

Verlässliche Echtzeitsysteme

Redundante Ausführung

Tobias Klaus

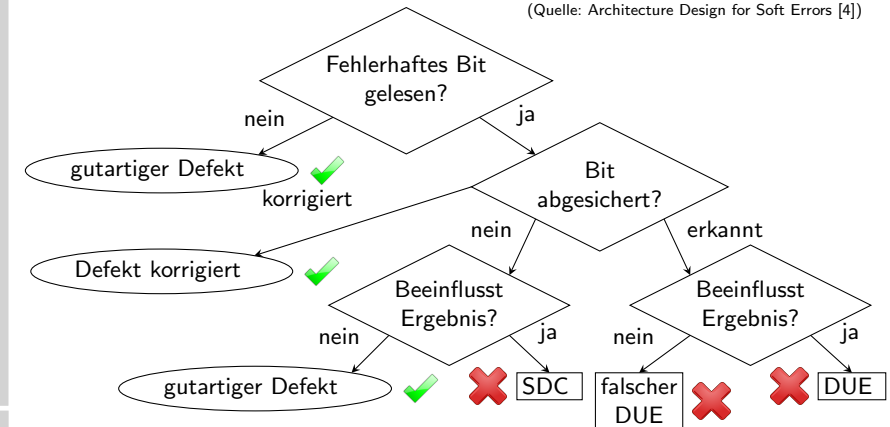
Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.informatik.uni-erlangen.de

19. Mai 2015

Wiederholung – Fehlerarten

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

(Quelle: Architecture Design for Soft Errors [4])



© fs, pu (FAU/INF4) Verlässliche Echtzeitsysteme (SS 2015) – Kapitel VIII Redundante Ausführung
1 Überblick

2/39

Problem – Fehlervermeidung

- **Ungehemmte Fehlerfortpflanzung** führt zum Systemversagen
 - Unerkannte Datenfehler (engl. *silent data corruption*)
 - 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 je nach Anwendung erforderlich
 - **Problematisches**: eine entsprechend robuste Auslegung einzelner Komponenten ist häufig nicht möglich
 - Diese Komponente müsste frei von konzeptionellen Fehler sein, also keinerlei Hardware- oder Softwaredefekte etc. enthalten
 - Sie müsste auch allerlei widrigen äußeren Umständen trotzen
 - **Lösung**: man benötigt ein System, das einzelne **Fehler tolerieren kann**
 - Einzelne Komponenten können ausfallen ...
 - Dies wird durch andere **redundante Komponenten** aufgefangen,
 - \leadsto Die gewünschte Funktionalität an der Schnittstelle bleibt erhalten
 - Der Anwender bekommt davon möglichst nichts mit (\leadsto **Transparenz**)

Redundanz und Fehlertoleranz – Übersicht

Was kann man gegen eine unzuverlässige Hardware tun?

- Maskierung **transienter Hardwarefehler** durch **redundante Ausführung**
 - **Grundlegender Aufbau** replizierter Systeme
 - Auf **welcher Ebene** wird Redundanz angewandt?
 - Welche Eigenschaften müssen die einzelnen **Replikate** erfüllen?
- **Hardwarebasierte Replikation**
 - Die **klassische Lösung** für die Auslegung fehlertoleranter Systeme
 - Replikation auf **Ebene des Knotens bzw. der Hardware**
 - Fokussierung auf **Triple Modular Redundancy**
 - Prinzipiell sind n-fach redundante Systeme denkbar, $n = 2, \dots$
- **Softwarebasierte Replikation**
 - Process-Level Redundancy: Zuhilfenahme von **Mehrkernprozessoren**
 - Replikation auf **Ebene von Prozessen bzw. Software**
- Vermeidung von Gleichtaktfehlern durch **Diversität**
 - „replizierte Entwicklung“ der einzelnen Redundanzen

Gliederung

- 1 Überblick
- 2 Grundlagen
 - Redundanz
 - Replikation
 - Fehlerhypothese
- 3 Hardwarebasierte Replikation
- 4 Softwarebasierte Replikation
- 5 Diversität
- 6 Zusammenfassung



Arten von Redundanz

- Redundanz ist eine Grundvoraussetzung für Fehlertoleranz
- strukturelle Redundanz
 - Replikation \leadsto hardwarebasierte Fehlertoleranzlösungen (typisch)
 - Mehrfache Auslegung: Prozessoren, Speicher, Sensoren, Aktoren, ...
 - Gleichartige Instanzen, agieren häufig **simultan**
- funktionelle Redundanz
 - Mehrfache Herleitung desselben Sachverhalt auf verschiedenen Wegen
 - Ventilstellung \leadsto Stellungsgeber bzw. Durchflussmengenmesser
 - Funktionswächter (engl. *watchdog*) für bestimmte Parameter
- Informationsredundanz
 - Zusätzliche Informationen (nicht zwingend erforderlich)
 - Speicherung von **Brutto-** und **Nettobetrag**
 - Typischerweise in Form von **Codierung** (Prüfsummen, CRC, ...)
- zeitliche Redundanz
 - über den Normalbetrieb hinausgehende Zeit
 - Z.B. Numerische Algorithmen, Schlupf in einem EZS, ...



Ziel der Redundanz

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*)

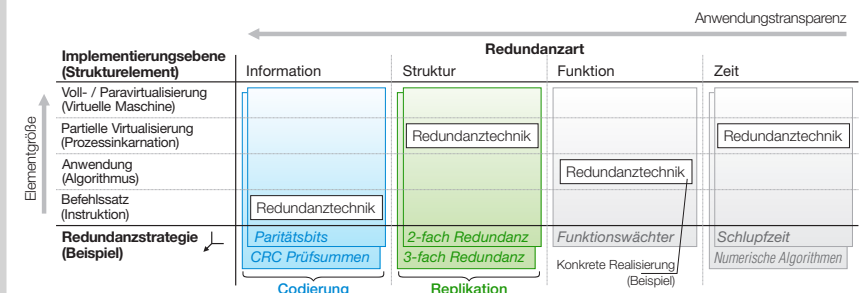
- Wiederherstellen eines funktionsfähigen Zustands nach Fehlern
 - Reparatur (engl. *repair*) bzw. Rekonfiguration (engl. *reconfiguration*)

Fokus der Vorlesung: Fehlererkennung und Fehlermaskierung



Koordinierter Einsatz von Redundanz

Ein zweites Rechensystem nützt alleine nicht viel!



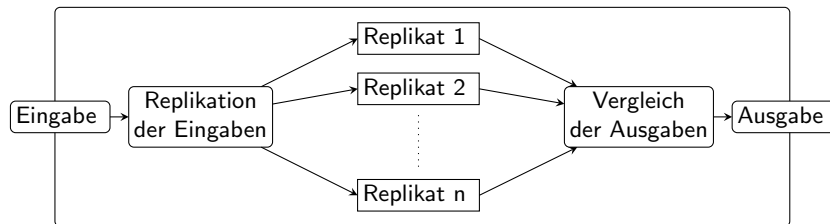
- Es gibt viele Implementierungsalternativen
- In der Praxis: **Mischformen** \leadsto was ermöglicht Fehlererkennung?
- Funktionelle und zeitliche Redundanz erfordern umfangreiches Vorwissen

Fokus der Vorlesung: strukturelle und Informationsredundanz



Replikation

- Replikation ist der **koordinierte Einsatz struktureller Redundanz**
Anordnung in einer „**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?
 - Wie behebt man verbliebene **kritische Bruchstellen**?
 - Was passiert bei Fehler in der Eingabe oder im Ausgangsvergleich?

Wie viele Replikate braucht man?

Allgemeiner Fall mit unabhängigen Replikaten (Knoten)

- Zahl benötigter Replikate hängt von der Art des Fehlverhaltens ab [3]
 - 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 Fehlverhalten führt zum Stillstand
 - \leadsto **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*)

Wie viele Replikate braucht man? (Forts.)

- **Vorabwissen und Systemaufbau** helfen Replikate einzusparen
Hohe **Fehlererkennungsrate** (engl. *error detection coverage*)
 - Das Fehlverhalten wird **erkannt**
 - Innerhalb des Replikats oder durch zentrale Prüfinstanz
 - Ein Ausbrechen des Fehler ist **nicht tolerierbar**
 - Umfasst sowohl Fehlverhalten im **Wertebereich** ...
 - Falsche Eingabewerte oder Berechnungsergebnisse können beispielsweise durch Zusicherungen abgefangen werden
 - Durchführung häufig im Rahmen von **Akzeptanztests**
 - ... als auch Fehlverhalten im **Zeitbereich**
 - Maximale Antwortzeit der Replikate ist bekannt und durchsetzbar
 - „**quasselnde Idioten**“ (engl. *babbling idiot*) überlasten das Kommunikationssystem durch zeitlich unkoordinierten Nachrichtenversand
- \leadsto Das korrekte Systemverhalten ist a-priori bekannt
 - Und kann genutzt werden, um „fail-silent“-Verhalten zu implementieren
 - **Zwei Replikate** reichen in diesem Fall aus, um einen Fehler zu tolerieren
- **Sonst:** Mehrheitsentscheid liefert das korrekte Verhalten
- \leadsto hierfür benötigt man dann ein **drittes Replikat**

Zustand redundanter Systeme

Wie verhalten sich die redundanten Systeme zueinander?

hot standby: Redundante Systeme arbeiten **simultan**

- Sie verarbeiten gleichzeitig dieselben Eingaben
- Ihr Zustand ist **jederzeit konsistent**
 - \leadsto **nahtloser Ersatz** für ausgefallene Redundanzen

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
 - \leadsto Höherer Aufwand im Falle der Fehlererholung

cold standby: Sekundärsystem startet im Fehlerfall

- **Unregelmäßige und eher seltene Zustandsicherung**
 - \leadsto Potentiell **großer Abstand der Redundanzen**
 - \leadsto Potentiell **langwierige Fehlererholung**

■ **Fokus:** redundante Systeme im „hot standby“-Betrieb

Fehlerisolation

- Replikate fallen **unabhängig** voneinander aus
 - **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** sind z. B. ...
 - **Softwaredefekte** und ...
 - ... das **Übergreifen eines Fehlers** auf andere Replikate

☞ einzelne Replikate sind **gegeneinander abzuschotten**

- Ein Dienst, den die SoR zur Verfügung stellt

räumliche Isolation des internen Zustands

- Dieser darf nicht durch andere Replikate korumpiert werden
 - Ein verfälschter Zeiger hat großes Schadenspotential

zeitliche Isolation anderer Aktivitätsträger

- Eine Monopolisierung der CPU 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 **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 Sortiervverfahren, ...



Fehlerhypothese (engl. *fault hypothesis*)

Annahmen über das Verhalten einzelner Replikate im Fehlerfall

- In der Praxis betrachtet man für Echtzeitsysteme Replikate, die:
 - **Einen transienten Fehler** tolerieren können
 - Sich „fail-silent“ oder zumindest „fail-consistent“ verhalten
 - **Unabhängig voneinander** ausfallen
 - Gleichtaktfehler müssen also ausgeschlossen werden
 - Sich **replikdeterministisch** verhalten
 - Ermöglicht eine einfache Umsetzung des Mehrheitsentscheids
- 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
 - $f + 1$ Kommunikationsrunden 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**



Gliederung

1 Überblick

2 Grundlagen

- Redundanz
- Replikation
- Fehlerhypothese

3 Hardwarebasierte Replikation

4 Softwarebasierte Replikation

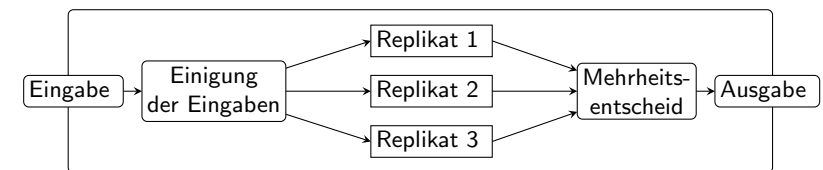
5 Diversität

6 Zusammenfassung



Triple Modular Redundancy (TMR)

- Falls **Fehler im Wertebereich** nicht zu verhindern sind



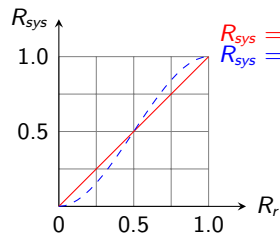
- üblicherweise **dreifache Replikation kompletter Rechenknoten**
 - **Räumlich redundante** Systeme im „hot standby“-Betrieb
 - ↪ Weitgehende räumliche und zeitliche Isolation
- **Abstimmung der Eingabewerte** zwischen den Replikaten
 - Die Replikate verfügen über eine gemeinsame globale Zeitbasis
 - Das Kommunikationssystem verhindert die Steuerfehlerausbreitung
 - ↪ Vollständige zeitliche Isolation [5, Kapitel 8] und Replikdeterminismus
- **Mehrheitsentscheid** (engl. *voter*) stimmt Ausgabewerte ab
 - Vereinigung von **Fehlermaskierung und -erkennung**



Wann hat TMR einen Nutzen?

Hilft viel grundsätzlich viel?

- Erhöht sich durch TMR in jedem Fall die Zuverlässigkeit?
 - Anders formuliert: $R_{tmr} > R_r$?
 - R_{tmr} – Zuverlässigkeit des TMR-Verbunds, R_r des einzelnen Replikats
 - der TMR-Verbund arbeitet korrekt, solange ...
 - Der Mehrheitsentscheid korrekt funktioniert $\leadsto R_v$
 - Zwei Replikate korrekt funktionieren $\leadsto R_{2/3} = R_r^3 + 3R_r^2(1 - R_r)$
 - Alle drei Replikate arbeiten korrekt oder ...
 - Ein Replikat fällt aus, hierfür gibt es drei Möglichkeiten
- \leadsto Insgesamt $R_{tmr} = R_v(R_r^3 + 3R_r^2(1 - R_r)) > R_r$?



- 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 also die fehlertolerierende Eigenschaft der SoR
- \leadsto In der SoR auf Folie VIII/9 sind dies **Eingabe und Ausgabe**



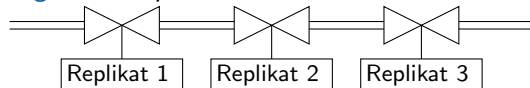
Lösungsmöglichkeiten

- Bestimme Eingabedaten aus **mehreren Sensoren**
 - Dies erfordert eine **Einigung der Replikate** über den Eingabewert, allen muss exakt derselbe Wert zugestellt werden
 - Anwendung funktionaler Redundanz \leadsto **Sensorfusion** (engl. *sensor fusion*)
- **Repliziere den Ausgangsvergleich**
 - Erneuter Mehrheitsentscheid über die Ergebnisse des replizierten Vergleichs
 - \leadsto Das ist wieder eine kritische Bruchstelle, aber die **Fehlerwahrscheinlichkeit sind insgesamt geringer, verschwinden tut sie nie ...**
- **Robuste Implementierung** des Ausgangsvergleichs
 - Zusätzliche Absicherung des Ergebnisses durch z. B. **arithmetische Signaturen**
 - Durchführung des Mehrheitsentscheids durch den **Aktor**

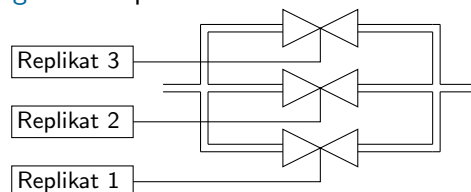
Mehrheitsentscheid am Aktor

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

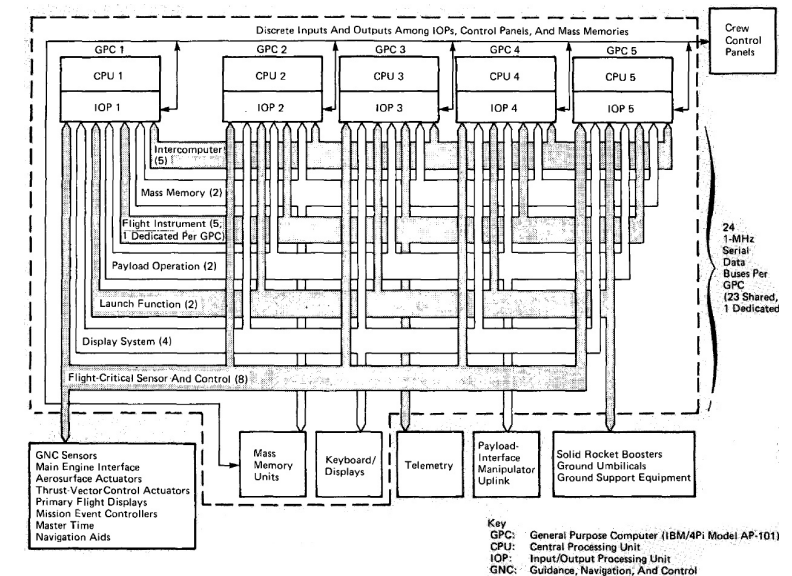


- Um den Fluss zu stoppen, genügt ein korrektes Replikat
- **Parallelschaltung** von Absperrventilen



- Um den Fluss zu ermöglichen, genügt ein korrektes Replikat

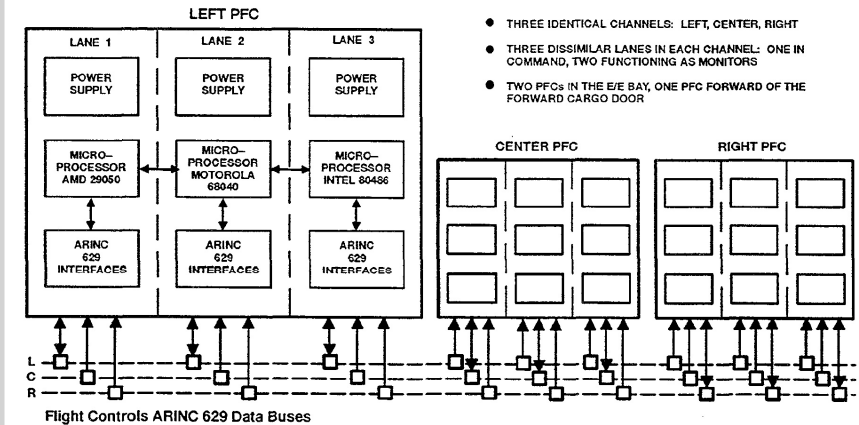
Beispiel: Steuerung des Space Shuttle [2]



Beispiel: Steuerung des Space Shuttle (Forts.)

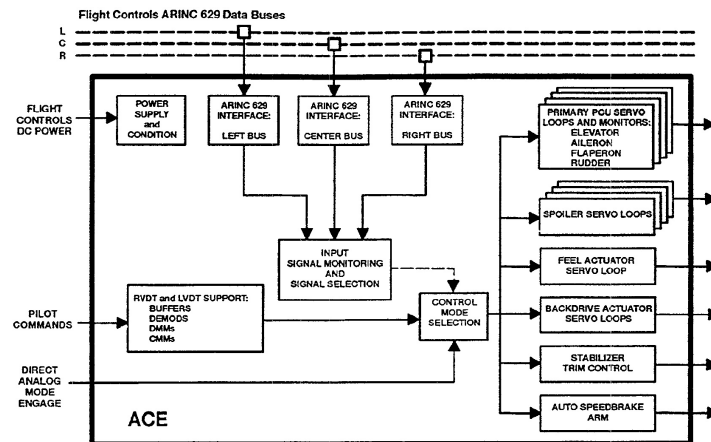
- Insgesamt **fünf redundante Rechensysteme** [1, Kapitel 4.4]
 - Ursprünglich gewünschte: **fail-operational/fail-operational/fail-safe**
 - Verlust eines Kontrollrechners ändert nichts an der Funktionsfähigkeit
 - Das Gesamtsystem behält immer noch die Eigenschaft **fail-operational**
 - Das war jedoch **zu teuer** → Reduktion auf vier Systeme
 - Dies bedeutet **fail-operational/fail-safe**
 - Das fünfte System war aber bereits überall eingeplant
 - Es wurde zu einem Backup-System „degradiert“ → „**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 → „**cold standby**“
 - Während des Shuttle in der **Umlaufbahn** ist, wird die Redundanz reduziert
 - **Zwei Systeme** laufen weiterhin simultan
 - Das dritte System übernimmt Lebenserhaltungssysteme, ...
 - Das vierte und fünfte Systeme sind Backup → „**cold standby**“

Beispiel: Steuerung des Boeing 777 [7]



- **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 Blitzeinschlägen

Beispiel: Steuerung des Boeing 777 [7] (Forts.)



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

Gliederung

- 1 Überblick
- 2 Grundlagen
 - Redundanz
 - Replikation
 - Fehlerhypothese
- 3 Hardwarebasierte Replikation
- 4 Softwarebasierte Replikation
- 5 Diversität
- 6 Zusammenfassung

Vorteile und Nachteile von TMR

Vorteile von TMR

- Sehr hohe Zuverlässigkeit bei richtigem Einsatz

Nachteile von TMR

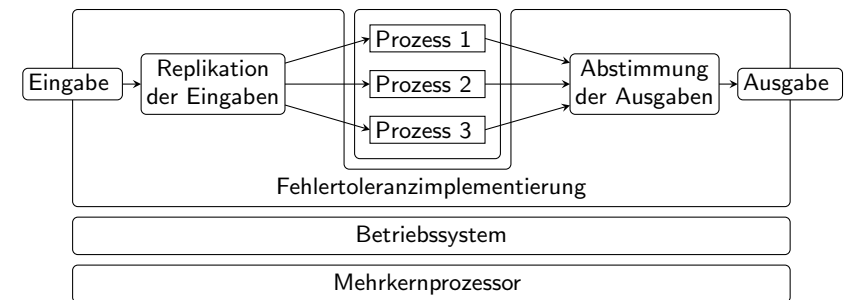
- Enorm hoher Hardwareaufwand
 - Ein Großteil der Hardwarekomponenten wird redundant ausgelegt
- Hiermit direkt verbunden sind
 - Hohe Kosten – viel Hardware kostet viel
 - Hohes Gewicht – viel Hardware wiegt viel
 - Hoher Energieverbrauch – viel Hardware benötigt viel Energie

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



- eine dedizierte Fehlertoleranzimplementierung sorgt für
 - Die Replikation der Eingaben und die Abstimmung der Ausgaben
 - Und die zeitliche Isolation der einzelnen Replikate
- hierfür greift sie auf ein Betriebssystem zurück
 - 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 Systemaufrufsschnittstelle

- Lesende Systemaufrufe → Replikation der Eingabedaten
 - So findet automatisch eine Einigung über die Eingaben statt
- Schreibende Systemaufrufe → 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 Ablauf 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

- 1 Überblick
- 2 Grundlagen
 - Redundanz
 - Replikation
 - Fehlerhypothese
- 3 Hardwarebasierte Replikation
- 4 Softwarebasierte Replikation
- 5 Diversität
- 6 Zusammenfassung



Beispiel: Ariane 5

- 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. [3, 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
 - **Unschärfe Mehrheitsentscheide** (engl. *non-exact voting*) sind aus heutiger Sicht hingegen nicht besonders vielversprechend ...
- Diversität findet dennoch erfolgreich Anwendung (s. Folie 26)
 - 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 23



Gliederung

- 1 Überblick
- 2 Grundlagen
 - Redundanz
 - Replikation
 - Fehlerhypothese
- 3 Hardwarebasierte Replikation
- 4 Softwarebasierte Replikation
- 5 Diversität
- 6 Zusammenfassung



Zusammenfassung

Fehlertypen \mapsto Toleranz von SDCs und DUEs

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] KOPETZ, H. :
Real-Time Systems: Design Principles for Distributed Embedded Applications.
Kluwer Academic Publishers, 1997. –
ISBN 0–7923–9894–7
- [4] MUKHERJEE, S. :
Architecture Design for Soft Errors.
San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2008. –
ISBN 978–0–12–369529–1
- [5] SCHELER, F. :
Echtzeitsysteme.
http://www4.cs.fau.de/Lehre/WS11/V_EZS/, 2011



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] 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

