

Verlässliche Echtzeitsysteme

Reintegration fehlgeschlagener Knoten

Fabian Scheler

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)

www4.informatik.uni-erlangen.de

17. Juni 2013

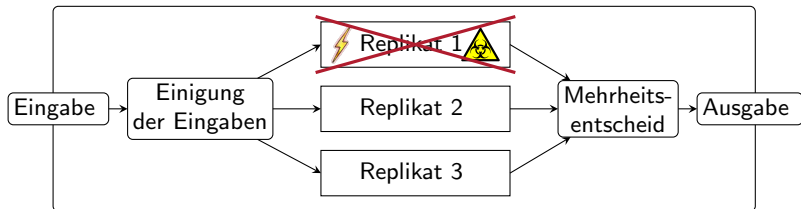


- Was passiert eigentlich **nach** einem transienten Fehler?
 - Welchen Einfluss hat der Fehler auf das betroffene Replikat?
 - Was bedeutet dies für die Fähigkeit des Systems Fehler zu tolerieren?
- Grundlagen der **Reintegration fehlgeschlagener Knoten**
 - Welche Varianten gibt es hier?
 - Und welche hiervon eignen sich für Echtzeitsysteme?
 - Wie kann man ein Replikat von einem Fehler erholen?
- zwei Varianten der Fehlererholung
 - Eigenständige Fehlererholung durch „**Recovery Blocks**“
 - der Fehler wird durch das Replikat selbst korrigiert
 - **Zustandstransfer** von einem funktionsfähigen Replikat
 - um eine korruptierte Datenbasis zu reparieren



- 1 Überblick
- 2 Grundlagen**
 - Vorgehen bei der Reintegration
 - Fehlererholung
 - Interner Zustand
- 3 Distributed Recovery Blocks
- 4 Zustandstransfer
- 5 Zusammenfassung





Normalbetrieb alle Replikate arbeiten

- Redundanz und Fehlertoleranz sind in vollem Umfang gegeben

Fehlerfall ein Replikat erleidet einen **transienten Fehler**

~> der interne Zustand wird dadurch korrumpiert

~> **das Replikat fällt aus ...**



dies bedeutet eine **Reduktion der Fähigkeit Fehler zu tolerieren**

- nur noch zwei funktionsfähige Replikate ~> das System ist **verwundbar**

~> Replikat 1 muss sich **erholen** und **reintegriert** werden

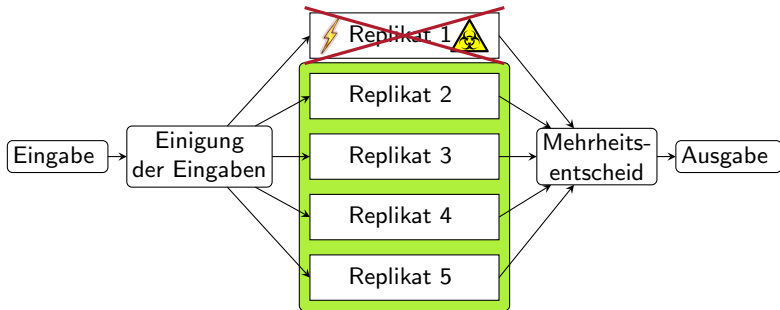


Reintegration fehlgeschlagener Knoten

Wie geht man grundsätzlich mit dem Ausfall eines Knotens um?

- 1 Fehlererkennung** \mapsto Ist ein Fehler aufgetreten?
 - zu Beginn steht die Erkennung, dass ein Replikat fehlerhaft ist
 - dies kann durch den Mehrheitsentscheid erfolgen, oder
 - das fehlerhafte Replikat teilt dies selbst mit (s. Folie VIII/11, **crash failure**)
- 2 Fehlerdiagnose** \mapsto Wo ist der Fehler aufgetreten?
 - Welches Replikat ist ausgefallen?
- 3 Rekonfiguration** \mapsto Das fehlerhafte Replikat „aussperren“.
 - das fehlerhafte Replikat darf nicht mehr am Betrieb teilnehmen
 - ggf. wird das Replikat durch ein Backup-System ersetzt
 - so wird die volle Fähigkeit Fehler zu tolerieren schnell wiederhergestellt
- 4 Fehlererholung** \mapsto Den Fehler beseitigen.
 - Kann der Fehler behoben werden, oder liegt ein permanenter Fehler vor?
 - Umsetzung durch einen Selbsttest, den das Replikat durchführt.
- 5 Reintegration** \mapsto Den Ursprungszustand wiederherstellen.
 - das Replikat wird wieder in den Verbund aufgenommen





- Replikate $\{1,2,3,4\}$ arbeiten in einem 4-fach redundanten Verbund
 - Replikat 5 ist das Backup-System
- ☞ Replikat 1 fällt nun aufgrund eines transienten Fehlers aus
 - zunächst wird Replikat 1 aus dem Verbund ausgeschlossen
 - dann wird Replikat 5 in den Verbund aufgenommen
 - um die Fähigkeit zur Fehlertoleranz wieder vollkommen herzustellen
 - schließlich bleibt noch Replikat 1 zu erholen und neu zu integrieren



reaktive Fehlererholung \leadsto nachdem der Fehler aufgetreten ist

- setzt die Erkennung des Fehlers voraus
- anschließend wird die Fehlersituation repariert
 - ein **Wiederholungsversuch** (engl. *retry*) wird unternommen
 - es wird zu einem **Sicherungspunkt** (engl. *checkpoint*) zurückgekehrt

proaktive/reaktive Fehlererholung \leadsto Vorbereitungen treffen

- proaktive Maßnahmen bereiten eine reaktive Fehlererholung vor
 - das regelmäßige Erstellen von Sicherungspunkten ist proaktiv
 - deren Wiederherstellung im Fehlerfall hingegen reaktiv

proaktive Fehlererholung \leadsto Vermeidung von Fehlersituationen

- vorsorglich durchgeführte Maßnahmen, die
 - regelmäßig zu festen Zeitpunkten, oder
 - als Reaktion auf die Veränderung bestimmter Indikatoren stattfinden
 - wenn ein Leistungsabfall (z. B. mittleren Antwortzeit) beobachtbar ist
- sie umfassen z. B. regelmäßige Neustarts, leeren von Puffern, ...



Rückwärtsbewegung (engl. *roll backward, backward recovery*)

- Rückkehr zu einem **Sicherungspunkt** im Fehlerfall
 - ↪ der Sicherungspunkt muss entsprechend vertrauenswürdig sein
- Bearbeitung muss komplett **umkehrbar** sein
 - ↪ keine **nicht-zurücknehmbaren Aktionen** (engl. *irrevocable actions*)
 - ↪ ansonsten wäre eine Rückkehr zum Sicherungspunkt unmöglich
- **Dauer der Fehlererholung** hängt von folgenden Faktoren ab
 - **Größe bzw. Umfang** des Sicherungspunkts
 - Wie viele Daten müssen kopiert werden? Wie lange dauert dies?
 - **Alter** des Sicherungspunkts
 - evtl. müssen verpasste Anfragen nachgeholt werden (engl. *audit trail*)
- ↪ für Echtzeitsysteme häufig **zu langwierig**

Vorwärtsbewegung (engl. *roll forward, forward recovery*)

- kommt **ohne Rückgriff auf vorher gesicherten Zustand** aus
 - ↪ die **bevorzugte Variante für Echtzeitsysteme**
 - Zustand wird in jedem Zyklus neu erfasst ↪ Maskierung alter Werte
 - **kurze Fehlererholung** ↪ im Idealfall durch den Normalbetrieb



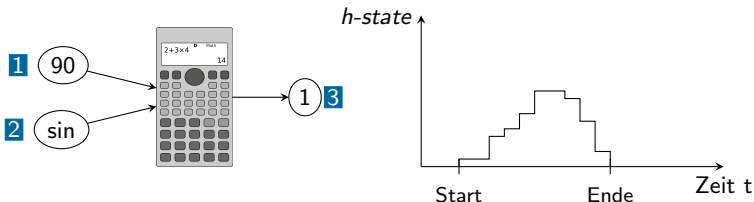


- zwei Teile sind maßgeblich für den Zustand eines Replikats
 - Initialzustand (engl. *initialization state*, *i-state*)
 - statische, zur Laufzeit unveränderliche Datenstruktur
 - umfasst z. B. den Programmcode und Initialisierungsdaten
 - auch zur Laufzeit konstante Daten zählen hierzu
 - ↪ Ablage im nicht-schreibbaren Speicher (engl. *read-only memory*, *ROM*)
 - dynamischer Zustand (engl. *history state*, *h-state*)
 - zur Laufzeit veränderliche Daten
 - beinhalten Informationen zum Fortschritt einer Berechnung
 - ↪ ist bei der Reintegration ausgefallener Replikate zu restaurieren
 - ↪ Ablage im schreibbaren Speicher (engl. *random-access memory*, *RAM*)



Beispiel: Dynamischer Zustand eines Taschenrechners

- Verwendung eines Taschenrechners
 - Eingabe der Operanden und Operatoren \leadsto Ergebnis
 - der dynamische Zustand ist zu Beginn und am Ende der Berechnung leer
 - immer, wenn man Berechnungen als **unteilbar** (engl. *atomic*) betrachtet



- angenommen die Sinus-Funktion wird iterativ approximiert
 - \leadsto der Taschenrechner verwaltet Zustandsvariablen für diese Berechnung
 - \leadsto diese Zustandsvariablen sind für die Berechnung unersetzlich
 - ihr Transfer ist erforderlich, soll die Berechnung anderswo fortgesetzt werden
- weiteres Beispiel: Summation mehrerer Summanden
 - Zwischenergebnisse \leadsto dynamischer Zustand der Summation



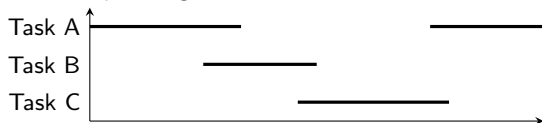
Minimierung des dynamischen Zustands

- der dynamische Zustand einer Berechnung ist für ihre Fortführung auf einem anderen Replikat zwingend erforderlich
 - die Reintegration eines Replikats erfordert seine Restauration
 - was letztendlich zu einem Zustandstransfer zwischen den Replikaten führt
- ~> für eine schnelle Fehlererholung sollte er so klein wie möglich sein
- ~> aus dem Taschenrechner-Beispiel (s. Folie IX/10) wird klar:
 - dies ist der Fall, wenn keine Berechnung aktiv ist
- ☞ zu bevorzugen ist der Grundzustand (engl. *ground state*)
 - systemweit ist keine Berechnung aktiv, alle Nachrichtenkanäle sind leer
 - der dynamische Zustand ist im Grundzustand also am kleinsten
 - ~> der Grundzustand ist für die Reintegration besonders geeignet
 - ~> der Grundzustand sollte regelmäßig eingenommen werden
 - eine zyklische Ablaufstruktur (s. Folie VIII/16) unterstützt dies
 - nach jedem Zyklus „lesen, rechnen, schreiben“ wäre dies möglich



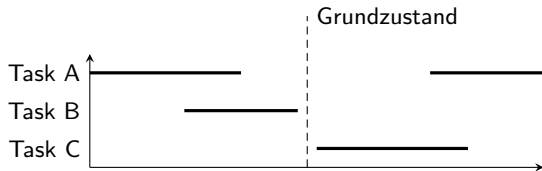
Erreichbarkeit des Grundzustands

- die Einplanungsstrategie hat hier einen signifikanten Einfluss
 - ereignisgesteuerte Einplanung:



- zeitlicher Ablauf hängt vom Eintreffen der Ereignisse ab
- ~ das Erreichen des Grundzustands kann **nicht garantiert** werden

- zeitgesteuerte Einplanung:



- zyklensynchrone Ausführung erlaubt die **Einplanung des Grundzustands**
- für Replikdeterminismus (s. Folie VIII/17) ist dies ohnehin sinnvoll



- der dynamische Zustand lässt sich weiter aufteilen:

Zustand des physikalischen Objekts

- kann durch Sensoren erneut erfasst werden
 - das Replikat wird hier mit der physikalischen Umwelt synchronisiert
- ↪ zur Restauration **kein Transfer erforderlich**

Standardausgaben (engl. *restart vector*)

- **sichere Ausgabewerte** für das kontrollierte Objekt
 - sie werden für die „Initialisierung“ von Aktoren etc. verwendet
 - sind i. d. R. bereits zum Entwurfszeitpunkt festlegbar
- ↪ zur Restauration **kein Transfer erforderlich**

sonstiger dynamischer Zustand

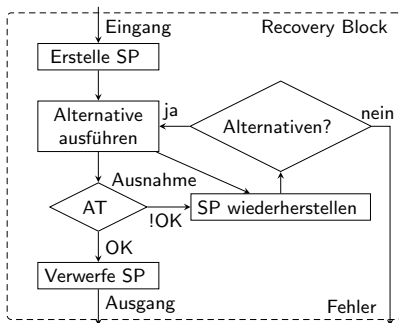
- „der Rest“, gehört also nicht zu den obigen Kategorien
- ↪ zur Restauration ist ein **Transfer notwendig**
 - kann durch eine Überarbeitung des Systementwurfs evtl. reduziert werden
 - ↪ z. B. indem ein zusätzlicher Sensor zum Einsatz kommt




- 1 Überblick
- 2 Grundlagen
 - Vorgehen bei der Reintegration
 - Fehlererholung
 - Interner Zustand
- 3 Distributed Recovery Blocks**
- 4 Zustandstransfer
- 5 Zusammenfassung



- Rückgriff auf **alternative Implementierungen** im Fehlerfall
 - **wiederholte Ausführung** \rightsquigarrow Toleranz **transienter Fehler**
 - **alternative Implementierung** \rightsquigarrow Toleranz von **Software-Defekten**
- ☞ **Sicherungspunkte (SP)** ermöglichen wiederholte Ausführung
 - Fehlererkennung/Validierung des Ergebnisses \mapsto Akzeptanztest (AT)
 - AT umfasst sowohl logische als auch zeitliche Fehler (\rightsquigarrow „Watchdog“)




Distributed Recovery Blocks (DRB, [7])

- parallele Ausführung verschiedener Alternativen in zwei Redundanzen
 - primäres und sekundäres Replikat **überwachen sich gegenseitig**
 - jedes Replikat für sich implementiert denselben „Recovery Block“
 - **primäres Replikat** \rightsquigarrow Ausführung der bevorzugten Implementierung
 - **sekundäres Replikat** \rightsquigarrow Ausführung der alternativen Implementierung
 - **vorsorgliche Fehlererholung** im Sekundärreplikat
 - falls die bevorzugte Implementierung im Primärreplikat fehlschlägt
 - \rightsquigarrow das Ergebnis wurde bereits vom Sekundärreplikat berechnet
 - \rightsquigarrow sehr schnelle Fehlererholung, **keine Rückwärtsbewegung**
 - \rightsquigarrow Sicherungspunkt wird nur für die Fehlerbeseitigung benötigt
 - \rightsquigarrow verschiedene Rekonfigurationsszenarien denkbar [4]
 - Primär- und Sekundärreplikat tauschen die Rollen
 - tausche die bevorzugte und die alternative Implementierung
-  **Problemfall:** Primär- und Sekundärreplikat schlagen fehl
- eigenständige Fehlererholung **nicht möglich**
 - schließlich sind beide Implementierungsvarianten bereits fehlgeschlagen
 - \rightsquigarrow Zustand muss „von außen“ restauriert werden

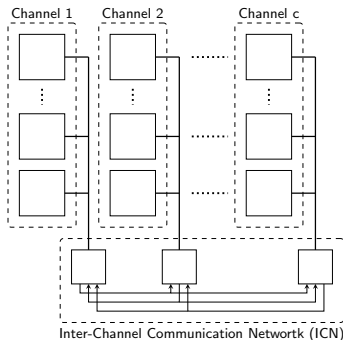


- 1 Überblick
- 2 Grundlagen
 - Vorgehen bei der Reintegration
 - Fehlererholung
 - Interner Zustand
- 3 Distributed Recovery Blocks
- 4 Zustandstransfer**
- 5 Zusammenfassung



- dynamischer Zustand \rightsquigarrow **Zustandsabgleich** (engl. *state alignment*)
 - vorher kann das fehlgeschlagene Replikat nicht integriert werden
 - auch als **Zustandsrestaurierung** (engl. *state restoration, SR*) bekannt
 - idealerweise gelingt dies in „einem Rutsch“ (engl. *one-shot SR*)
 - entweder ist der zu restaurierende Zustand hinreichend klein, oder
 - das ist **abhängig von der konkreten Anwendung**
 - oder das System kann zu diesem Zweck angehalten werden
 - dies ist in sicherheitskritischen Echtzeitsysteme **i. d. R. nicht möglich**
-  normalerweise muss der Abgleich **im laufenden Betrieb** erfolgen [2]
- **Echtzeitanforderungen** verbieten i. d. R. das Anhalten des Systems
 - allenfalls ein **reduzierter Betriebsmodus** (engl. *degraded mode*) ist denkbar
 - dieser räumt dem Zustandsabgleich **mehr Ressourcen** ein
 - ohne jedoch die Ausführung kritischer Aufgaben zu gefährden
 - im folgenden zwei Modi für den Zustandsabgleich:
 - „**Running SR**“ (s. Folie IX/21) und „**Recursive SR**“ (s. Folie IX/24)





- Grundbaustein: redundante Kanäle (engl. *channel*) \approx Replikat
 - Kanäle sind fehlereingrenzende Sicherheitshüllen (engl. *fault containment*)
 - ein Kanal besteht aus mehreren Rechenknoten
 - agieren als Mehrprozessorsystem mit gemeinsamen Speicher
 - dynamische Ablaufplanung auf den einzelnen Knoten
- verfügen über ein gemeinsames Kommunikationssystem \mapsto ICN
 - statische, zyklische Kommunikation, Uhrensynchronisation



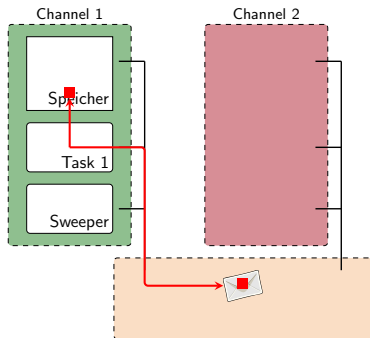
Grundsätzlicher Ablauf des Zustandsabgleichs

- ein GUARDS-System umfasse C redundante Kanäle
 - der Zustand einer dieser Kanäle muss restauriert werden
- ↪ die verbleibenden $C - 1$ Kanäle sind die Lieferanten des Zustands
- 1 das System nimmt den Modus „SR Mode“ ein
 - ggf. wird hierfür die Leistung des Normalbetriebs reduziert
 - kritische Aufgaben werden weiterhin abgearbeitet
- 2 aktive Kanäle gehen in den Zustand „put state“ ↪ Lieferung
 - hierfür wird eine zusätzliche Aufgabe, der „Sweeper“ eingeplant
 - wickelt den Speichertransfer zum ausgefallenen Kanal ab
 - teilt sich die Ressourcen mit der verbliebenen Anwendung
- 3 ausgefallener Kanal geht in den Zustand „get state“ ↪ Annahme
 - hierfür wird eine zusätzliche Aufgabe, der „Catcher“ eingeplant
 - nimmt die über das ICN „gelieferten“ Datenpakete an und verarbeitet sie
 - läuft exklusiv als einzige Aufgabe im ausgefallenen Kanal
- 4 Zustandsabgleich endet am Ende eines ICN-Zyklus
 - im nächsten Zyklus setzt die normale Ablaufplanung ein
- ↪ Transfers müssen im selben Zyklus gestartet und vollendet werden



„Running State Restoration“

Exemplarisch für ein zweikanaliges System

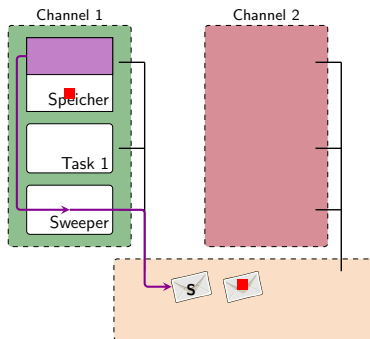


- Speichermodifikation während des Abgleichs ist möglich
 - Mitteilung der Änderungen per ICN



„Running State Restoration“

Exemplarisch für ein zweikanaliges System

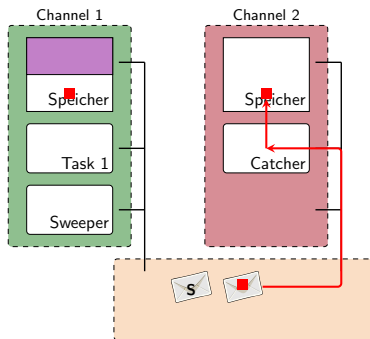


- Speichermodifikation während des Abgleichs ist möglich
 - Mitteilung der Änderungen per ICN
- **Sweeper** übernimmt den Rest
 - Versand ebenfalls per ICN



„Running State Restoration“

Exemplarisch für ein zweikanaliges System

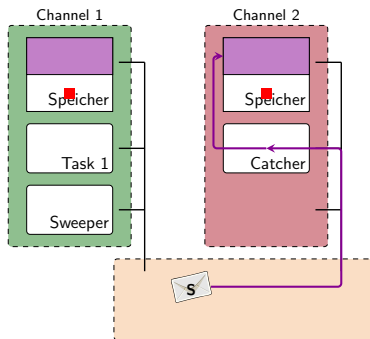


- Speichermodifikation während des Abgleichs ist möglich
 - Mitteilung der Änderungen per ICN
- **Sweeper** übernimmt den Rest
 - Versand ebenfalls per ICN
- **Catcher** nimmt Nachrichten an
 - aktualisiert den Speicher



„Running State Restoration“

Exemplarisch für ein zweikanaliges System

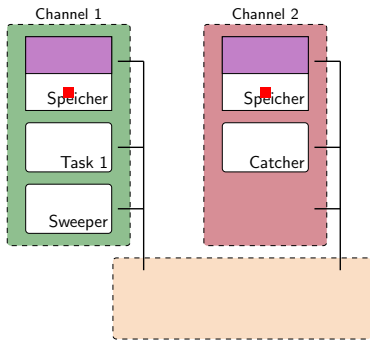


- Speichermodifikation während des Abgleichs ist möglich
 - Mitteilung der Änderungen per ICN
- **Sweeper** übernimmt den Rest
 - Versand ebenfalls per ICN
- **Catcher** nimmt Nachrichten an
 - aktualisiert den Speicher
 - **Reihenfolge** muss beachtet werden!



„Running State Restoration“

Exemplarisch für ein zweikanaliges System



- Speichermodifikation während des Abgleichs ist möglich
 - Mitteilung der Änderungen per ICN
- **Sweeper** übernimmt den Rest
 - Versand ebenfalls per ICN
- **Catcher** nimmt Nachrichten an
 - aktualisiert den Speicher
 - **Reihenfolge** muss beachtet werden!

- der Abgleich endet deterministisch nach endlich vielen ICN-Zyklen
 - solange genügend Bandbreite vorhanden ist
 - so dass der „Sweeper“ mindestens ein Byte/Zyklus kopieren kann



hohe Anforderung an das **Kommunikationssystem**

- mehrere Klienten müssen gleichzeitig bedient werden
 - laufende Anwendungen, der „Sweeper“, kontinuierliche Aktualisierungen



„Running State Restoration“ – Bitte Beachten!

- kont. Aktualisierung \mapsto Speicherzugriff & Nachrichtenversand
 - \rightsquigarrow unteilbare Ausführung nötig \mapsto Konsistenz der Speicherstelle wahren
- ☞ Reihenfolge der Speichertransfers ist wichtig!
 - „Catcher“ \rightsquigarrow trivial
 - Nachrichten werden in Ankunftsreihenfolge verarbeitet
 - „Sweeper“ \rightsquigarrow schwierig
 - Nachrichten müssen in richtigen Reihenfolge ans ICN übergeben werden
 - \rightsquigarrow Aufgaben und der „Sweeper“ müssen am ICN ausgerichtet werden
 - \rightsquigarrow Beeinflussung der Ablaufplanung durch gezielte Terminvergabe
- mögliche Implementierung:
 - für den „Sweeper“ wird Platz am Ende des ICN-Zyklus reserviert
 - der „Sweeper“ wird nach allen anderen Aufgaben eingeplant
 - \rightsquigarrow keine „überlappenden“ Zustandstransfers
 - der „Catcher“ muss nur den Zyklus n beenden, bevor er mit $n + 1$ startet



„Running State Restoration“ – für $C > 2$ Kanäle

- mehrere Kanäle ermöglichen eine Verbesserung des Abgleichs



Beschleunigung durch parallele Zustandstransfers

- **Segmentierung** des abzugleichende Speichers
 - jeder funktionsfähige Kanal gleicht einen Teil des Zustands ab
 - analoge Aufteilung der kontinuierliche Aktualisierung
- begrenzender Faktor ist **das Kommunikationssystem**
 - parallele Zustandstransfers erfordern auch **mehr Bandbreite**



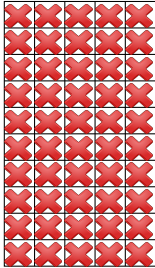
Absicherung des Zustandsabgleichs

- latente Fehler im Zustand der übrigen Kanäle würden einfach kopiert
 - ~> erhöht die Gefahr von Gleichtaktfehlern
- explizite **Einigung der Zustände** unter den $C - 1$ übrigen Kanälen
 - ~> Ausführung eines vom ICN zu unterstützenden **Einigungsprotokolls**
 - ~> **sehr hoher Kommunikationsaufwand**
- statt der Daten selbst werden nur **Signaturen** ausgetauscht
 - Reduktion des Kommunikationsaufwands
 - **Fehlererkennungsrate** steht und fällt mit der verwendeten Signatur



„Recursive State Restoration“

- Zustandsänderungen werden nicht kontinuierlich mitgeteilt
- Verwendung von **indiziertem Speicher** (engl. *tagged memory*)
 - jede Speicherstelle besitzt einen Merker, ob sie abgeglichen wurde



- 1 zunächst sind alle Speicherstellen markiert
 - d. h. sie müssen abgeglichen werden

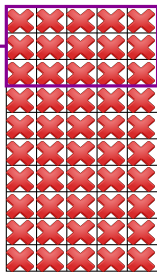
Task 1

Sweeper

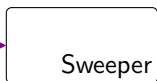
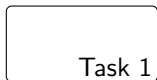


„Recursive State Restoration“

- Zustandsänderung werden nicht kontinuierlich mitgeteilt
- ☞ Verwendung von **indiziertem Speicher** (engl. *tagged memory*)
 - jede Speicherstelle besitzt einen Merker, ob sie abgeglichen wurde

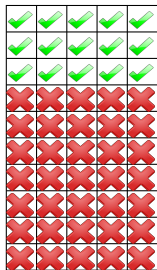


- 1 zunächst sind alle Speicherstellen markiert
 - d. h. sie müssen abgeglichen werden
- 2 „Sweeper“ gleicht nun Speicherstellen ab
 - wieviele hängt von der Bandbreite ab



„Recursive State Restoration“

- Zustandsänderungen werden nicht kontinuierlich mitgeteilt
- ☞ Verwendung von **indiziertem Speicher** (engl. *tagged memory*)
 - jede Speicherstelle besitzt einen Merker, ob sie abgeglichen wurde



Task 1

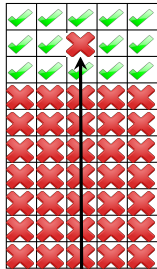
Sweeper

- 1 zunächst sind alle Speicherstellen markiert
 - d. h. sie müssen abgeglichen werden
- 2 „Sweeper“ gleicht nun Speicherstellen ab
 - wieviele hängt von der Bandbreite ab
 - und setzt die Markierung zurück



„Recursive State Restoration“

- Zustandsänderungen werden nicht kontinuierlich mitgeteilt
- Verwendung von **indiziertem Speicher** (engl. *tagged memory*)
 - jede Speicherstelle besitzt einen Merker, ob sie abgeglichen wurde



Task 1

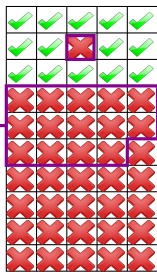
Sweeper

- 1 zunächst sind alle Speicherstellen markiert
 - d. h. sie müssen abgeglichen werden
- 2 „Sweeper“ gleicht nun Speicherstellen ab
 - wieviele hängt von der Bandbreite ab
 - und setzt die Markierung zurück
- 3 Anwendung modifiziert eine Speicherstelle
 - Markierung wird wieder gesetzt



„Recursive State Restoration“

- Zustandsänderungen werden nicht kontinuierlich mitgeteilt
- Verwendung von **indiziertem Speicher** (engl. *tagged memory*)
 - jede Speicherstelle besitzt einen Merker, ob sie abgeglichen wurde

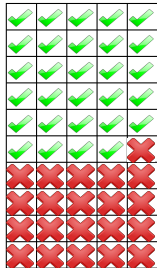


- 1 zunächst sind alle Speicherstellen markiert
 - d. h. sie müssen abgeglichen werden
- 2 „Sweeper“ gleicht nun Speicherstellen ab
 - wieviele hängt von der Bandbreite ab
 - und setzt die Markierung zurück
- 3 Anwendung modifiziert eine Speicherstelle
 - Markierung wird wieder gesetzt
- 4 anschließend tritt der „Sweeper“ in Aktion
 - gleicht wieder Speicherstellen ab



„Recursive State Restoration“

- Zustandsänderungen werden nicht kontinuierlich mitgeteilt
- Verwendung von **indiziertem Speicher** (engl. *tagged memory*)
 - jede Speicherstelle besitzt einen Merker, ob sie abgeglichen wurde



Task 1

Sweeper

- 1 zunächst sind alle Speicherstellen markiert
 - d. h. sie müssen abgeglichen werden
- 2 „Sweeper“ gleicht nun Speicherstellen ab
 - wieviele hängt von der Bandbreite ab
 - und setzt die Markierung zurück
- 3 Anwendung modifiziert eine Speicherstelle
 - Markierung wird wieder gesetzt
- 4 anschließend tritt der „Sweeper“ in Aktion
 - gleicht wieder Speicherstellen ab
 - und setzt die Markierungen zurück
- 5 bis der Zustand abgeglichen ist ...





genau das ist das Problem: **Wie lange dauert der Abgleich?**

- ↪ gesonderter **Zähler** verwaltet Umfang des abzugleichenden Speichers
 - eine bestimmte Anzahl von Bytes kann „in einem Rutsch übertragen“ werden
- ↪ dies wird durch den Zähler überwacht ↪ „**last shot**“

■ **Vorteile** der „Recursive State Restoration“

- **Konsistenz** der übertragenen Werte ist jederzeit gegeben
 - nur der „Sweeper“ überträgt den Zustand zum fehlerhaften Kanal
- es erfolgt **kein kontinuierlicher Abgleich** ↪ **Entlastung des ICN**
- indizierter Speicher ist **gut untersucht und verstanden**
 - es existieren auch dedizierte hardware-basierte Implementierungen [1]

■ **Nachteile** der „Recursive State Restoration“

- **unklarer Dauer** bis zum erfolgreichen Abschluss des Abgleichs
- „**last shot**“ impliziert ein **Stillhalten**
 - bis zur Vollendung des Abgleichs sind keine Änderungen erlaubt
- mehrere „**Sweeper**“ müssen sich über den „**last shot**“ einigen
 - schließlich verwalten sie **unabhängige Zähler**



- 1 Überblick
- 2 Grundlagen
 - Vorgehen bei der Reintegration
 - Fehlererholung
 - Interner Zustand
- 3 Distributed Recovery Blocks
- 4 Zustandstransfer
- 5 Zusammenfassung**



Problemstellung \mapsto ein Replikat fällt aus

- dies führt zu einer **verminderten Fehlertoleranz**
- \leadsto Reintegration des ausgefallene Knotens

Grundlagen für die Reintegration

- reaktiv, proaktiv und reaktiv-proaktiv
- Vorwärts- und Rückwärtsbewegung
- Initialzustand und dynamischer Zustand
- Bestandteile und Minimierung des dynamischen Zustands

„Recovery Blocks“ Reintegration durch Rückwärtsbewegung

- „Distributed Recovery Blocks“ \mapsto parallele Ausführung
- vorsorgliche Fehlererholung \leadsto Vorwärtsbewegung
 - Rückwärtsbewegung nur für die Fehlerbeseitigung

Zustandstransfer von einem funktionsfähigen Replikat

- „one-shot SR“ vs. Zustandstransfer über mehrere Schritte
- „Running SR“ vs. „Recursive SR“



- [1] ADAMS, S. ; SIMS, T. :
A tagged memory technique for recovery from transient errors in fault tolerant systems.
In: *Proceedings of the 11th International Conference on Real-Time Systems (RTSS '90)*.
Washington, DC, USA : IEEE Computer Society Press, Dez. 1990. –
ISBN 0-8186-2112-5, S. 312-321
- [2] BONDAVALLI, A. ; DI GIANDOMENICO, F. ; GRANDONI, F. ; POWELL, D. ; RABEJAC, C. :
State restoration in a COTS-based N-modular architecture.
In: *Proceedings of the 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98)*.
Washington, DC, USA : IEEE Computer Society Press, Apr. 1998. –
ISBN 0-8186-8430-5, S. 174-183
- [3] 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

- [4] KIM, K. ; YOON, J. :
Approaches to implementation of a repairable distributed recovery block scheme.
In: *Proceedings of the 18th International Symposium on Fault-Tolerant Computing (FTCS-18)*, 1988, S. 50–55
- [5] KOPETZ, H. :
Real-Time Systems: Design Principles for Distributed Embedded Applications.
Kluwer Academic Publishers, 1997. –
ISBN 0–7923–9894–7
- [6] POWELL, D. ; ARLAT, J. ; BEUS-DUKIC, L. ; BONDAVALLI, A. ; COPPOLA, P. ;
FANTECHI, A. ; JENN, E. ; RABEJAC, C. ; WELLINGS, A. :
GUARDS: a generic upgradable architecture for real-time dependable systems.
In: *IEEE Transactions on Parallel and Distributed Systems* 10 (1999), Jun., Nr. 6, S.
580–599.
<http://dx.doi.org/10.1109/71.774908>. –
DOI 10.1109/71.774908. –
ISSN 1045–9219
- [7] PULLUM, L. L.:
Software fault tolerance techniques and implementation.
Norwood, MA, USA : Artech House, Inc., 2001. –
ISBN 1–58053–137–7