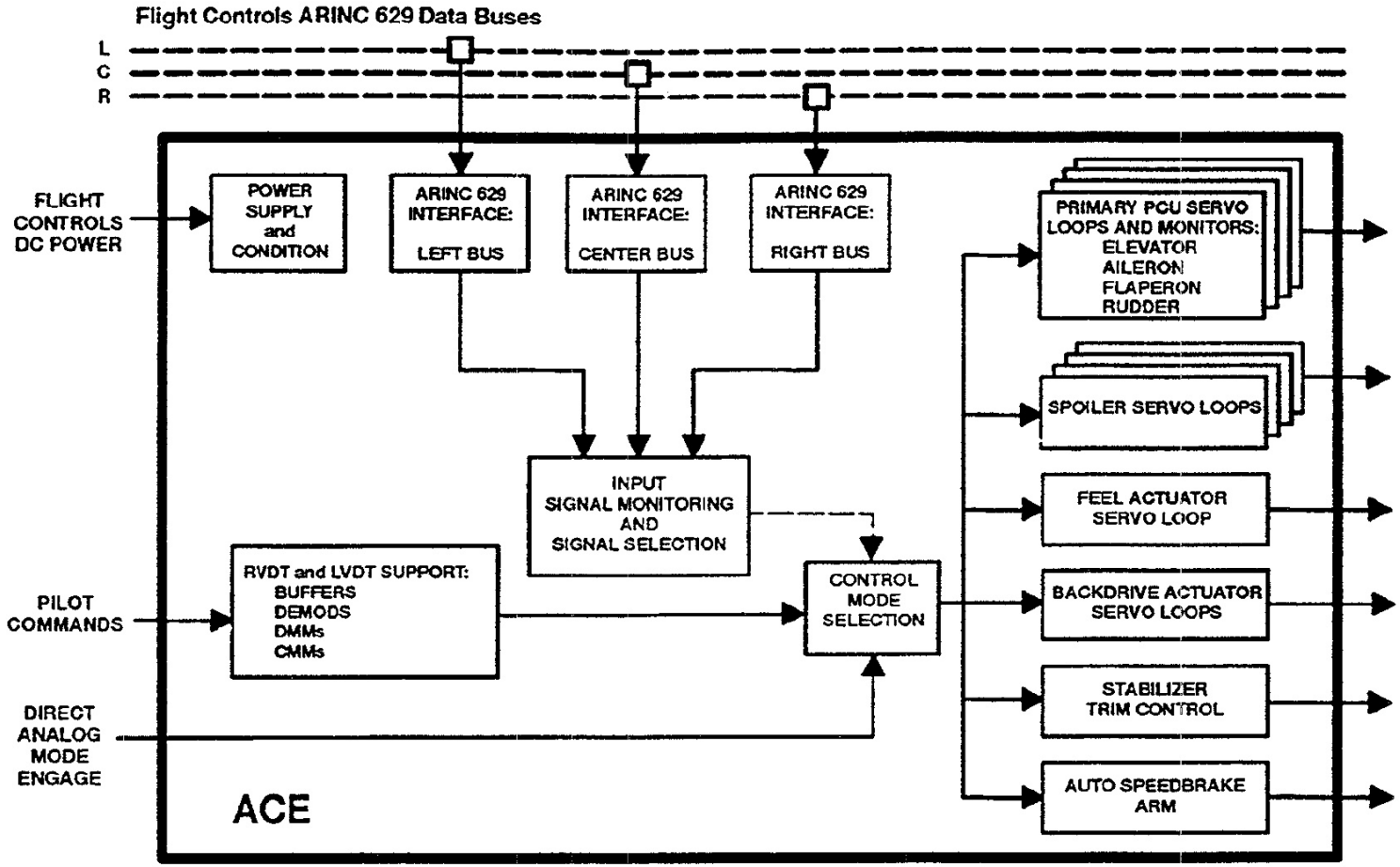
    $\rightsquigarrow$ Es wurde zu einem Backup-System „degradiert“ $\rightsquigarrow$ „cold standby“
- unterschiedliche Konfiguration der Rechner je nach Missionsabschnitt
  - TMR nur im Steigflug bzw. Sinkflug
    - Drei Systeme laufen simultan im „hot standby“-Betrieb
    - Das vierte System läuft im „warm standby“
    - Das fünfte System ist das Backup $\rightsquigarrow$ „cold standby“
  - Während des Shuttle in der Umlaufbahn ist, wird die Redundanz reduziert
    - Zwei System laufen weiterhin simultan
    - Das dritte System übernimmt Lebenserhaltungssysteme, . . .
    - Das vierte und fünfte Systeme sind Backup $\rightsquigarrow$ „cold standby“

Beispiel: Steuerung des Boeing 777 [8]



- Drei identische redundante Kanäle: links, mitte, rechts
  - Bestehend aus jeweils drei diversitären redundanten Pfaden
- Räumliche Verteilung innerhalb des Flugzeugs
  - Minimierung der Auswirkungen z. B. von Blitzschlägen

Beispiel: Steuerung des Boeing 777 [8] (Forts.)

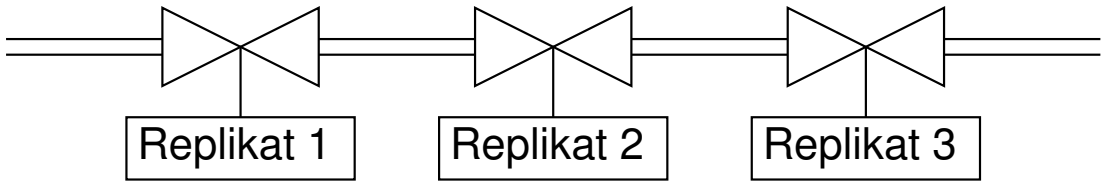


- Mehrheitsentscheid beim Aktor
  - ACE = actuator control electronics
  - Die Aktoren selbst sind ebenfalls redundant

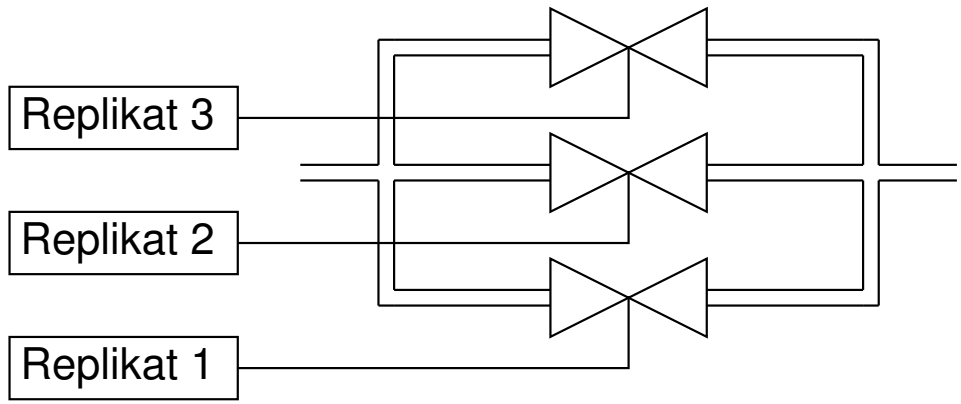
Mehrheitsentscheid am Aktor

Am Beispiel von Rohrleitungen und Ventilen

- Jedes Replikat kontrolliert jeweils ein Ventil
  - Vorgehensweise und Schaltfunktion ist hochgradig problemspezifisch
  - Auch anwendbar auf elektronische Schaltkreise und Relais
- Reihenschaltung von Absperrventilen



  - Um den Fluss zu stoppen, genügt ein korrektes Replikat
- Parallelschaltung von Absperrventilen



  - Um den Fluss zu ermöglichen, genügt ein korrektes Replikat

Hardwarebasierte Replikation – Vorteile und Nachteile

Vorteile

- Sehr hohe Zuverlässigkeit bei richtigem Einsatz

Nachteile

- Enorm hoher Hardwareaufwand
  - Ein Großteil der Hardwarekomponenten wird redundant ausgelegt
- Hiermit direkt verbunden sind
  - Hohe Kosten – viel Hardware kostet viel
  - Hohes Gewicht – viel Hardware wiegt viel
  - Hoher Energieverbrauch – viel Hardware benötigt viel Energie

Die höhere Integrationsdichte moderner Hardware könnte uns helfen

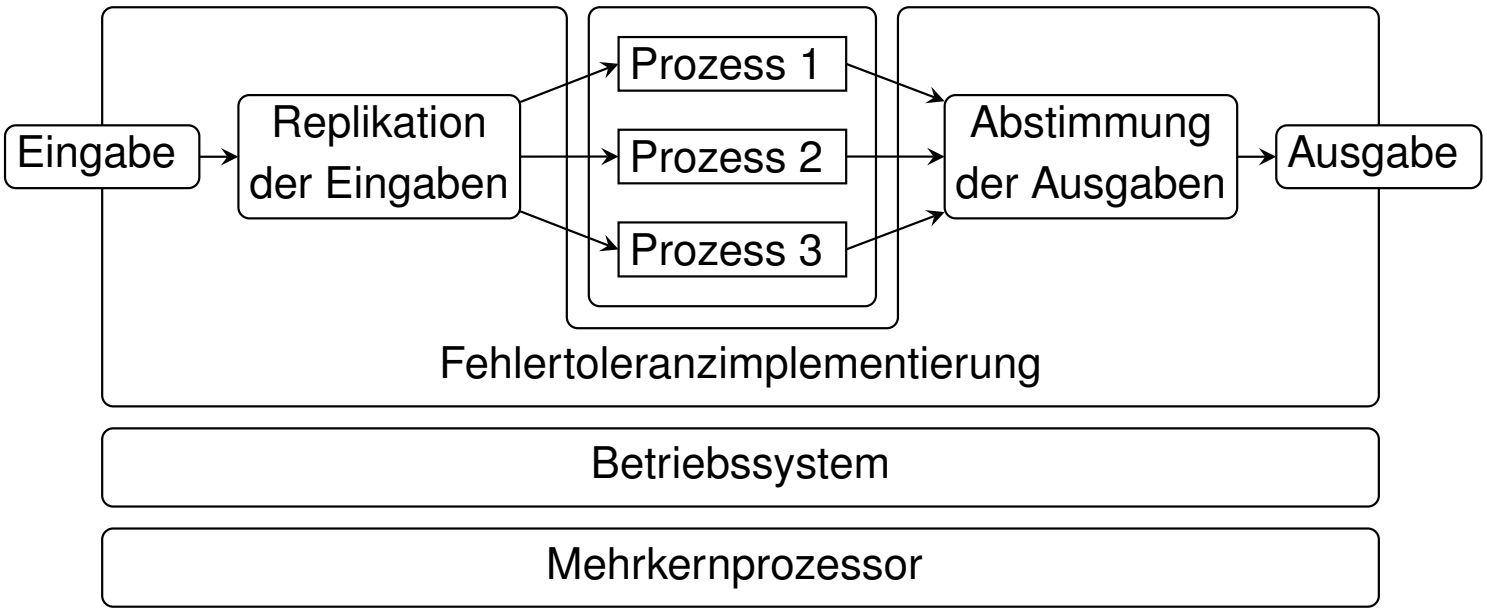
- Auch wenn sie andererseits höhere Fehlerraten bedingt

→ Mehrkernprozessoren „replizieren“ Rechenkerne

- Sie erlauben die Ausführung mehrerer Replikate auf demselben Prozessor

Process-Level Redundancy [6]

- Grundprinzip bleibt erhalten, nur der Inhalt der SoR ändert sich
  - Es werden keine kompletten Rechenknoten mehr repliziert
  - Sondern nur die Berechnung selbst, repräsentiert durch einen Prozess



  - Dedizierte Fehlertoleranzimplementierung sorgt für
    - Replikation der Eingaben und die Abstimmung der Ausgaben
    - Zeitliche Isolation der einzelnen Replikate
  - Basierend auf einem Echtzeitbetriebssystem
    - Das räumliche Isolation sichert und Mehrkernprozessoren unterstützt

Process-Level Redundancy [6] (Forts.)

- Funktionsweise der Fehlertoleranzimplementierung
  - Annahme: Replikate kommunizieren nach außen nur über Systemaufrufe
    - Diese Annahme ist für Prozesse unter Linux durchaus valide

☞ Emulation der Systemaufrufschnittstelle
  - Lesende Systemaufrufe $\rightsquigarrow$ Replikation der Eingabedaten
    - So findet automatisch eine Einigung über die Eingaben statt
  - Schreibende Systemaufrufe $\rightsquigarrow$ Ausgaben puffern & Mehrheitsentscheid
    - Nicht zurücknehmbare Seiteneffekte sind problematisch
    - Sie dürfen erst durchgeführt werden, wenn ihre Korrektheit gesichert ist
- Synchronisation der einzelnen Replikate
  - Zu ähnlichen Zeitpunkten werden identische Systemaufrufe getätigt
    - Sofern sich die einzelnen Replikate korrekt verhalten
  - Überwachung durch Ausgangsvergleich und durch Auszeiten
    - Die Fehlertoleranzimplementierung weiß, wann Systemaufrufe stattfinden

  → Replikdeterminismus
- Zeitliche Isolation durch Überwachung der Laufzeit
  - Überschreitung der Laufzeit führt z. B. zum Ablaufen einer Auszeit

Vergleich mit TMR

- Vorteil: Hardwareaufwand wurde deutlich reduziert
  - Nur ein Prozessor (mit mehreren Rechenkernen)
  - Kein gesondertes Kommunikationssystem zwischen den Replikaten
  - Damit sind direkt verbunden
    - Geringere Kosten, Gewicht, Energieverbrauch
- Nachteil: Der Grad an Redundanz nimmt unweigerlich ab
  - Fehler in gemeinsamen Teilen können zu Gleichtaktfehlern führen
    - Prozessorcaches, Stromversorgung, Kommunikationssystem

→ Kompromiss aus Kosten und Nutzen

Dennoch: Technologie der Zukunft

- Mehrkernprozessoren sind unaufhaltsam auf dem Vormarsch
  - Erste dedizierte Mehrkernprozessoren im Automobilbereich
- Gleichzeitig: einzelne Rechenkerne sind nicht mehr sicher genug
  - Transiente Fehlerrate macht Redundanz unvermeidbar

Gliederung

Beispiel: Ariane 5 (vgl. Folien II/16 ff)

- Beide Inertialmesssysteme SRI1 und SRI2 fallen gleichzeitig aus
  - Ein Ganzzahlüberlauf wegen einer Eingabe außerhalb der Spezifikation

  → Die Bordcomputer OBC1 und OBC2 interpretieren den Fehlerwert falsch

  → Fehlerhaftes Lenkmanöver führt zur Zerstörung der Rakete

⚠ Ursache war ein Gleichtaktfehler in homogenen Redundanzen
  - Softwaredefekte sind typische Quellen für Gleichtaktfehler
  - Wie geht man mit Softwaredefekten um?

  → Wende Redundanz bei der Entwicklung solcher Systeme an!

☞ Diversität (engl. *diversity*) ⇝ heterogene Redundanzen
  - Auch N-version programming, mehr dazu siehe z. B. [4, Kapitel 6.6]
  - Man nehme „mehrere verschiedene von allem“
    - Entwicklungsteams, Programmiersprachen, Übersetzer, Hardwareplattformen
    - Alle entwickeln dasselbe System in mehreren Ausführungen
  - Annahme: Die Ergebnisse sind für sich wahrscheinlich nicht fehlerfrei

    → Aber sie enthalten wahrscheinlich auch nicht dieselben Fehler

    → Gleichtaktfehler dürften hier nicht mehr auftreten

Diversität ist sehr umstritten!

Problem: Diese Annahme stimmt nicht unbedingt

- Gleichtaktfehler verursachende Defekte rühren oft aus der Spezifikation

→ Diese betrifft alle diversitären Entwicklungsvorhaben gleichermaßen

  - Was auch auf die Ariane 5 zugetroffen hätte . . .

Verwende verschiedene Spezifikationen als Ausgangspunkt

- Wie bekommt man dann die „verschiedenen“ Ausgaben unter einen Hut?
- Dies erfordert komplexe Verfahren beim Mehrheitsentscheid
  - Exakte Mehrheitsentscheide (engl. *exact voting*) sind vergleichsweise trivial
  - Unscharfe Mehrheitsentscheide (engl. *non-exact voting*) sind aus heutiger Sicht hingegen nicht besonders vielversprechend . . .

Diversität findet dennoch erfolgreich Anwendung (s. Folie 27)

- z. B. in asymmetrisch redundanten Systemen
  - Eine komplexe Berechnung wird durch eine einfache Komponente kontrolliert
  - Gepaart mit fail-safe-Verhalten im Fehlerfall
  - Was bei Eisenbahnsignalanlagen sehr gut funktioniert
- z. B. in der Reaktornotabschaltung vieler Kernkraftwerke
  - Der Mehrheitsentscheid funktioniert nach dem Schema auf Folie 29

Zusammenfassung

Redundanz $\mapsto$ hat mehrere Dimensionen

- Grundvoraussetzung für Fehlertoleranz
- Redundanz in Struktur, Funktion, Information, oder Zeit
- Fehlererkennung, -diagnose, -eindämmung, -maskierung

Replikation $\mapsto$ koordinierter Einsatz struktureller Redundanz

- Replikation der Eingaben, Abstimmung der Ausgaben
- Replikate für fail-silent, fail-consistent, malicious
- Zeitliche und räumliche Isolation einzelner Replikate

Hardwarebasierte Replikation $\mapsto$ Umfassend und teuer

- Dreifache Auslegung, toleriert Fehler im Wertbereich
- Zuverlässigkeit von Replikat und Gesamtsystem

Softwarebasierte Replikation $\mapsto$ Flexibel aber eingeschränkt

- Process Level Redundancy reduziert Kosten von TMR, zulasten eines geringeren Schutzes

Diversität $\mapsto$ versucht Gleichtaktfehler auszuschließen

Literaturverzeichnis

[1] *Computers in Spaceflight: The NASA Experience*.
http://history.nasa.gov/computers/contents.html, Apr. 1987

[2] Carlow, G. D.:
Architecture of the space shuttle primary avionics software system.
In: *Communications of the ACM* 27 (1984), Nr. 9, S. 926–936.
http://dx.doi.org/10.1145/358234.358258. –
DOI 10.1145/358234.358258. –
ISSN 0001–0782

[3] Echtle, K. :
Fehlertoleranzverfahren.
Berlin, Germany : Informatik Springer, 1990. –
ISBN ISBN 978–0–3875–2680–5

[4] Kopetz, H. :
Real-Time Systems: Design Principles for Distributed Embedded Applications.
First Edition.
Kluwer Academic Publishers, 1997. –
ISBN 0–7923–9894–7

[5] Mukherjee, S. :
Architecture Design for Soft Errors.
San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2008. –
ISBN 978–0–12–369529–1

Literaturverzeichnis (Forts.)

[6] Shye, A. ; Moseley, T. ; Reddi, V. J. ; Blomstedt, J. ; Connors, D. A.:
Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance.
In: *Proceedings of the 37th International Conference on Dependable Systems and Networks (DSN '07)*.
Washington, DC, USA : IEEE Computer Society Press, Jun. 2007. –
ISBN 0–7695–2855–4, S. 297–306

[7] Ulbrich, P. :
Echtzeitsysteme.
`http://www4.cs.fau.de/Lehre/WS16/V_EZS/`, 2016

[8] Yeh, Y. :
Triple-triple redundant 777 primary flight computer.
In: *Proceedings of the 1996 IEEE Aerospace Applications Conference*.
Washington, DC, USA : IEEE Computer Society Press, Febr. 1996. –
ISBN 978–0780331969, S. 293–307