


Zusammenfassung der Echtzeitsysteme VO SS2024

 ... Wurde in einer mündlichen Prüfung gefragt

 ... Wurde in keiner mündlichen Prüfung gefragt

1. Definitions

Was ist ein RTS?

o1. Def: RT-Systeme sind Systeme, in denen die Korrektheit abhängt von:

- den logischen Ergebnissen der Berechnungen
- der physikalischen Zeit, wann diese Ergebnisse produziert wurden

o2. Def: RT-Systeme sind Systeme, bei deren Design die Dynamiken eines physikalischen Prozesses beachtet werden.

o RTS ist meist ein Teil von einem embedded System:

- Interaktion mit Sensoren / Aktoren
- Dependability
- Ressourceneffizienz
- Erhöhung der Wichtigkeit von Security

Challenges / Schwierigkeiten bei RTS

1) Reaktives Verhalten

- o kontinuierliches Arbeiten
- o Umgebung bestimmt Geschwindigkeit

2) Parallelität

- o Geräte arbeiten alle parallel
- o Determinismus schwer zu behalten

⇒ Reproduzierbarkeit der Testfälle sehr schwer, da alles parallel läuft

3) Garantierte Response Time

4) Interaktion mit Special Purpose Hardware

↑

zB Sensoren / Aktoren

5) Maintenance schwer

zB Software Update bei Auto nur in Werkstatt

⇒ dazwischen muss es funktionieren

6) Anspruchsvolle Umgebung

7) Begrenzte Ressourcen

8) Zuverlässigkeit & Sicherheit

Arten / Klassen von Echtzeitsystemen

◦ Soft RTS

- Ergebnis hat auch nach Deadline Nutzen

◦ Firm RTS

- Ergebnis hat kein Nutzen nach Deadline

◦ Hard RTS

- Ergebnis hat kein Nutzen nach Deadline

- Verpassen einer Deadline kann katastrophale Folgen haben (= harte Deadline)

- Min. 1 harte Deadline

Guaranteed Timeliness

- Fehler-/Lasthypothese vorhanden
- Temporale Korrektheit kann durch analytische Argumente gezeigt werden
- Assumption Coverage wichtig

Best Effort

- Temporale Korrektheit kann nicht durch analytische Argumente gezeigt werden
- ⇒ Temporale Korrektheit basiert auf statistischen Argumenten, auch innerhalb der vorgegebenen Fehler-/Lasthypothese

⇒ **Hard** RTS basieren auf **Guaranteed Timeliness** & müssen auch **Resource Adequacy** berücksichtigen

Resource Adequacy

Es muss **immer genug** Rechenkapazität vorhanden sein, um auch **peak load** (Hochlast) und Fehlerszenarien zu beherrschen.

Rare Event

Ist ein wichtiges Event, das selten vorkommt, aber für Hochlast sorgt.

⇒ In manchen Fällen hängt der Nutzen eines Systems vom erwarteten Verhalten bei Rare Events ab (z.B. Flugsteuerungssystem)

Charakteristik	Hard RTS	Soft RTS
Deadlines	Hard	Soft
Taktung gegeben durch	Umgebung	Computer
Peak-Load Performance	Erwartbar	verringerte Performance
Fehlererkennung	System	User

Fail-Safe System

- Hat einen **safe state** im Environment, der bei einem Systemausfall erreicht werden kann
- Hohe Fehlererkenntnisabdeckung wichtig
- Verwendung von Watchdog, heart-beat signal

Fail-operational System

- Muss bei einem Systemausfall weiter „funktionieren“, da kein safe state vorhanden ist
- Aktive Redundanz

2a Time & Order

Kausale Ordnung

- **nicht** abhängig von physikalischer **Zeit**, sondern Ableitung aus **kausaler** Abhängigkeit zwischen 2 Ereignissen

1: $a \rightarrow b \iff a, b$ sind Events in einem sequentiellen Prozess, wobei a vor b ausgeführt wird

2: $a \rightarrow b \iff a$ ist ein send-Event einer Nachricht vom Prozess p_i und b ist ein receive-Event einer Nachricht vom Prozess p_j

3. $a \rightarrow b \wedge b \rightarrow c \implies a \rightarrow c$ (Transitivität)

Zeitliche Ordnung

- Ableitung von Zeitstempeln der physikalischen Zeit
- Aus kausaler Ordnung folgt zeitliche Ordnung
 - ⇒ Zeitliche Ordnung ist notwendig für kausale Ordnung

Logical Clocks

- Verwendet logische Zeitstempel (Counter), um Informationen über kausale Abhängigkeit von Events darzustellen
- Verwendet keine physikalische Zeit
- Erwünschte Eigenschaften:
 - $a \rightarrow b \Rightarrow C(a) < C(b)$... Monotonie, Konsistenz
 - $a \rightarrow b \Leftrightarrow C(a) < C(b)$... Strenge Konsistenz

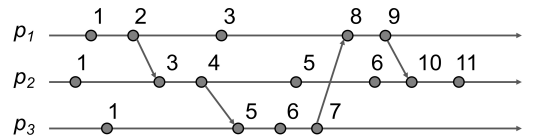
Lamport's Logical Clocks

- Logical Clocks der einzelnen Prozesse p_i stellen deren lokale Sicht der globalen Sicht dar
- Ganze Zahl C_i stellt die lokale Uhrzeit von p_i dar
- Clock Update Regeln:
 - bei jedem lokalen Event (z.B. event, send) bei p_i :
 - $C_i = C_i + 1$
 - jede Nachricht transportiert den Wert C_{msg} der Uhr des Senders

- bei **Empfangen** einer Nachricht mit dem Zeitstempel C_{msg} :

$$C_i = \max(C_i, C_{msg}) + 1$$

o Eigenschaften:



- Totale Ordnung

- Konsistent: $a \rightarrow b \Rightarrow C(a) < C(b)$

- **Keine** starke Konsistenz ∇

Vector Time

o Basiert auf **Logical Clocks**.

o **n-dimensionaler** Vektor bei p_k : $V_k = \begin{bmatrix} 1 \\ \vdots \\ k \\ \vdots \\ n \end{bmatrix}$

$(V_k)_k \dots$ Wert der lokalen Uhr von p_k

$(V_k)_i \dots p_k$'s Wissen über lokale Zeit von p_i

o Clock Update Regeln:

- p_k updatet $(V_k)_k$ für **jedes lokale** Event:

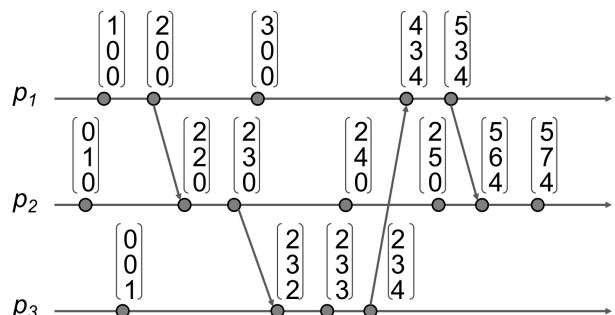
$$(V_k)_k = (V_k)_k + 1$$

- jede Nachricht von p_k transportiert Vektor $V_k (=V_{msg})$

- wenn p_k eine Nachricht mit V_{msg} erhält:

$$1 \leq i \leq n: (V_k)_i = \max((V_k)_i, (V_{msg})_i)$$

$$(V_k)_k = (V_k)_k + 1$$



◦ Eigenschaften:

- $a \parallel b \iff \exists i, k: (V_a)_i > (V_b)_i \wedge (V_a)_k < (V_b)_k$
↑
Parallelität

- $a \rightarrow b \iff \forall i: (V_a)_i \leq (V_b)_i \wedge \exists i: (V_a)_i < (V_b)_i$

⇒ **Strenge Konsistenz**

Physical Clocks

◦ Unter Zeit wird ein kontinuierlicher physikalischer Prozess verstanden, d.h. die Zeitachse ist eine **unendliche, geordnete, dichte** Menge von Zeitpunkten

◦ Physical Clock c ist eine Uhr, die dieses Konzept realisiert:

- Besteht aus **Zähler & Oszillator**

- **Microticks** werden generiert durch periodisches Inkrementieren des Zählers (verhält sich nach physikalischen Regeln)

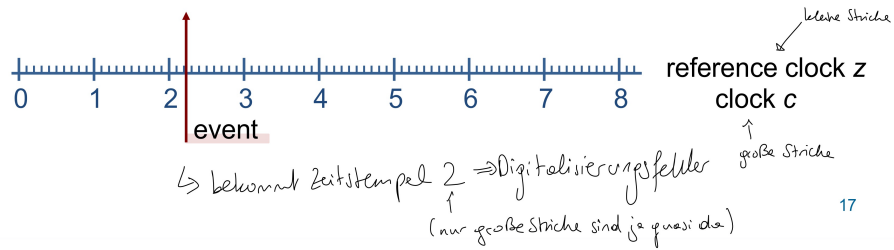
- Reference Clock z ist eine theoretisch **perfekte** Uhr, mit der man Uhren vergleicht

- Zeitstempel eines Events gibt den Wert der Uhr c in dem Moment des Events an. Notation: $c(\text{event})$

- **Granularität** von c beschreibt die nominelle Anzahl der **Microticks** der **Reference Clock z** zwischen 2 **aufeinanderfolgenden** Microticks der **Uhr c** :

$$g^c = z(\text{microtick}_{i+1}^c) - z(\text{microtick}_i^c)$$

- **Digitalization error**: Events, die zu unterschiedlichen, aber **nahen** Zeitpunkten stattfinden, können dennoch **gleiche** Zeitstempel bekommen aufgrund der **Granularität**.



Global Time

- In einem **verteilten** System haben alle Uhren **dieselbe Granularität** g^{global} (i guess?).

Precision

- Beschreibt, **wie nah** Uhren aneinander sind.
- Offset zwischen Uhren j und k bei tick i :

$$\text{offset}_i^{jk} = |z(\text{microtick}_i^j) - z(\text{microtick}_i^k)|$$
- Precision = **max.** Offset zwischen 2 **beliebigen** Uhren eines **Systems** bei microtick i :

$$P_i = \max_{j,k} \{ \text{offset}_i^{jk} \}$$

Accuracy

- **Offset** zwischen der Uhr k & der Reference Clock z bei tick i :

$$\text{offset}_i^{k,z(k)} = |z(\text{microtick}_i^k) - z(\text{microtick}_i^{z(k)})|$$

- **Accuracy** beschreibt den **maximalen Offset** einer **bestimmten** Uhr von der **Referenzuhr** innerhalb eines **bestimmten Intervalls**.
- Wenn **alle** Uhren eines Systems in der Accuracy A liegen, dann

hat das System eine Precision $\Pi \leq 2A$

Wieso braucht man eine Global Time Base?

- Events aus verschiedenen Rechnern könnte man nicht mehr in Beziehung setzen
- Intervallmessungen wären nicht möglich
- Zeitliche Abweichung in der Kommunikation könnte Konsequenzen haben
- State estimation wäre zu schwer

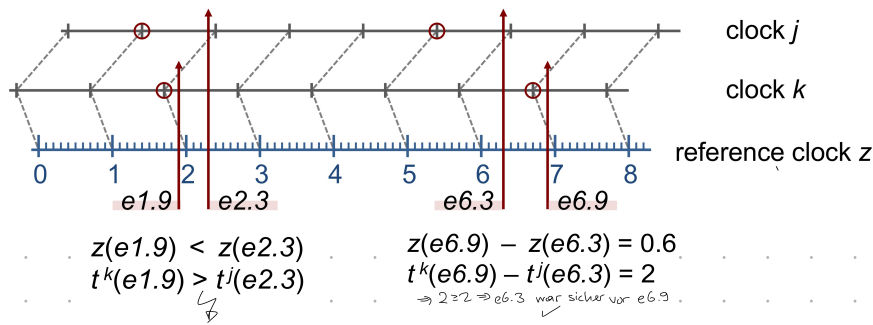
Bedingungen für eine Global Time Base

- Gleichmäßiges Verhalten / Vor allem bei Resynchronisation keine Diskontinuitäten
- Bekannte Precision Π
- Hohe Verlässlichkeit
- Metrik einer Sekunde (für Steuerungen & Regelungstechnik)

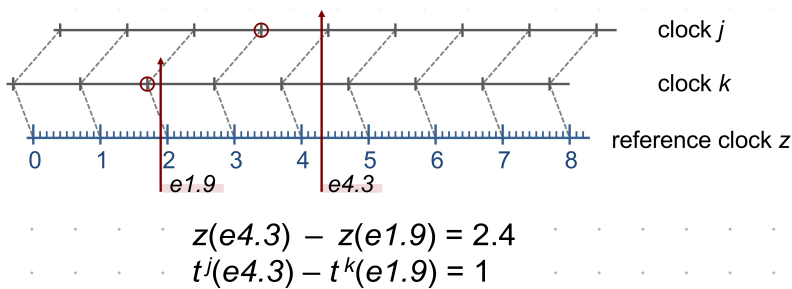
Reasonableness Condition

- Verwendet die Precision Π , um die benötigte Granularität zu bestimmen
- Eine Zeitbasis heißt reasonable, wenn: $g^{\text{global}} > \Pi$
($g^{\text{global}} < \Pi \Rightarrow$ Informationsverlust)
- Aus der Reasonableness folgt:
 - Synchronisationsfehler ist kleiner als ein Makrogranulat (Abstand zwischen 2 Ticks)
 - Für ein Event sind die Abstände der Zeitstempel der logischen Clocks maximal 1, also: $\forall e: |t^j(e) - t^k(e)| \leq 1$

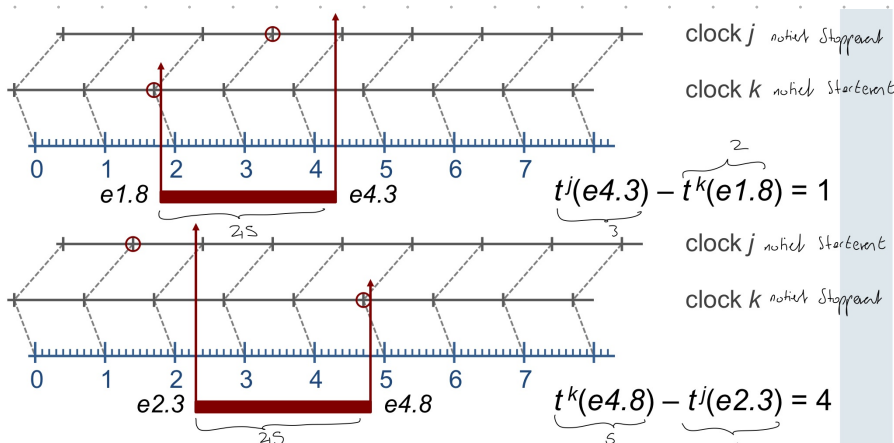
- Um die zeitliche Ordnung zweier Events zu bestimmen, müssen die (globalen) Zeitstempel der Events um mindestens 2 Ticks auseinander liegen



- Ein Abstand von $2 \cdot g^{\text{global}}$ reicht nicht, um die zeitliche Ordnung zu bestimmen!



Measurement of Durations



\Rightarrow Max. $1g$ -Fehler beim Registrieren von einem Event $\Rightarrow 2g$ bei 2 Events

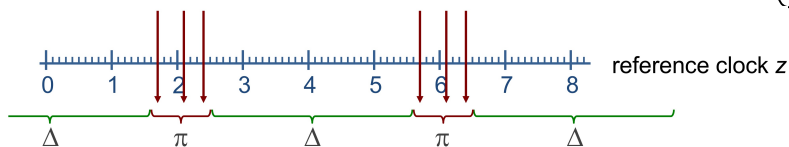
\Rightarrow Real duration: $d_{\text{obs}} - 2 \cdot g^{\text{global}} < d_{\text{true}} < d_{\text{obs}} + 2 \cdot g^{\text{global}}$

Agreement on Event Order - Dense Time

- Wird ein Event von 2 Knoten beobachtet, können deren lokale Zeitstempel um 1 Tick voneinander abweichen
 - ⇒ Kommunikation zwischen Knoten nötig, um eine konsistente Sicht über den globalen Zeitpunkt des Events zu etablieren (⇒ dadurch Ordnung der Events möglich)
 - ⇒ **Agreement Protocol** um explizites Agreement on Event Order herzustellen
- Es ist unmöglich, bei 2 Events in einer dense Timeline die temporale Order konsistent für alle Events festzustellen, die **innerhalb** eines Intervalls mit Dauer $< 3 \cdot g^{\text{global}}$ liegen
 - ⇒ **Explicit Agreement** nötig für arbitrary Events
 - ⇒ **Alternative**: O/Δ -Precedent Event Set mit $\Delta \geq 3 \cdot g^{\text{global}}$

Π/Δ -Precedence von Mengen von Events

- Es wechseln sich Intervalle der Länge Π , in denen Ereignisse auftreten, mit **längeren** Intervallen Δ , in denen keine Ereignisse auftreten, ab.



4 Fundamentals über Zeitmessung

- Wenn ein Ereignis von 2 Knoten **observiert** wird, weichen die lokalen Zeitstempel **höchstens** um 1 Tick voneinander ab
- Wird ein **Intervall** mit Dauer Δ gemessen, liegt der

tatsächliche Wert des Intervalls zwischen $d_{obs} - 2 \cdot g^{global}$ und $d_{obs} + 2 \cdot g^{global}$

- Bei 2 Events kann man Aussagen über die Ordnung nur machen, wenn die Differenz der 2 Zeitstempel ≥ 2 ist, also:

$$|t^j(e_1) - t^k(e_2)| \geq 2 \quad j, k \dots \text{Clocks}$$

- Man kann die Ordnung von Events immer bestimmen, wenn das Event Set O/Δ -precedent und $\Delta \geq 3 \cdot g^{global}$ ist

Dense Time

- Ereignisse dürfen jederzeit auftreten

Sparse Time (Π/Δ -sparse Timebase)

- Ereignisse dürfen nur innerhalb der Zeitintervalle Π stattfinden, gefolgt von einem Intervall der Stille Δ

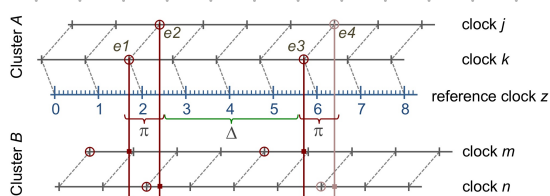
Agreement on Event Order - Sparse Time

- Annahme: 2 Berechnungs-Cluster A, B:

- In jedem Cluster sind die Uhren synchronisiert ($g = g^{global}$)
- Keine Synchronisation zwischen A und B
- A generiert Events, die B ordnen muss

⇒ Timebase von A muss $1g/Ng$ -sparse sein mit $N \geq 4$

$N=3$ nicht ausreichend:



$e1, e2 \dots$ generated in same activity interval: $t^m(e2) - t^m(e1) = 2$
 $e2, e3 \dots$ gen. in different activity interval: $t^m(e3) - t^m(e2) = 2$

nicht unterscheidbar

2b Clock Synchronization

Clock Drift

◦ Abstand zweier Ticks (in Microticks, gemessen von z) im Verhältnis zur Granularität g :

$$\text{drift}_i^k = \frac{z(\text{microtick}_{i+1}^k) - z(\text{microtick}_i^k)}{g^k} \quad \text{Optimal: drift} = 1$$

Drift Rate

◦ Gibt an, um wieviel die Uhr prozentuell pro lokalen Taktzyklus von der idealen Uhr abweicht:

$$p_i^k = \left| \frac{z(\text{microtick}_{i+1}^k) - z(\text{microtick}_i^k)}{g^k} - 1 \right| \quad \text{Optimal: } p = 0$$

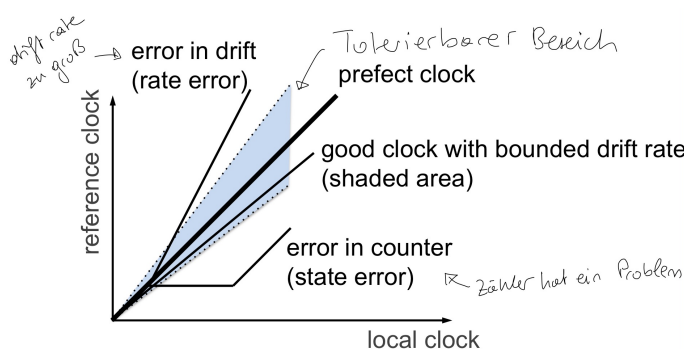
◦ Drift Rate einer realen Uhr: $10^{-8} \dots 10^{-2}$

Clock Failure Modes

◦ Uhr besteht aus Zähler & Oszillator:

- rate error: Schwingung zu schnell/langsam

- error in counter: Zähler springt vor oder bleibt stecken (zB Bit Flip)



Externe Clock Synchronization

- Resynchronisation mittels Referenzuhr
 - zu externen Clocks hat man nicht immer gute Availability (zB im Tunnel)
 - aber dafür long-term Stabilität

Interne Clock Synchronization

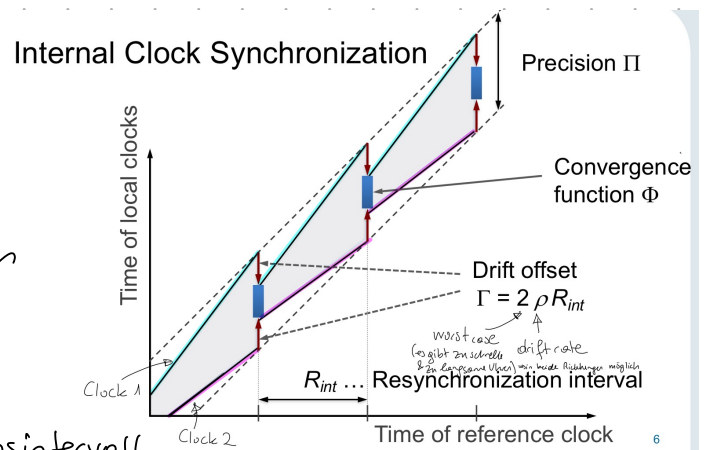
- Gegenseitiges Resynchronisieren von Clocks innerhalb eines Ensembles, um eine bestimmte Precision beizubehalten
 - gute kurze Stabilität
 - hohe Availability

worst case (zu schnell oder zu langsam)
↳ in beide Richtungen möglich

Drift Offset $\Gamma = 2 \cdot \rho \cdot R_{int}$

Drift Rate ρ

Resynchronisationsintervall R_{int}



Synchronization Condition

- Damit die Uhren mit Precision Π synchronisiert bleiben, muss gelten: $\Phi + \Gamma \leq \Pi$
- Φ ... Konvergenzfunktion = Max. Offset nach Synchronisation
Abhängig von:
 - Synchronisations-Algorithmus
 - Message latency jitter ϵ (= transmission-time Differenz zwischen schnellster & langsamster Nachricht)
- Γ ... Drift Offset

Central Master Algorithm

- Ein Master Knoten, der den Takt vorgibt
 - ⇒ Sendet periodisch Synchronisation-Nachrichten inkl. lokaler Uhr
- Alle anderen Knoten passen sich an:
 - Speichern der lokalen Zeit bei Erhalten der Nachricht
 - Differenz master clock - local clock berechnen
 - Entsprechendes Update der lokalen Zeit
- Precision des Algorithmus: $T_{\text{central}} = T + \epsilon$, $\epsilon = d_{\text{max}} - d_{\text{min}}$
 - ↑ Drift Offset
 - ↳ Zeitdelay bei Kommunikation
- Problem: Single-Point-of-Failure

Distributed Clock Synchronization

- Wird bei sicherheitskritischen Systemen verwendet, um kein Single-Point-of-Failure zu haben
- 3 Phasen:
 - Knoten senden einander die global-time Counterwerte
 - Jeder Knoten analysiert die Informationen (error detection) und berechnet mittels Konvergenzfunktion einen Correction Term für eigenen global-time Counter
- 2 beliebige Algorithmen:
 - Averaging Algorithm
 - Fault-Tolerance Average Algorithm

Averaging Algorithm

- Die Knoten nehmen den Mittelwert aller Uhrenwerte
- Problem bei Byzantine Clocks ⇒ Ab $N \geq 3k+1$ Uhren können k byzantine Clocks bei N Clocks toleriert werden

Fault-Tolerant Average (FTA) Algorithm

- Eliminiert bei N Uhren die k kleinsten und k größten Uhrenwerte und bildet den Mittelwert des Rests ($=N-2k$)
 - Error-Term eines byzantinischen Errors: $E_{\text{byz}} = \frac{\pi}{N-2k}$
 - Konvergenzfunktion $\Phi(N, k, \varepsilon) = k \cdot \frac{\pi}{N-2k} + \varepsilon$
 - Precision $\pi(N, k, \varepsilon, \Gamma) = (\varepsilon + \Gamma) \cdot \frac{N-2k}{N-2k} = (\varepsilon + \Gamma) \cdot \mu(N, k)$
 $= \pi_{\text{master}}$
- $\Rightarrow \mu(N, k)$ gibt an, wieviel schlechter die Precision vom FTA im Vergleich zu der Precision vom Central Master Alg. ist
- $\mu(N, k)$... Byzantinischer Error Term

Interactive Consistency Algorithm

- Basiert auf FTA Algorithm
 - Führt noch eine Kommunikationsrunde ein
- \Rightarrow Jede Uhr teilt zusätzlich seine Sicht der Werte aller anderen Uhren
- \Rightarrow So können byzantinische Knoten eliminiert werden
- $\Rightarrow \mu(N, k) = 1$

Limits der Uhrensynchronisation

- Beste erreichbare Precision π_{opt} ist: $\pi_{\text{opt}} = \frac{\varepsilon}{1 - \frac{1}{N}}$

Qualitätsbestimmende Parameter der Uhrensynchronisation

- Drift Offset $T = 2 \cdot p \cdot R_{\text{int}}$
- Delay jitter $\varepsilon = d_{\text{max}} - d_{\text{min}}$

- Byzantine failures: rare events
- Uhrensynchronisations-Algorithmen (Effekt im Vergleich zu ϵ klein)

Drift Offset klein halten

- Klein halten durch das Minimieren der Drift Rates der Uhren.
 - Rate Master mit präziser Clock im Cluster vorhanden
 - Rate Correction (= Clock Rate anpassen) bei lokalen Uhren, um an Rate Master anzupassen
 - State Correction (= Wert des Counters anpassen \rightarrow Sprung!) beim FTA-Algorithmus einsetzen, um Fehler in Rate Correction bei den lokalen Uhren auszugleichen

Delay Jitter

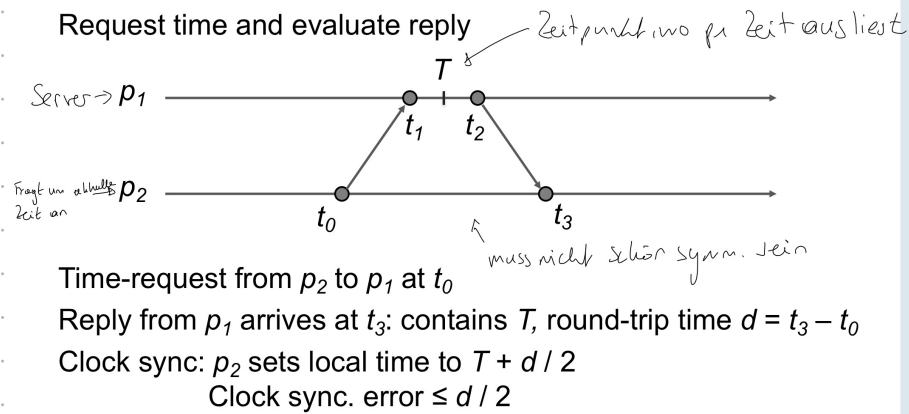
- Hängt stark davon ab, wo der Zeitstempel generiert & interpretiert wird:
 - Application Software Level: $50\mu s \dots 5ms$
 - Operating System Kernel: $10\mu s \dots 100\mu s$
 - Hardware: communication controller: $< 10\mu s$

Qualitätsparameter einer Global Time Base

- Precision
- Accuracy
- Fault-Tolerance: Anzahl & Typen der Fehler, die toleriert werden
- Blackout Survivability: Tolerierte Blackout-Dauer ohne Synchronisationsverlust

Cristian's Algorithm

◦ Wird verwendet bei **externer** Clock-Synchronisation:



Network Time Protocol (NTP)

◦ Basiert auf **Idee** von Cristian's Algorithmus:

◦ **Hierarchische** Time Server.

- Klasse 1: Verbunden mit Atomuhren, GPS ...

- Klasse 2: Holen sich Zeit von höherer Klasse, Synchronisation untereinander

- Klasse 3: Holen sich Zeit von höherer Klasse, Synchronisation untereinander usw.

◦ Clock Correction wird basierend auf statistischer Analyse von mehreren Clock Readings durchgeführt

◦ Precision Time Protocol (**PTP**) basiert auf NTP, verwendet Hardware Support für das Generieren von Zeitstempel, um jitter ϵ klein zu halten

Time Standards

International Atomic Time (TAI)

- physikalischer Standard
- Sekunde ist über Schwingungen der Strahlung vom Cesium Atom definiert
- misst seit 1.1.1958 00:00:00
- chronoskopisch (= keine Sprünge)
- gut für Intervall, aber nicht für Tageszeit

Universal Time Coordinated (UTC)

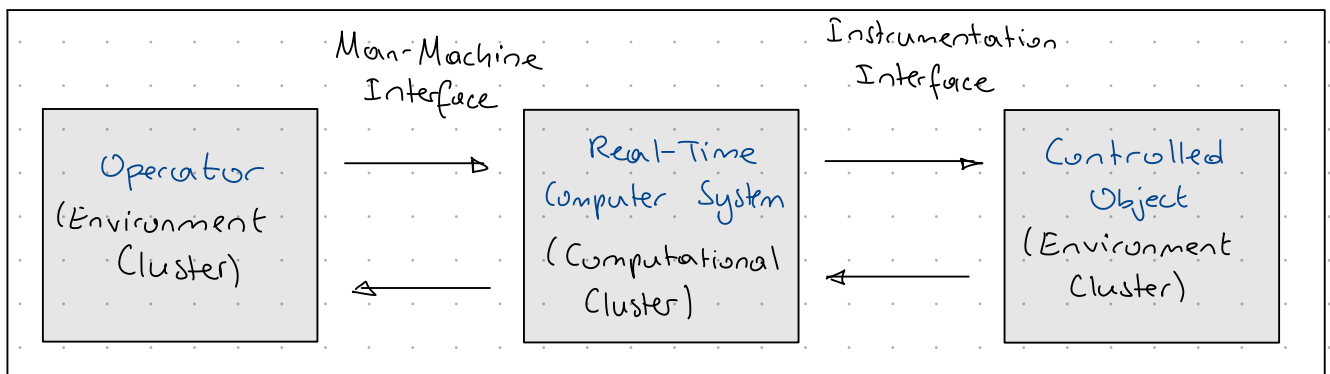
- Astronomischer Standard
- Basis für die Alltagsuhrzeit
- Dauer einer Sekunde entspricht TAI-Standard
- Enthält ab und zu Schaltsekunden, um astronomischen Phänomenen zu entsprechen

3a RTS-Modeling: Clusters, Components, Interfaces

RTS-Modell: Essentielle Elemente

- Repräsentation der Echtzeit
- Semantik der Datentransformationen bzw. Größen
- Dauer der Operationen

Struktur eines Echtzeitsystems



- RTS: Controlled Object + Computer System + Operator
- Cluster: Subsystem eines RT-Systems mit hoher innerer Konnektivität
- Node: Hardware-/Software-Einheit mit bestimmter Funktion
- Task: Exekution eines Programms innerhalb einer Komponente

Computational Cluster

- Menge von Komponenten, die zusammenarbeiten und:
 - ein spezifisches Service bieten
 - über eine einheitliche Repräsentation an Infos verfügen
 - eine hohe innere Konnektivität haben
 - wenig Interfaces zu anderen Clusters haben

Component

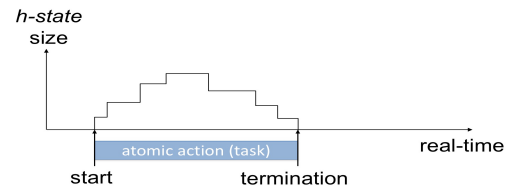
- Ein Baustein eines größeren Systems (Clusters)
- Liefert ein klar definiertes Service
- Spezifiziertes Interface, das In-/Outputs und zeitliche Relationen zwischen ihnen festlegt
- Integration darf kein Wissen über interne Details benötigen
- RT-Component = Komplettes Computer System, das aus Hardware UND Software besteht
- Eine Komponente besteht aus:
 - Hardware
 - Betriebssystem (vorinstalliert)
 - State:
 - i-state (initialization state): statische Datenstruktur, d.h. Anwendungsprogrammcode, Initialisierungsdaten
 - h-state (history state): dynamische Datenstruktur, die Infos über aktuelle und vergangene Berechnungen enthält
- Um einen systemweit konsistenten State haben zu können, ist ein systemweit konsistentes Verständnis von der

Zeit notwendig
 ⇒ Sparse Time Base notwendig

History State

- Beinhaltet alle nötigen Infos, um einen initialisierten Knoten oder Task(i-state) zu einem bestimmten Zeitpunkt zu starten
- Größe ist zeitlich veränderlich
- Relatives Minimum direkt nach Ende der Berechnung eines Tasks
- Sollte klein sein bei Reintegration
 ⇒ g-state (ground state)
- g-state: Ist ein minimaler h-state zu einem Zeitpunkt, wo kein Task aktiv ist und keine Nachricht unterwegs ist.

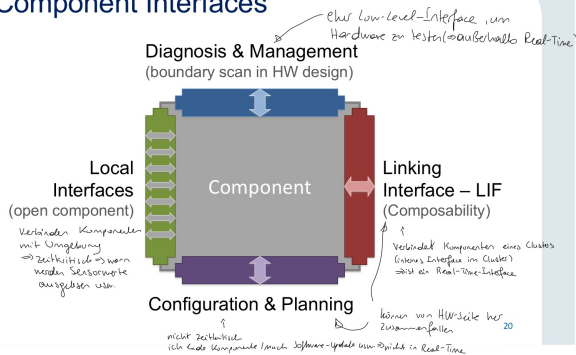
History-State Size during Atomic Action



Interface

- Ist die gemeinsame **Boundary** zwischen 2 Systemen.
- Charakterisiert durch:
 - Data Properties: Struktur & Semantik der Daten, die über das Interface gehen
 - Temporal Properties: Zu erfüllende zeitliche Bedingungen
 - Control Properties: Protokoll bzw. Ablauf der Schritte, die zum Datenaustausch verwendet werden

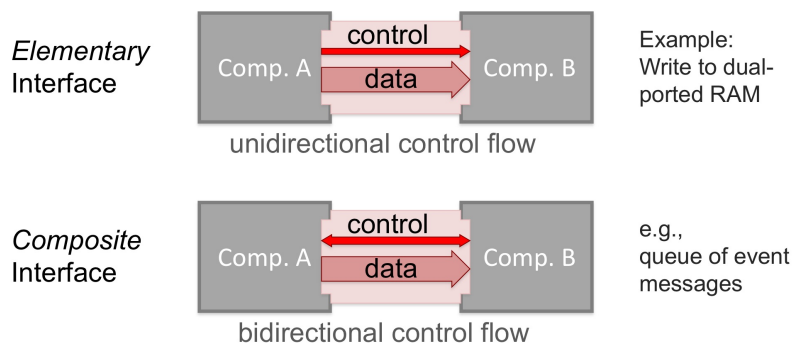
Component Interfaces



Component Interfaces

- Realtime Service oder **Linking Interface**:
 - Verbindet die Komponenten eines Clusters
 - Ist ein Realtime-Interface
 - Typischerweise periodischer Nachrichtenaustausch
- **Diagnosis & Management Interface**:
 - Eher ein Low-Level-Interface, um Komponente zu testen
 - kein Realtime-Interface (⇒ nicht zeitkritisch)
- **Configuration & Planning Interface**:
 - Komponente wird initialisiert /geupdatet usw.
 - kein Realtime-Interface
- **Local Interfaces**
 - Verbinden die Komponenten mit der Umgebung
 - Realtime-Interface

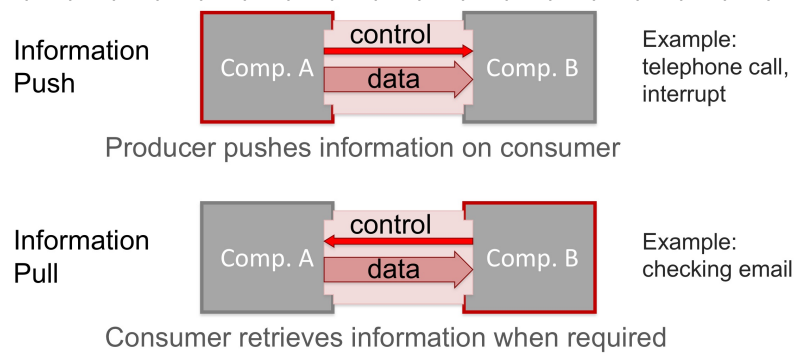
Interface Control Strategies



Elementary interfaces are simpler!

Information Push and Pull

- Sind Arten von **Elementary Interface**
- Information **Push**: Komponente, die schreiben will, gibt Infos weiter, sobald der Output vorhanden ist (Kontrollrichtung **gleich** Datenrichtung?)
- Information **Pull**: Komponente, die Daten erhalten will, fragt bei schreibender Komponente an, sobald Input benötigt wird (Kontrollrichtung **ungleich** Datenrichtung?)
- In **Echtzeitkomponenten** ist eine **Kombination** aus beiden erwünscht.



Temporall. Firewall

- Komponente macht Information Push, wenn sie schreiben will, und Information Pull, wenn sie Daten braucht
- Temporal Firewall = Interface, dass **keine Zugriffe von außen** auf die Komponente erlaubt



component with two temporal-firewall interfaces

⇒ Es werden nur **elementare** (=unidirektional) Interfaces nach außen verwendet. **Kontrollfluss** geht immer von der Komponente **weg**

Component Categories

- Closed Component:
 - Linking Interface zu den anderen Komponenten
 - Kein lokales Interface zu environment
 - ⇒ keine Sensoren/Aktoren
- Semi-Closed Component:
 - closed component
 - time-aware (⇒ Zeit hat äußeren Einfluss)
- Open Component:
 - Local real-world Interface
 - Reagiert auf In-Outputs vom Umfeld
- Semi-Open Component:
 - Holt sich Umgebungswerte selber (zB sampling)
 - ⇒ Also keine Kontrolle von außen

Shared Hardware bei RTS

- Folgende Möglichkeiten und Probleme:
 - Memory ⇒ erfordert Mutual Exclusion
 - Bus ⇒ erfordert Arbitration (immer nur von 1 Komponente verwendbar)
 - Cache ⇒ Pollution möglich (Daten verschiedener Komp kollidieren)
- ⇒ Kontrolle von außen auf Komponenten (und auf Timing) nötig!
- ⇒ Fehler propagieren viel leichter
- ⇒ Statt Shared Hardware muss das Echtzeit-Interface ein Message-Interface sein (message = atomare Einheit)

Connector System

- Löst Interface-Property Mismatches
- Ist quasi ein Adapter zwischen 2 Interfaces, die in irgendeiner Form inkompatible Eigenschaften (zB bei Time Base, Datenformat, Daten-Endianmess usw.) haben

Informationsrepräsentation bei Interfaces

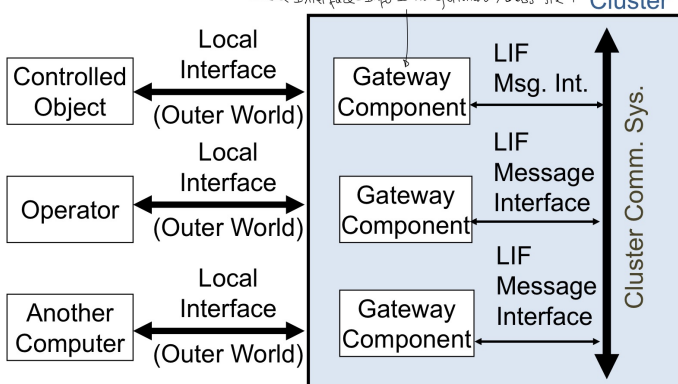
- Jedes Interface knüpft an 2 Subsysteme
- Information kann verschieden encodet sein in diesen Subsystemen:
 - **Abstract** Interface: Unterschiede in Info-Repräsentation unbedenklich, solange die semantischen Inhalte & die zeitlichen Eigenschaften der Informationen durch das Interface maintained werden
 - **Low-Level** Interface: Info-Repräsentation ist relevant

Message Interface vs. Real-World Interface

- World Interface: Konkretes Low-Level-Interface zu Devices
- Message Interface: Internes abstraktes Interface für Nachrichten-basierte Kommunikation innerhalb eines Clusters
- Resource Controller: Verbindet Real-World mit Message Interface und versteckt das physische Interface, womit man bei der standardisierten Info-Repräsentation bleiben kann
 - ⇒ "Information Transducer"
 - ⇒ Ist eine Art von Gateway

World vs. Message Interface

Local Interface-Info so transformieren, dass sie für Cluster nutzbar ist



3b RTS-Modeling: RT Entities, Images, Objects

RT-Entity

- Ist eine Größe, die für eine Echtzeitanwendung relevant ist
- Zustandsvariable
- Entweder im Computersystem oder in der Umgebung
- Zustand ändert sich mit der real-time
- kann kontinuierlich oder diskret sein
- Bsp: Position eines Schalters
- Eigenschaften:
 - statisch: Name · Typ · Zulässiger Wertebereich
Max. Änderungsrate
 - dynamisch: Aktueller Wert an einem bestimmten Zeitpunkt

Sphere of Control

- Jede RT-Entity ist in der Sphere of Control (SOC) eines Subsystems, das autorisiert ist, den Wert der RT-Entity zu setzen
- Außerhalb der SOC kann eine RT-Entity nur beobachtet werden

Observation

- Ist das Erfassen von Infos über den State einer RT-Entity
- Observation = $\langle \text{Name, Time, Value} \rangle$
- Observations werden über Nachrichten transportiert

Continuous RT-Entity

- Werte sind immer definiert

Discrete RT-Entity

- Diskrete Wertemenge
- Wert bleibt konstant über ein Intervall \rightarrow State
- Change event kann nicht observed werden \rightarrow neuen state observen
- Außerhalb der Intervalle undefinierter Wert

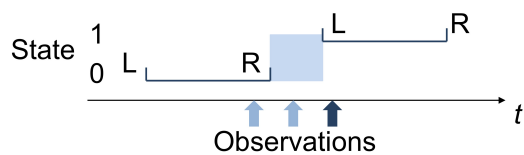
Arten von Observation

◦ State Observation:

- Liefert absoluten Wert, enthält also direkt den Zustand der RT-Entity
- Observation Time = Zeitpunkt, an dem RT-Entity gesampled wurde

◦ Event Observation:

- Wert beschreibt Differenz zwischen altem und neuem Zustand
- Observation Time: Zeitpunkt vom L-Event des neuen Zustands



(= 1. Observation, die den neuen Zustand wahrgenommen hat)

RT-Image

- Ist ein Abbild einer RT-Entity
- Gültig für jeden Zeitpunkt, an dem es eine akkuratere Repräsentation der korrespondierenden RT-Entity in Wert und Zeit darstellt
- Kann auf Observation oder State Estimation basieren
- Kann in Rechner gespeichert sein oder sich in Umgebung manifestieren

RT-Object

- Container für ein RT-Image oder RT-Entity im Computersystem
- Ist assoziiert mit einer real-time clock mit Granularität t_k

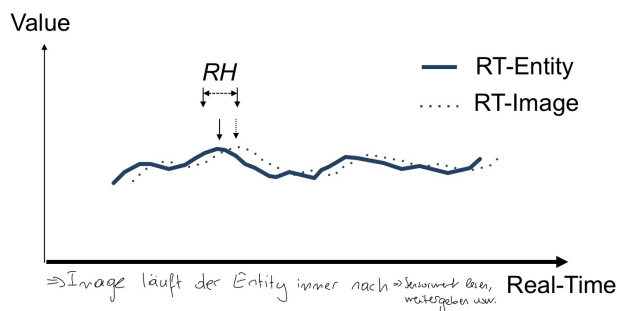
Temporal Accuracy

- Beschreibt wie lange man den Wert des RT-Images verwenden darf
- Temporal Accuracy Interval d_{acc} folgt aus der Dynamik der zugehörigen RT-Entity
- Recent History RH_i zur Zeit t_i :
 - Geordnetes Set von Punkten $\langle t_{i-k}, \dots, t_{i-1}, t_i \rangle$
 - Länge der Recent History ist $d_{acc} = t_i - t_{i-k}$
- Annahme: RT-Entity wurde zu allen Zeitpunkten $\langle t_{i-k}, \dots, t_{i-1}, t_i \rangle$ observed
- ⇒ RT-Image ist temporally accurate zum aktuellen Zeitpunkt t_i , wenn es in der Recent History einen Zeitpunkt t_j gibt, für den gilt, dass der Wert des Images zum Zeitpunkt t_i irgendwann mal in der Recent History vorgekommen ist, also:

$$\exists t_j \in RH_i : \text{Value}(\text{RT-Image at } t_i) = \text{Value}(\text{RT-Image at } t_j)$$

RT-Image Error

Temporal Accuracy of RT-Objects



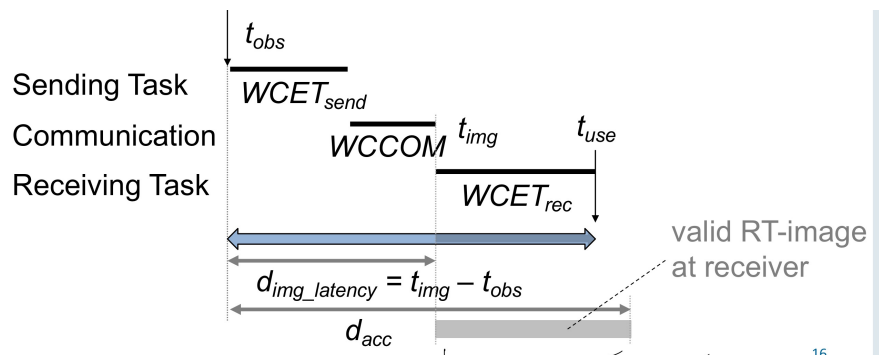
For an RT-object, updated by observations, there will always be a delay between the state of the RT-entity and that of the RT-object.

12

- RT-Image Error = Abweichung zwischen Wert zu Observation und Verwendung: $\text{error}(v(t_{use}), t_{obs}) = v(t_{use}) - v(t_{obs})$
- Worst-Case-Approximation: $\text{error}_{\max}(v) \leq \underbrace{\max\left(\frac{d}{dt} v(t)\right)}_{\text{max. Veränderungsrate}} \cdot \underbrace{d_{acc}}_{\text{max. Zeitabweichung}}$

Phase-aligned Transactions

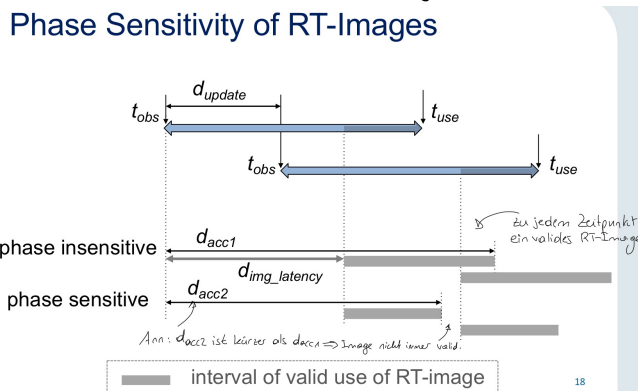
- Zeitliches Aufeinanderabstimmen der Zeitpunkte des Lesens, Sendens und Verarbeitung von Werten. (Schedules auf jedem Knoten so ausrichten, dass sie aufeinander abgestimmt sind.)
- Wenn sich die RT-Entity schnell ändert, muss d_{acc} klein sein
- Phase-aligned transactions garantieren, dass der observierte Wert temporally correct ist, also: $t_{use} - t_{obs} \leq d_{acc}$



Phase (In)Sensitivity von RT-Images

- Relevant bei:
 - Periodisches Update des RT-Image mit d_{update}
 - Kurze Temporal Accuracy
 - Phase-aligned Transactions

⇒ Wann kann man das RT-Image verwenden?



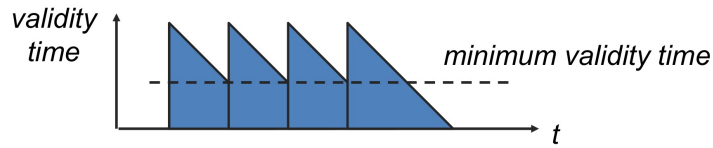
◦ Phase-Insensitive / Parametric RT-Image:

- Zu jedem Zeitpunkt steht ein gültiges RT-Image zur Verfügung
- $d_{acc} > d_{img_latency} + d_{update}$

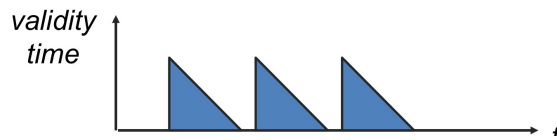
Phase-Sensitive RT-Image:

- Es gibt Intervalle, in denen es kein gültiges RT-Image gibt
- $d_{acc} \leq d_{img-latency} + d_{update}$ und $d_{acc} > d_{img-latency}$

Phase-insensitive RT-image



Phase-sensitive RT-image



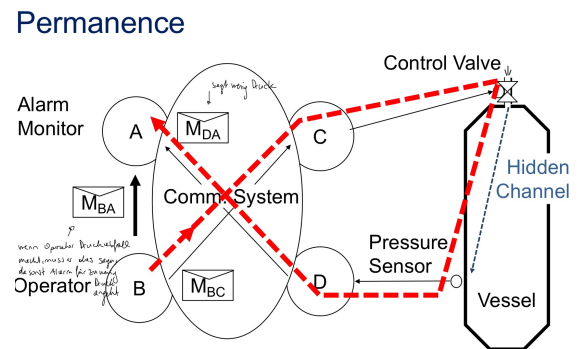
State Estimation

- Wenn RT-Image phase-sensitive ist, gibt es nicht immer gültige Werte
⇒ Aktueller State der RT-Entity kann approximiert werden
- Wird periodisch berechnet innerhalb des RT-Objekts aus den zuvor beobachteten Werten
- Basiert auf Modell der Dynamik der RT-Entity
- $v(t_{use}) = v(t_{obs}) + (t_{use} - t_{obs}) \cdot \frac{d}{dt} v(t_{obs})$
- Timing requirements: Um das Nachrichtendelay zu kompensieren, benötigt ein State-Estimation-Programm:
 - den Zeitpunkt der Observation
 - die geplante Zeit der Actuation (= wann ein Output erfolgen soll)
- Qualität der Estimation hängt ab von:
 - Precision der Clock Synchronisation
 - Latenz & Qualität der Latenzmessung
 - Qualität des State-Estimation-Modells

Permanence

- Eine Nachricht M_i ist permanent im Objekt O , sobald alle vor M_i gesendeten Nachrichten M_{i-1}, M_{i-2}, \dots bei O angekommen sind
⇒ Erst dann darf auf M_i reagiert werden!

Bsp:



Action Delay

- Ist das Intervall zwischen dem Zeitpunkt, an dem eine Nachricht geschickt wird, und dem Zeitpunkt, an dem der Empfänger weiß, dass es permanent ist

- Verteilte RT-Systems ohne global time base:

$$\text{max. action delay} = d_{\max} + \epsilon = 2 d_{\max} - d_{\min}$$

↑
max. Latenz

⇒ Empfänger wartet auf langsamstmögliche weitere Nachrichten

- RT-Systems mit global time base:

$$\text{max. action delay} = d_{\max} + 2g$$

↑
1g-Senderfehler + 1g-Empfängerfehler

⇒ Empfänger wartet nur $d_{\max} +$ Zeitstempelgenauigkeit

Accuracy vs. Action Delay

- Accuracy = wie lange ist eine Message gültig
- Action Delay = ab wann darf man die Message verwenden
- Es muss gelten: Action Delay $<$ d_{acc}
⇒ Wenn das nicht gilt ⇒ State Estimation nötig!

4a RT-Communication: Principles & Protocol Types.

Vorteile von verteilten RT-Systemen

- Composability: Konstruktionen von neuen Systemen aus vorhandenen Komponenten
- Intelligent Instrumentation: Integration von Sensoren / Aktuatoren, lokale Berechnung & Kommunikation auf einem einzigen Die
- Reduction of wiring harness: Reduktion der Verkabelung
- Avoidance of a Single-Point-of-Failure: Bieten Redundanz

Ansprüche eines RT-Communication Systems

- Determinism: Timeliness, Predictability, Testability, ...
- Composability: Zeitliche Kapselung (Kommunikation & Berechnung getrennt) \Rightarrow Simple Flow Control
- Dependability: Error-Containment von fehlerhaften Knoten
- Support für verschiedene Applikationen
- Nicht allzu teuer
- Kompatibilität mit Ethernet Standard \Rightarrow Ist weit verbreitet

Message

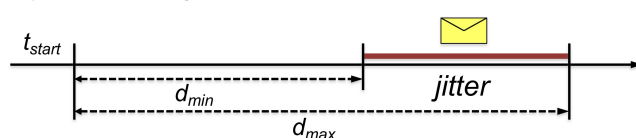
- Ist eine **atomare** Datenstruktur, **transportiert** von einem Sender an Empfänger

Transmission timing

t_{start} ... start instant at sender

d_{min}, d_{max} ... minimum / maximum transmission delay

$d_{max} - d_{min}$... **jitter** of the transmission channel



Flow Control

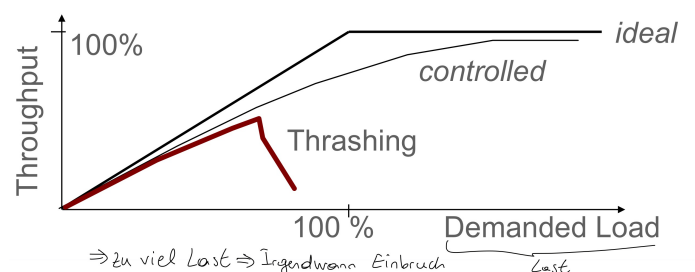
- Steuert den Informationsfluss zwischen kommunizierenden Partnern
- Sender darf nicht schneller als Empfänger sein
⇒ Infos sollen nur so schnell geschickt werden, wie sie vom Empfänger verarbeitet werden können
- Explicit Flow Control: Flow Control durch Nachrichtenaustausch
- Implicit Flow Control: Vorher vereinbart

Explicit Flow Control

- „Explicit“ = Sender & Empfänger machen sich den Informationsfluss zur Laufzeit untereinander aus
- Sender schickt Nachricht & wartet auf ACK des Empfängers
⇒ PAR... Positive Acknowledgement or Retransmission
- Fehlererkennung durch Sender
- Sender ist in SOC des Empfängers (Empfänger kann Sender verlangsamen)

Thrashing

- Explicit Flow Control ist unter Last nicht stabil?
⇒ Kollisionen, weil mehrere Nachrichten gleichzeitig
⇒ Viele Nachrichten müssen wiederholt gesendet werden
⇒ Zu viel Last ⇒ Thrashing ⇒ Durchsatz sinkt massiv



Implicit Flow Control

- Es gibt a priori eine Vereinbarung, mit welcher Rate Nachrichten übertragen werden
- Vereinbarte Rate muss für Empfänger akzeptabel sein
- Fehlererkennung durch Empfänger
 - ⇒ kein ACK ⇒ unidirektional
- Gut geeignet für Multicast-Kommunikation

Limits in Protocol Design

- Temporal Guarantees:
 - Bei offenen Kommunikationsprotokollen kann man keine worst case Zeiten angeben
 - ⇒ Einschränkungen notwendig ⇒ Lösung: Konfliktfreies Sendemuster (Schedule) im Voraus definieren (=closed system)
- Synchronization Domain:
 - Es darf nur eine Kontrollinstanz geben
 - Synchronization erreichbar durch:
 - Referenz zu einer global time base
 - Referenz zu einer führenden Datenquelle (Koordinator)
- Error Containment:
 - Zeitliche Fehler durch einen fehlerhaften Komponenten müssen erkannt werden ⇒ Ein fehlerhafter Knoten darf nicht die Kommunikation stören

- Konsistente Ordnung von Events:
 - Sparse time base ist notwendig für Konsistenz

Kategorien von RT-Kommunikationsprotokollen

- Event-triggered (ET) Protocols:
 - Kommunikation wird von außen gesteuert
 - nur eingeschränkt für RT-Systeme brauchbar
- Rate-Constrained (RC) Protocols:
 - Datenrate begrenzt \Rightarrow Implicit Flow Control
 - Knoten bekommen nur gewisse Bandbreiten \Rightarrow Zeitliche Garantien
- Time-triggered (TT) Protocols:
 - Nachrichten werden zeitabhängig in geplantem Muster versendet

ET-Protocols

- Event beim Sender triggert das Senden der Nachricht
- Erfordert Explicit Flow Control:
 - Fehlererkennung durch Sender
 - Erhöhter Traffic durch ACK-Messages
 - Sender ist in SOC des Empfängers
- Hohe max. execution time & Jitter wegen möglicher Retransmission
- Keine zeitliche Kapselung
- Bsp: CSMA/CD, CAN

RC-Protocols

- Jeder Knoten erhält eine **minimale** Bandbreitengarantie
- Sender passen Nachrichtenrate an **verfügbare** Bandbreite an
- Max. Latenz einer Nachricht kann garantiert werden, solange die garantierte Bandbreite **nicht** überschritten wird
- **Keine** global time
- Bsp: Token protocol, AFDX

Traffic Shaping

- Traffic Metering = **Überprüfung**, ob Bandbreite eingehalten wird
- Traffic Shaping = **Maßnahmen** treffen, damit Bandbreiten **nicht** überschritten werden (z.B. durch Switches, Buffering)

Token Bucket Algorithm

- Ist eine **Möglichkeit**, Traffic **Shaping** zu betreiben
 - Ziel: **Nicht mehr** Nachrichten als **erlaubt** pro Zeiteinheit zulassen
 - Tokens kommen mit gewisser Rate r in einen „Kübel“ mit Kapazität C (überschüssige Tokens gehen verloren)
 - Um eine Nachricht mit **n bytes** senden zu dürfen, müssen **n Tokens** im Kübel sein
- ⇒ Die Tokens werden dann vom Kübel entfernt & die Nachricht über das Netzwerk weitergeleitet

TT-Protocols

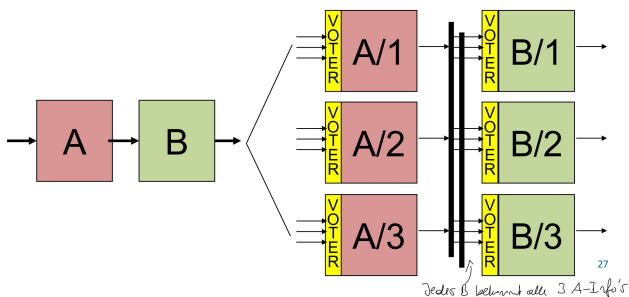
- **Implicit** Flow Control
- Fortschritt der globalen Zeit triggert das Senden von Nachrichten entsprechend einem Schedule \Rightarrow **State Message**!
- Sendezeiten sind im **Voraus** bekannt
- Fehlererkennung durch **Empfänger**
- **unidirektional**, da keine ACK-Nachrichten

Idempotenz

- Eine Menge an Nachrichten ist **idempotent**, wenn das Empfangen von mehreren Nachrichten dieses Sets den **gleichen Effekt** wie das Empfangen von **nur einer** Nachricht hat
- Duplizierte **State** Messages sind idempotent, zB „Es hat 20°“
- Duplizierte **Event** Messages sind nicht idempotent, zB „Temp. ist um 2° gestiegen“
- Idempotente Nachrichten können **für Redundanz** verwendet werden.
 \Rightarrow Vereinfacht das Design von fehlertoleranten Systemen

TMR

- TMR... Triple Modular Redundancy
- Ist eine Möglichkeit, um Node Failures zu **umgehen**



\Rightarrow 3 Instanzen je Knoten \Rightarrow Durch **Voting** wird entschieden, welcher Wert verwendet wird

- Leicht zu realisieren durch **Broadcast** bei TT-Protokoll

End-to-End Feedback

- Zuverlässige Kommunikation alleine ist nicht genug, denn man kann nicht dadurch darauf schließen, dass die Nachrichten das bewirken, was sie sollen (zB ein Aktuator fällt aus)
- ⇒ Erfolgreiche Nachrichtenübermittlung garantiert nicht korrekten Service!

◦ End-to-End Feedback:

- Semantisches Feedback auf Anwendungs-Level, dass ein Subsystem seinen Zweck erfüllt
- zB zusätzlicher Sensor, der den erwünschten Effekt am Endpunkt kontrolliert
- Hohe Error-Detection Coverage
- Zusätzliche Error-Detection in Protokollen auf anderen Leveln:
 - Notwendig wenn die Kommunikation eine Schwachstelle
 - Vereinfacht die Diagnose

3 Levels einer RT-Kommunikationsarchitektur

- Fieldbus: Verbindet Knoten mit Sensoren
 - real-time
 - billig
 - robust
- Real-Time Bus: Verbindet Knoten mit RT-Cluster
 - real-time
 - fault-tolerant
- Backbone Network: Verbindet Clusters untereinander
 - non real-time

Kommunikations-Performance Kriterien

- Bandbreite
- Propagation Delay
- Bitlänge
- Protocol Effizienz

Bandbreite

- Anzahl an Bits, die den Kanal in einer Zeiteinheit durchlaufen können
- Abhängig von:
 - Physikalischen Charakteristiken des Kanals (zB single wire, twisted pair)
 - Umgebung (zB elektromagnetische Störungen)

Propagation Delay

- Die Dauer, in der ein Bit vom Anfang bis zum Ende des Kanals wandert
- Abhängig von:
 - Transmission speed im Kanal
 - Länge des Kanals
- zB: Licht im Kabel $\approx 2 \cdot 10^8 \frac{m}{s} \Rightarrow$ Ein Signal wandert ca. $200 \frac{m}{\mu s}$
 \Rightarrow Propagation delay bei 1km-Kabel = $5 \mu s$

Bit Length

- Anzahl der Bits, die während eines Propagation Delays gleichzeitig durch den Kanal wandern können

$$\text{Bit Length} = \frac{\text{Bandbreite}}{\text{Transmission Speed}} \cdot \text{Kanallänge}$$

$$\begin{aligned} \text{◦ Bsp: Bandbreite} &= 100 \text{ M} \frac{\text{bit}}{\text{s}} \\ \text{Kanallänge} &= 1000 \text{ m} \end{aligned} \quad \Rightarrow \text{Bitlänge} = \frac{10^8 \frac{\text{bit}}{\text{s}}}{2 \cdot 10^8 \frac{\text{m}}{\text{s}}} \cdot 1000 \text{ m} = 500 \text{ bit}$$

Protocol Efficiency

- Max. Anteil der Kanal-Bandbreite, die eine Anwendung für ihre Nachrichten verwenden kann
 - Zwischen einzelnen Messages muss mindestens das Propagation Delay gewartet werden, um Collisions zu verhindern
 - Data efficiency $d_{eff} < \frac{\text{Bit length einer Message}}{\text{Bit length des Kanals}}$
 - Bsp: Bandbreite = $100 \text{ M} \frac{\text{bit}}{\text{s}}$
Kanallänge = 1000 m
Bit length einer Message = 100 bit
} \Rightarrow Bit length = 500 bit
} $\Rightarrow d_{eff} < \frac{100}{600}$
 $= 16,67\%$
- \Rightarrow Bandbreite erhöhen hilft meist nicht \Rightarrow Propagation Delay ist das Entscheidende!

4b RT-Communication: Communication Protocols

Medium Access Protocols

- CSMA/CD
- CSMA/CA
- Token Bus
- Minislotting
- Central Master
- TDMA

Maximum Protocol Execution Time (d_{max})

- d_{max} hängt ab von:
 - Protokoll-Stack des Senders (inkl. Error Handling)
 - Nachrichten-Scheduling Strategie des Senders
 - Medium Access Protocol
 - Transmission Time (hängt vom Übertragungsmedium ab \Rightarrow im Wesentlichen von der Länge des Kanals)
 - Protokoll-Stack des Empfängers
 - Task-Scheduling beim Empfänger

CSMA/CD

- CSMA: Carrier Sense Multiple Access
- CD: with Collision Detection
- Eigentlich nicht für RTS konzipiert, aber oft im RT-Kontext verwendet
- Communication Controller, der eine Nachricht versenden will, prüft den Bus auf Traffic \Rightarrow Wenn kein Carrier-Signal erkannt wird, wird die Nachricht versendet (\Rightarrow Carrier Sense)
- Collision wenn mehrere Knoten gleichzeitig senden
- Falls Collision:
 - Jam-Signal wird ausgesendet & die Nachricht nach einer zufälligen Zeitdauer neu gesendet
 - Begrenzte Anzahl an Neuversuchen
- Bsp: Ethernet (Bus Topology, Shared Medium)

CSMA/CA

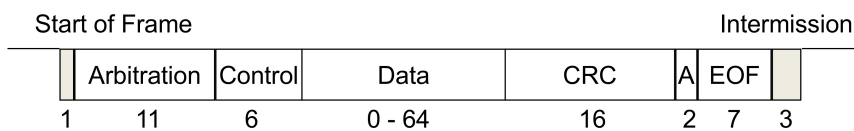
- CA: with Collision Avoidance
- MAP, das auch für RTS geeignet ist
- Carrier Sense wie bei CSMA/CD
- Jeder Nachricht wird ein Identifier hinzugefügt, der die Priorität kodiert
 - \Rightarrow Bit Arbitration: dominant bit (zB 0) & recessive bit (zB 1)
- Bei Collision:
 - Die Nachricht mit dem dominanten Bit wird durchgelassen
 - \Rightarrow Bit für Bit wiederholt, bis ein recessive Bit nachgeben muss
 - Sender erkennen das \Rightarrow Erfolgreicher Knoten sendet nochmal
- Für Knoten mit höchster Priorität stabiles Zeitverhalten, für andere wird die Latenz variieren

- Jedes Bit muss sich also **zuerst stabilisieren**, d.h. das Propagation Delay muss **deutlich kleiner** sein als die Zeit, für die ein Bit anliegt

CAN-Control Area Network

- Bus Arbitration **wie bei CSMA/CA**
- Communication Speed: $1 \text{ M} \frac{\text{bit}}{\text{s}}$
- Kanallänge: $\sim 40 \text{ m}$
- Standard Format: 2032 Identifiers, 11 bit arbitration field
- Extended Format: $> 10^8$ Identifiers, 32 bit arbitration field

Frame format



Token Bus / Ring

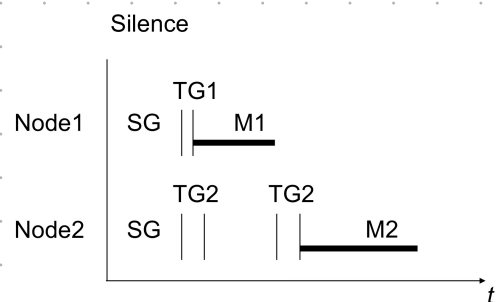
- Basiert auf Token: Spezielle Control Message, die **erlaubt**, zu senden
- Nur der Knoten, der den Token hat, darf senden
- Token wird von Knoten zu Knoten **weitergereicht**, in der Regel im logischen oder physischen Kreis
- Timing-Parameter:
 - Token-Hold-Time: Max. Dauer, in der ein Knoten den Token hat
 - Token Rotation: Max. Dauer für eine vollständige Rotation des Tokens
- Problem bei Hard RT: **Token loss**
 - Token **Recovery**: Knoten erzeugt neuen Token nach random Timeout
 - ⇒ Kann zu **Collisions** führen, wenn das mehrere Knoten machen
 - ⇒ Retry

Minislotting

- Die Zeit ist in eine Folge von Minislots eingeteilt
- Die Dauer eines Minislots ist größer als das maximale Propagation Delay
- Jedem Knoten wird eine bestimmte Anzahl von Minislots zugeteilt, die ablaufen müssen, bevor der Knoten senden darf
- Erlaubt die Priorisierung von Nachrichten bzw. Knoten
- Wenn höher priorisierter Knoten nichts zu senden hat, können Knoten senden, sonst müssen sie warten
- Bsp: ARINC 629

ARINC 629

- Ist ein Minislotting-Protokoll & wird im Aerospace verwendet
- Medium Access wird gesteuert durch Intervalle:
 - TG: Terminal Gap, für jeden Knoten anders, länger als Propagation Delay, bestimmt die Sende-Reihenfolge (node-spezifische Minislots)
 - SG: Synchronization Gap, länger als der längste TG, startet eine neue Kommunikationsrunde



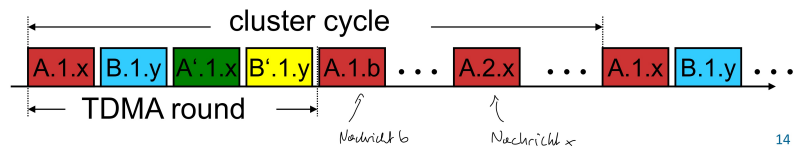
Central Master

- Ein zentraler Masterknoten ("Bus Arbitrator"), der den Zugriff auf den Kanal kontrolliert
- Beim Ausfall des Masterknotens muss ein anderer Knoten übernehmen
- Bus Arbitrator sendet periodisch per Broadcast Variablenamen
- Knoten, der diese Variable produziert, sendet ihren Wert

- Falls noch Zeit ist, kann er auch andere Daten mitsenden
- Bsp: FIP

TDMA

- TDMA: Time Division Multiple Access
- Ist ein zeitgesteuertes MAP
 - ⇒ Erfordert eine fehlertolerante global time base
- Zeit ist statisch geteilt in Zeitslots
- Noch vor der Laufzeit wird ein zyklisches Sendeschema definiert
- TDMA-Round: Periode, in dem jeder Knoten einmal vorkommt
- Cluster Cycle: Globale Sequenz von TDMA-Rounds, die periodisch wiederholt wird (Knoten können in verschiedenen TDMA-Rounds verschiedene Nachrichten senden)



14

Time-Triggered Protocol

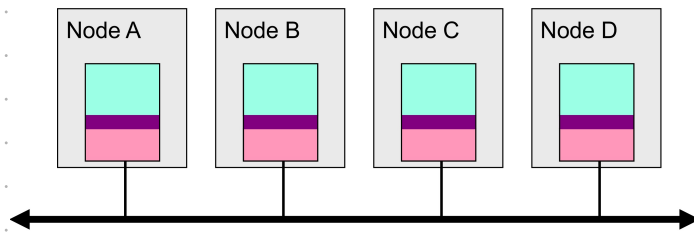
- Ist ein integriertes, zeitgesteuertes Protokoll für RT-Systems
- TTP sieht einen redundanten Aufbau (⇒ Fault Hypothesis: single fault) & Fault Management vor
- Bietet folgende Services:
 - Clock Synchronization
 - Zeitliche Kapselung
 - Membership Service
 - Composability
 - Support für Mode Changes
 - Fault-Tolerance Support

TTP-Aufbau

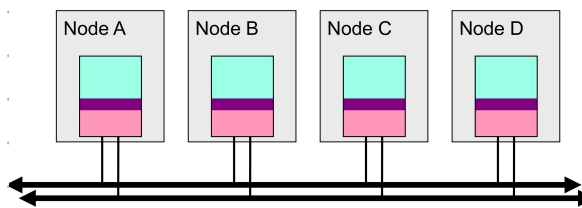
- TTP-Node: Eine electronic control unit (ECU)
- TTP-Cluster: Knoten, die über TTP-Kanal verbunden sind

o Bus-Topologie mit Broadcast-Kommunikation

- Publisher schreibt in den Bus (zB Sensorwerte)
- Subscriber liest vom Bus & reagiert (zB Aktor-Output)



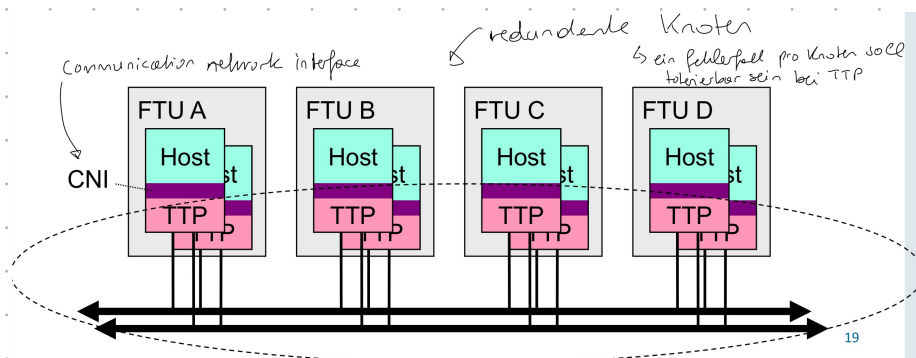
o Single-Fault Hypothesis: Kommunikationssystem muss einen Fehler tolerieren \Rightarrow Redundante Kommunikationskanäle



o Single-Fault Hypothesis: Ein fehlerhafter Knoten muss toleriert sein \Rightarrow Knoten bestehen aus 2 unabhängigen Parts:

- Host Computer: Führt Anwendungscode aus
- TTP Controller: Sorgt für TTP Service

o Fehler-tolerante Architektur: Komponenten werden dupliziert



Communication Network Interface (CNI)

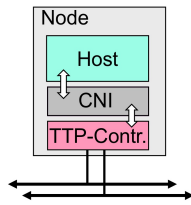
- o Ist die Datenschnittstelle zwischen Host & TTP Controller \Rightarrow Wirkt als Temporal Firewall für Kontrollsignale

Host

- writes data for sending to CNI
- writes control data to CNI
- reads received data/status info from CNI

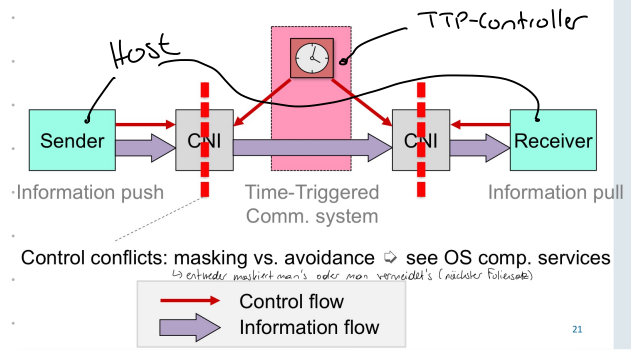
Controller

- writes received message data to CNI
- sets status
- sends messages composed from CNI data over the network



20

Handling Control Conflicts in TT Comm.

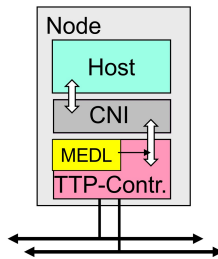


TTP-Communication

- Sendeschema (TDMA) enthält Infos darüber, wer wann was sendet ⇒ Diese Info muss nicht mitgegeben werden ⇒ Implicit Naming / Addressing

Message Schedule

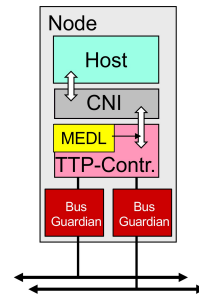
- Message Schedule ist in der Message Descriptor List (MEDL) im TTP-Controller



TTP-Fault Management

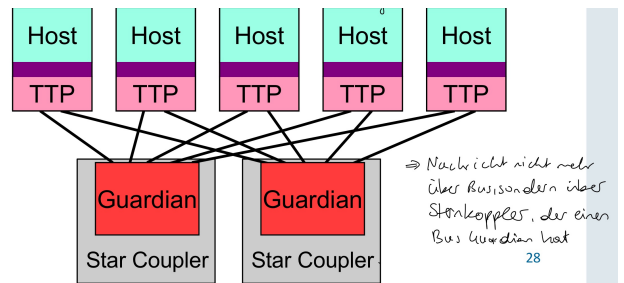
- Architektur-Ebene: Annahme, dass alle Fehler fail-silent sind
- Knoten-Ebene: Mechanismen (Fehlererkennung & -handling), die dafür sorgen sollen, dass der Knoten nach außen hin fail-silent ist
- Babbling Idiot: Fehlerhafter Controller sendet, obwohl er nicht dran ist ⇒ Verletzt Fault-Hypothese
- Bus Guardian:
 - Verhindert Babbling Idiots
 - Unabhängiger Controller für Bus-Zugriff (Gate Keeper) in jeder Node

- Weiß, wann der Node senden darf & öffnet das Gate nur dann



o Star Topology:

- Alternative zu Bus-Modell
- Guardians nur noch in Star-Coupler Switches nötig
- Star-Coupler Switches übernehmen gewisse Fehlermanagement-Funktionen: zB slightly-off-specification errors (SOS)
- SOS errors: Input, die für manche Knoten richtig, für andere falsch sind



o Continuous State Agreement:

- Jeder Knoten schickt ständig auch seinen internen State (Local Time, MEDL Position, Membership Infos)
- Protokoll funktioniert, wenn States von Sender & Empfänger übereinstimmen

TTP-Clock Synchronization

- o Erwartete Empfangszeit einer Nachricht a priori bekannt
- o Tatsächliche Empfangszeit durch Controller gemessen
- o Differenz ist Indikator für Abweichung der Clocks => Nützlich für FTA-Algorithm.
- => Es werden also nicht explizit Zeitinformationen geschickt

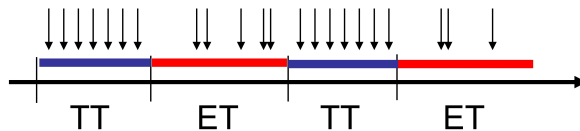
Kombination von TT- & ET-Messages

o ET-Messages **nicht** von TTP unterstützt \Rightarrow 2 Arten des Anpassens

1) Abwechselnde Zeitfenster für TT- & ET-Messages

- 2 verschiedene Kommunikationsprotokolle (TT, ET)

- **Verlust** von zeitlicher Kapselung (zu quasi beliebigen Zeitpunkten kann was kommen, worauf dynamisch reagiert werden muss)



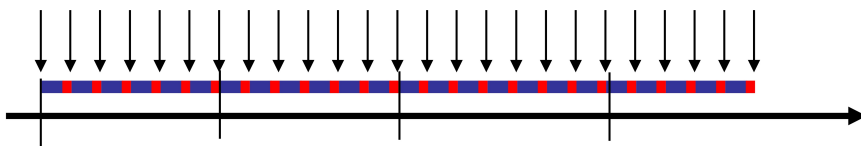
2) Layered Protocol: ET services on top of TTP

- **Nur ein** Kommunikationsprotokoll

- Möglichkeit von ET-Messages wird in TTP integriert durch **Containernachrichten**, die **Platzhalter** für ET-Messages sind

- Globale Bandbreite wird **verringert**

- **Zeitlich determiniert** \Rightarrow kein dynamisches Reagieren zu bel. Zeitpunkt!



5 RT-Component Software

Aus diesem Kapitel wird fast nichts gefragt. Hab hier nicht so viel Mühe gegeben!

Annahmen über Software

o Hard RT Software: **Closed World Assumption**

- Tasks & deren zeitliche Anforderungen sind **bekannt**

- Interaktion zwischen Tasks **bekannt** (Synchronisation usw.)

- I/O Requirements **bekannt**

\Rightarrow Runtime Guarantee kann für **worst case** gegeben werden

- Soft RT Software: Open World Assumption
 - Zur Verfügung stehendes Wissen ist nicht exakt
 - Nur QoS-Assessment vor Laufzeit möglich, während Laufzeit: Best Effort

Task Management

- Software einer Komponente besteht aus mehreren parallel laufenden Tasks
- Hard RT Software ist temporal & spatial von anderer Software isoliert
- Tasks nutzen selbe CPU, Scheduling (durch OS) nötig
- Hard RT Tasks sind cooperative, nicht competitive
- Komponente ist die unit of failure (nicht der Task)

Time Services, die ein System anbietet

- Verwaltung der Zeit: Clock Synchronization (sparse time)
- Zeitstempelvergabe, Messung von Dauern
- Message I/O
- Modellierung der Sekunde & Kalenderservice
- Event-Zeitstempel kann genutzt werden als:
 - Daten („time as data“): Zeitstempel eines Wert(-wechsels) einer RT-Entity, einfacher zu handhaben
 - Kontrolle („time as control“): System reagiert sofort auf Event, Kontrolldruck von außen

Replika-Determinismus

- Ein Set an replizierten RT-Objects ist replica determinate, wenn alle Objects dieses Sets denselben State innerhalb eines bestimmten Zeitintervalls (Real-Time) erreichen und dieselben Outputs erzeugen.
- Das Zeitintervall ist dabei bestimmt durch die Precision
- Notwendig für:
 - Implementierung konsistenter verteilter Systeme
 - Bessere Testbarkeit (Tests sind reproduzierbar genau dann wenn die Replicas deterministisch sind!)
 - Fault-Tolerance durch active replication

Error Detection Mechanismen

- Ein RT-OS muss Error Detection im Zeitbereich & im Wertebereich aufweisen
- Mechanismen:
 - Konsistenzchecks, CRC Checks
 - Überwachung von Laufzeiten von Tasks
 - Überwachung von Interrupts (MINT)
 - Double-Execution von Tasks (Zeitredundanz)
 - Watchdogs

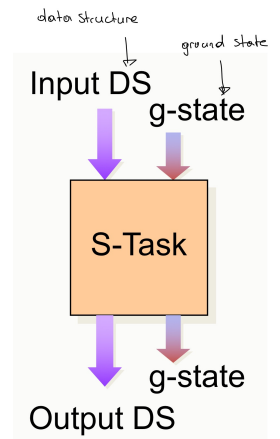
Simple Tasks

- Wird sofort von Anfang bis Ende ausgeführt, sobald's der CPU zugeteilt wurde
- Kein Blocking (keine Synchronisation/Kommunikation mit anderen Tasks)
- Fortschritt unabhängig von anderen Tasks

- Inputwerte sind schon bei Start des Tasks vorhanden (Input Data Structure)

- Outputwerte liegen vor, sobald der Task beendet wurde (Output Data Structure)

- API: Input DS, Output DS, Zustand (g-state)



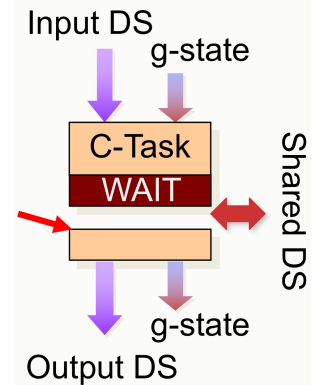
Complex Tasks

- Kann eine oder mehrere WAIT-Operationen enthalten

- Tasks hängen vielleicht voneinander ab (Synchronisation, Kommunikation)

- Fortschritt eines Tasks hängt von anderen Tasks innerhalb des Knotens oder von der Umgebung ab

- API: Input DS, Output DS, Zustand (g-state), shared DS, Dependencies



TT Task Control

- Ist eine Methode, um in strictly TT-Systems die Tasks zu schedulen (ähnlich wie bei TDMA)

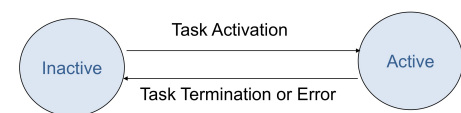
- Eine Task Descriptor List (TADL) wird vor der Laufzeit erstellt von einem statischen Scheduler (=Tabelle mit Zeitstempel & zugehöriges Event)

- Dispatcher kontrolliert die Ausführung der Tasks durch die TADL (zur Laufzeit)

⇒ Keine Dynamik zur Laufzeit

TT Task States

Non preemptive system



- Liefert zeitgesteuertes Ressourcenmanagement:

- statische CPU Allocation

- autonomes Memory-Management ⇒ wenig Aufwand für OS

- keine Queues, einfaches Buffer-Management

⇒ OS können formal analysiert werden

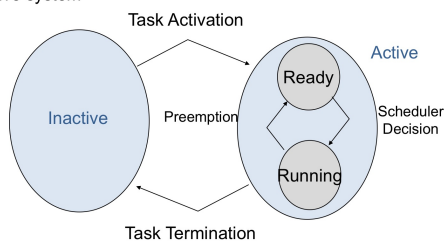
- Vorteile: Niedrige Komplexität; vorhersehbar; exakte Garantien möglich
- Nachteile: Wenig Flexibilität; Ressource-Reservation - Annahmen nur worst case
- Um dennoch flexibel zu sein, sind **Mode Changes** möglich
- ⇒ **Dynamisches** Wechseln zwischen **statischen** Modes

ET Task Control

- **Dynamischer** Scheduler entscheidet, welcher Task ausgeführt wird
- Vorteile:
 - Tatsächliche Last & Ausführungszeit wird (statt der worst case bei TT) betrachtet
 - Flexibler als TT Task Control
- Nachteile:
 - Erhöht Overhead für Scheduling
 - In der Realität ist das Scheduling NP-hard
 - Ressourcen-Vergaben sind viel komplizierter
- ET Resource Management:
 - Dynamische CPU-Allokation
 - Dynamisches Memory-Management
 - Timeout Handling von blocked States

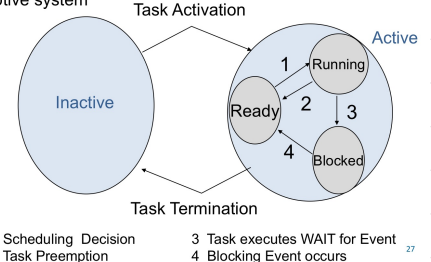
ET Task States with S-Tasks

Preemptive system



ET Task States with C-Tasks

Preemptive system



Task Interaktion

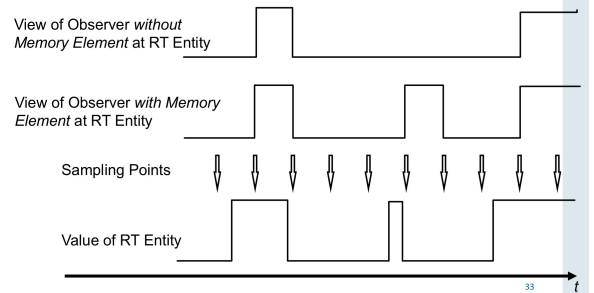
- **Reihenfolgenbestimmung**:
 - TT: Reflektiert in TADL
 - ET: WAIT

- **Datenaustausch**: - (Atomare) Nachrichten
- Shared data structure \Rightarrow Integrität wichtig
- TT: Garantiert Mutex
- ET: Semaphoren

Samplen von States / Events

- Sampling = **Periodisches** Abfragen des **Zustands** einer RT-Entity
- **States** können durch **Sampling** beobachtet werden
- Samplen von **Events** ist **nicht** möglich \Rightarrow Events müssen in **Memory Elements** zwischengespeichert werden

Sampling – Role of the Memory Element



Agreement Protokoll

- Wird benötigt bei der **Observation** einer RT-Entity
- Wenn eine RT-Entity **von 2 Nodes** observiert wird, kann Folgendes passieren:
 - Ein Event kann 2 **verschiedene Zeitstempel** bekommen
 - Beim Umwandeln analoger Werte in digitale Werte gibt es immer **Dis=kretisierungsfelder**

\Rightarrow Agreement Protokolle nötig, um eine **einheitliche** Sicht im **Zeit- und Wertebereich** zu schaffen:

- 1. Phase: Alle Partner tauschen **lokale Werte** **untereinander** aus
- 2. Phase: Jeder Partner führt **denselben** Algorithmus auf diese Daten aus (z.B. Mittelwertbildung)

\Rightarrow Agreement braucht eine **zusätzliche** Kommunikationsrunde

\Rightarrow Einfluss auf **Responsiveness**

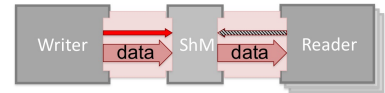
Mutex bei CNI

- Das CNI ist ein Shared Memory
- Non-Blocking Write (NBW) Protocol:

- Lesevorgänge immer konsistent
- Reader dürfen Writer nicht blockieren
- Max. Delay eines Readers hat eine obere Schranke

Mutex at CNI

- One writer, one or more reader(s) with private CPUs
- Communication via shared memory
- Intervals between writes >> duration of writes



Writer: immediate (non-blocking) write

Reader:

- async. access, mask read delay jitter → NBW variants, or
- synchronize node actions to TT network operation

Non-Blocking Write Protocol

Init:

```
CCF := 0; /* concurrency control flag */
```

Writer:

```
CCF_old := CCF;
CCF := CCF_old + 1;
write to shared struct;
CCF := CCF_old + 2;
```

Reader:

```
start: CCF_begin := CCF;
if CCF_begin mod 2 = 1
then goto start;
read from shared struct;
CCF_end := CCF;
if CCF_end ≠ CCF_begin
then goto start;
```

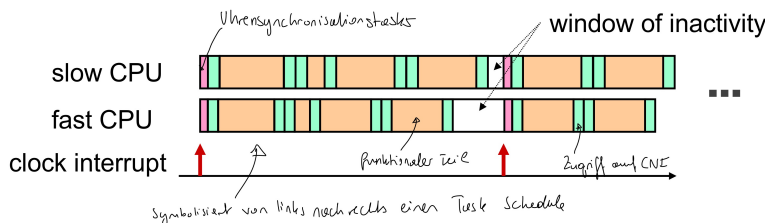
um zu erkennen, ob im Moment geschrieben wird

↳ non-block. access, wenn ein Konflikt war → wird ein Konflikt → kein Zugriff, wenn kein Konflikt → geben die beiden 2 Werteoperationen eine ganze Zeit lang

CCF arithmetics in practice: all CCF operations mod (2 * bigN)

Synchronization mit RT-Clock

- Damit Knoten mit unterschiedlich schnellen Clocks synchronisiert werden können, müssen im Task Schedule windows of inactivities vorgesehen werden (Resynchronisation durch Clock Interrupt von CNI)



6 Worst Case Execution Time Analysis

RTS-Konstruktionsablauf



Wozu WCET?

- Abarbeitungszeit von Tasks kann variieren
- Man kann nicht alle möglichen Szenarien modellieren
- ⇒ Worst Case: „Längste“ Abarbeitungszeit annehmen

RTS Timing Analysis

- Besteht aus:
 - Schedulability Objects:
 - Units of Execution (Simple Tasks) mit WCET
 - Abhängigkeiten in Reihenfolge der Abarbeitung
 - Synchronisation, Kommunikation, Mutex
 - Prioritäten
 - WCET-Analysis Objects:
 - Simple Tasks
 - Wechselwirkungen (Interferences):
 - Externe Änderungen des Task States, die die Execution Time beeinflussen (zB Cache)
 - ⇒ Erwünschenswert, dass das nicht passiert

Definition von WCET

- Die WCET einer Software ist die nötige maximale Zeit, um ein gegebenes Codestück in einem gegebenen Kontext (Inputs, State) auf einer gegebenen Maschine auszuführen
- ⇒ Worst Case Time muss ermittelt werden, nur Beobachten bei verschiedenen Fällen reicht nicht!
- Ziel: Eine obere Grenze für die Execution Time finden, die:
 - safe ist (WCET nicht unterschätzt)
 - möglich tight ist (nicht zu pessimistisch)
 - cost reasonable ist (Analyse nicht zu teuer)

Statische WCET Analyse

- Berechnet eine obere Schranke für die Execution Time eines Codestücks
- Modelliert dafür:
 - Software: Source code, Executable
 - Hardware: Prozessor (Pipeline), Memory (Caches, ...)
 - Kontext: Initial software + hardware state

WCET Determinants

- Mögliche Execution Time ist bestimmt durch:
 - Mögliche Sequenz von Aktionen eines Tasks (= Execution Path)
 - Dauer der einzelnen Aktionen

Path Timing - Simple vs. Complex Architecture

- Execution Time von Path $k = xt(p_k)$
- Simple Architecture: Konstante Dauer einzelner Aktionen a_i : $xt(p_k) = \sum_i n_{k,i} \cdot t(a_i)$
wie oft

◦ Complex Architecture: Veränderliche Dauer einzelner Aktionen a_i :

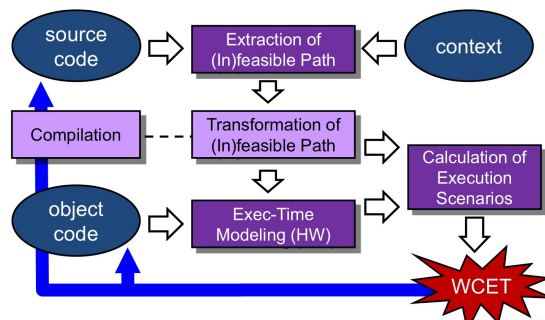
$$xt(pic) = \sum_i \sum_{j \in I_i} t(a_{i,j}(k_i))$$

⇒ Gründe: Pipelining, Caches, ...

Challenges

- Path-Analyse: (Un-)aufführbare Paths erkennen
 - Syntaktische Restriktionen (wenn A, dann B usw.)
 - Semantische Restriktionen (inhaltliche Analyse)
 - Input-Datenbereich
- Modellierung des Hardware-Timings
- WCET Berechnung
- Handling von verschiedenen Code-Repräsentationen

Generic WCET Analysis Framework



Flow Facts

- Path Information (= Flow Facts) können verwendet werden, um die Qualität der WCET-Analyse zu verbessern:
 - Loop Bounds
 - Charakteristiken & Abhängigkeiten im Code

```

for i := 1 to N do
  for j := 1 to i do
    begin
      if c1 then A.long
      else B.short
      if c2 then C.short
      else D.long
    end
  
```

← loop bound: N
 ← loop bound: N ; local: $i: 1..N$

braucht lange
 braucht kurz
 $\frac{(N+1)N}{2}$ executions

⇒ vielleicht kommt nie A.long & D.long zustande?

Automatisierung der Path-Analysis

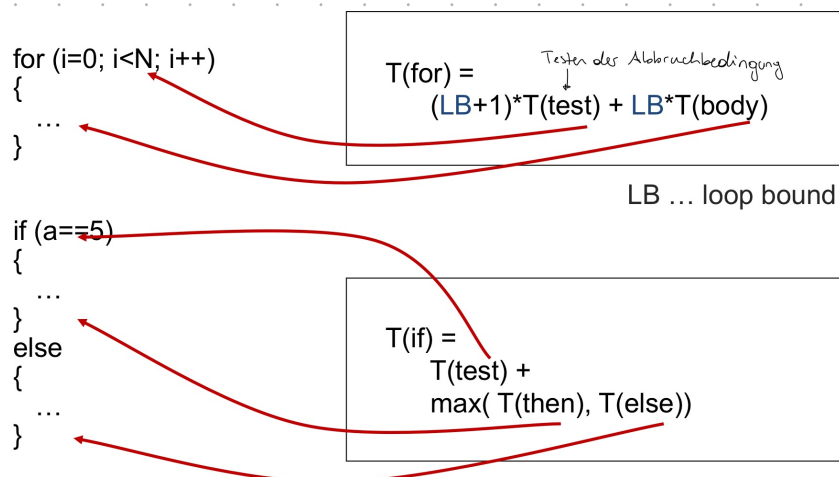
- Path-Analysis kann nicht automatisiert werden! (theoretisch äquivalent zu Halteproblem)
- Einige Infos können aber extrahiert werden durch:
 - abstract interpretation
 - symbolic modeling
 - simulation

WCET Calculation Techniques

- Tree-based WCET calculation
- Path-based WCET calculation (nicht behandelt in dieser LVA)
- WCET analysis based on implicit path enumeration (IPET)

Tree-based WCET Calculation

- Auch "timing-schema approach" genannt
- Bottom-up Traversierung vom Syntaxbaum
- Syntaxbaum repräsentiert syntaktische Konstrukte inkl. Timing-Informationen
- Timing Schema: Regel für syntactic unit um das Timing der syntactic unit aus dessen Bestandteilen zu ermitteln

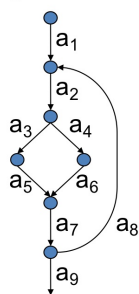


WCET Calculation using IPET

- Programm gegeben als **Control-Flow Graph (CFG)**
- Nutzt Methoden der **integer linear programming (ILP)** oder **constraint-solving** um WCET-Bound zu berechnen
- WCET Analyse als Optimierungs-/Maximierungsproblems:
 - **Goal function** beschreibt die Execution Time & soll **maximiert** werden
 - **Constraints**, die mögliche Pfade beschreiben, sind dabei **Randbedingungen**

WCET IPET: goal function (simple HW)

Program



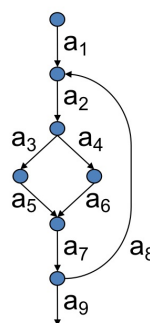
WCET: maximize $\sum x_i \cdot t_i$

- x_i ... variable: execution frequency of CFG edge a_i
- t_i ... coefficient: execution time of edge a_i

Example: $t_1: 40, t_2: 56, t_3: 82, t_4: 12, t_5: 10, t_6: 10, t_7: 32, t_8: 10, t_9: 102$
 Goal function: $40x_1 + 56x_2 + 82x_3 + 12x_4 + 10x_5 + 10x_6 + 32x_7 + 10x_8 + 102x_9$

WCET IPET: constraints (simple HW)

Program



Flow constraints:

$$\begin{aligned}
 x_1 &= 1 \\
 x_1 + x_8 &= x_2 \\
 x_2 &= x_3 + x_4 \\
 x_3 &= x_5 \\
 x_4 &= x_6 \\
 x_5 + x_6 &= x_7 \\
 x_7 &= x_8 + x_9 \\
 x_2 &\leq LB \cdot x_1
 \end{aligned}$$

in jedem Knoten kann ich gleich oft rein & wieder raus

Example: loop bound 20
 Loop constraint: $x_2 \leq 20 \cdot x_1$

◦ IPET solution = WCET bound

◦ Vorteile:

- Komplexe Flows können beschrieben werden
- Constraints sind leicht zu erstellen
- Guter Tool Support

◦ Nachteile:

- ILP ist NP-hard
- Flow Facts, die die Execution Order beschreiben, sind schwer integrierbar

Execution Time Modeling mit komplexer Hardware

- Bei komplexer Hardware (Cache, Pipelining) kommt ein zusätzlicher Schritt des Execution-Time Modeling vor der WCET-Analyse:

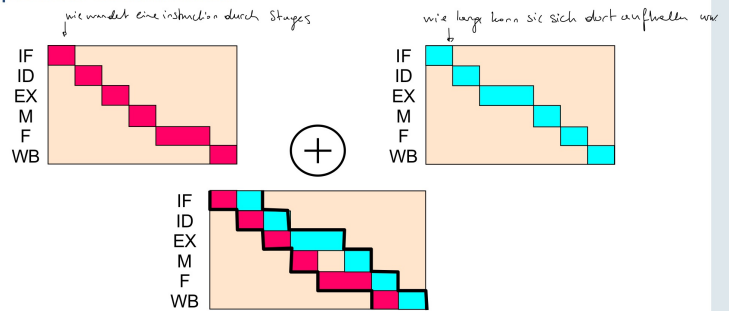
1. Cache Analysis:

- Man versucht, Cache-Verhalten zu modellieren durch Kategorisierung (always hit/miss, ...), um nicht immer einen cache miss anzunehmen

2. Pipeline Analysis:

- Basiert auf reservation tables

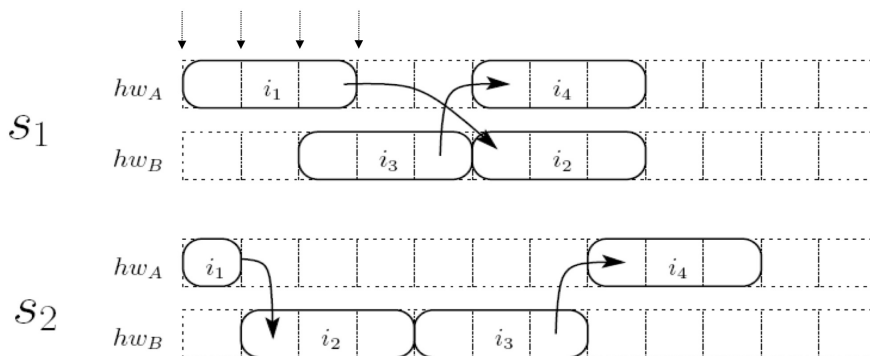
Sequential combination of two reservation tables



3. Path Analysis + WCET Calculation

Timing Anomalies

- Wenn zB eine schnellere lokale Ausführungszeit zu einer globalen längeren Gesamt-Ausführungszeit führt
- Möglichkeit für Timing Anomalies macht WCET schwierig, da alle lokalen Szenarien berücksichtigt werden müssen
- Erschweren Divide-and-Conquer Ansätze



obwohl i1 unter einem Zyklus braucht, dauert S2 länger!

7 RT Scheduling

Anforderungen / Requirements

- Abhängigkeiten in der Reihenfolge
- Mutual Exclusion
- Rate Requirements (Ausführungsrate der Tasks)
- Deadline & Response-time Requirements

Klassifizierung der Scheduling-Algorithmen

- **Guaranteed** (bei Hard RT) vs. **Best Effort** (bei Soft RT)
 - Werden Deadlines immer eingehalten oder versucht man, möglichst viele Tasks zu schaffen?
- **Statisch** vs. **Dynamisch**
 - Entscheidungen vor oder zur Laufzeit?
- **Preemptive** vs. **Non-Preemptive**
 - Können Tasks unterbrochen werden?
- **Single-Processor** vs. **Multi-Processor**
- **Central** vs. **Distributed**

Terminologie

- **Periodischer Task**:
 - **hard Deadline**
 - ausgeführt in **periodischen** Intervallen (üblich: **Periode = Deadline**)
 - Parameter: Periode T_i , Deadline D_i , WCET C_i
- **Aperiodischer Task**:
 - **soft** oder **gar keine** Deadline
 - Ziel: Quality of Service, Responsiveness optimieren

◦ Sporadischer Task:

- hard deadline
- wird sporadisch ausgeführt
- darf nicht beliebig oft bzw. knapp hintereinander ausgeführt werden
⇒ **MINI** (Minimum inter arrival time) = minimaler Abstand
- Parameter: $mint_i, D_i, C_i$

Clairvoyance

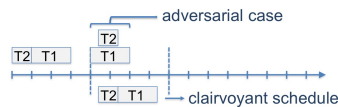
- Ein Scheduler ist **clairvoyant**, wenn er **alles** über die Zukunft weiß
 - Ist **real nicht zu erreichen**, wird aber als **Messlatte** verwendet
- Ein Scheduler ist **optimal**, wenn er einen Schedule finden kann, wann **immer** der clairvoyant Scheduler einen finden kann
- Allgemein gilt: Ein dynamischer Scheduler **kann nicht optimal** sein

⇒ Beweis:

Adversary Argument

Task set: T_1, T_2 , mutually exclusive

Task	Period / Deadl.	WCET	Type
T1	4 / 4	2	periodic
T2	4 / 1	1	sporadic



Although there is a solution, an online scheduler cannot find it.

Cyclic Executive

- Ist ein **non-preemptive statisches** Schedulingverfahren (ähnlich wie TDMA)
- **Vorlaufzeit** geplant
- (Pseudo-)Parallele Tasks werden gemeinsam zu einem Set / einer **Folge von Procedures / Procedure-Calls** zusammengefasst
- **Minor Cycle**: Minimale Interrupt-Periode, mit der **eine Folge** von Procedure-Calls aktiviert wird ⇒ Bestimmt minimale cycle time der Tasks
- **Major Cycle**: Sequenz aus **allen** minor cycles; Definiert die **Periode** des **gesamten** Systems ⇒ Bestimmt minimale cycle time

- Statische Planung, was in jedem der minor cycles passiert
- Deterministisches Verhalten & Timing

Cyclic Executive – Example

Task	Period	Exec. Time
a	10	2
b	10	3
c	20	4
d	20	2
e	40	2
f	40	1

muss in jedem 1. minor cycle geschaltet werden
 muss in jedem 2. minor cycle geschaltet werden
 muss in jedem 1. major cycle geschaltet werden

minor cycle: 10 time units
 major cycle: 40 time units

```

while (1) {
  wait_for_timer_interrupt();
  task_a(); task_b(); task_c(); task_f(); =10 time units
  wait_for_timer_interrupt();
  task_a(); task_b(); task_d(); task_e(); =10 time units
  wait_for_timer_interrupt();
  task_a(); task_b(); task_c(); =10 time units
  wait_for_timer_interrupt();
  task_a(); task_b(); task_d(); =10 time units
}
= 40 time units
10
  
```

Eigenschaften:

- Procedure-Calls statt Tasks zur Laufzeit
- Procedures teilen gemeinsamen Adressraum
- Mutex automatisch gegeben
- Alle Perioden müssen Vielfache vom minor cycle sein
- Keine aperiodischen/sporadischen Tasks möglich ⇒ Inflexibel
- Lange Tasks problemhaft

Fixed Priority Scheduling (FPS)

- Jeder Task bekommt eine statische Priority vor Laufzeit
- Priorities der ausführbaren Tasks bestimmen Reihenfolge
⇒ preemptive
- Priorities ergeben sich durch zeitliche Anforderungen
- Prominentester FPS im Echtzeitbereich: RMS

Rate-Monotonic Scheduler (RMS)

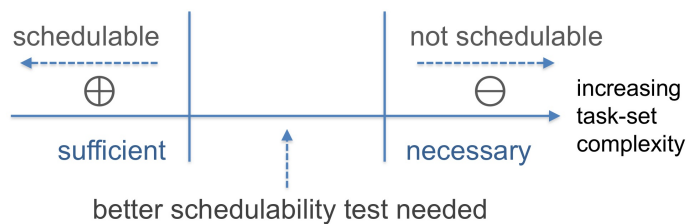
- FPS \Rightarrow also preemptive
- Rate-Monotonic \Rightarrow Kürzeste Periode hat höchste Priorität
 - \Rightarrow Task mit höchster Priority kommt dran
- Rate-Monotonic Priority Scheduling ist optimal für FPS
 - \Rightarrow Wenn ein Task Set mit FPS gescheduled wird, dann kann RMS den Set ebenfalls schedulen

Schedulability Test

- Ist eine Analyse, um für eine Scheduling-Strategie zu ermitteln, ob ein Task Set damit überhaupt schedulebar ist (= Werden alle Deadlines eingehalten?)
- Es gibt notwendige & hinreichende Schedulability Tests:

If a sufficient schedulability test is positive, the tasks are definitely schedulable

If a necessary schedulability test is negative, the tasks are definitely not schedulable



- \Rightarrow Mit dem notwendigen Test wird die notwendige Bedingung überprüft, ob die Möglichkeit besteht, dass das Task Set schedulebar ist.
- Mit dem hinreichenden Test wird die hinreichende Bedingung überprüft, ob der Task Set sicher schedulebar ist.

Utilization-Based Schedulability Test

◦ Utilization $U = \sum \frac{C_i}{T_i}$

⇒ Ausführungszeit durch die Periode = CPU-Auslastung für einen Task

⇒ Summe davon = **Wieviel Prozent** der Zeit wird ein beliebiger Task ausgeführt

◦ **Notwendiger** Schedulability Test für RMS:

$U \leq 1 \Rightarrow$ Schedulability **nicht** ausgeschlossen!

◦ **Hinreichender** Schedulability Test für RMS:

$U \leq n \cdot (2^{\frac{1}{n}} - 1) \Rightarrow$ Task Set **schedulebar!**

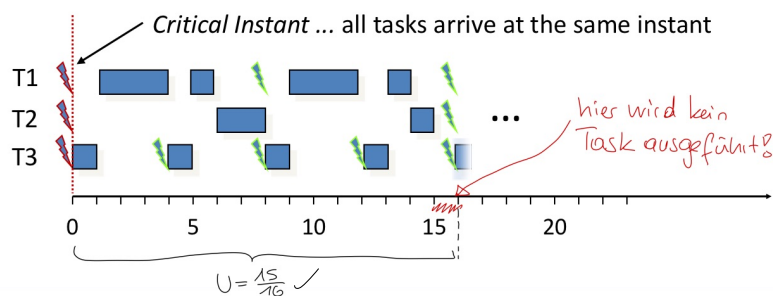
↳ für große n gilt: $n \cdot (2^{\frac{1}{n}} - 1) \approx \ln(2) = 0,69$

RMS Scheduling – Example

Task	C_i	T_i	Priority
T1	4	8	Low $\frac{1}{2}$
T2	3	16	Medium $\frac{3}{16}$
T3	1	4	High $\frac{1}{4}$

$U = \sum C_i \cdot T_i = \frac{15}{16} \leq 1$

↳ bei 16 Zeiteinheiten wird in 15 davon ein Task ausgeführt



Earliest Deadline First (EDF)

◦ **dynamisch, preemptive**

◦ Der Task mit der **frühesten** Deadline wird ausgeführt

◦ **Utilization-Based Schedulability Test für EDF:**

Notwendige & hinreichende Bedingung: $\sum \frac{C_i}{T_i} \leq 1$

◦ Allgemein erreicht man mit EDF **höhere** Utilization als mit RMS

FPS vs. EDF

- Implementierung von statischen Priorities bei FPS einfacher
- EDF: Tasks, die ready sind, müssen an die richtige Stelle der Ready Queue einsortiert werden (nach Deadline)
- FPS: Tasks ohne Deadlines einfach einzubinden (zB. niedrige Priorität vergeben)
⇒ Bei EDF ist das kritischer / komplexer
- Bei Überlast können bei FPS Low-Priority Tasks ihre Deadlines verpassen
- Überlast bei EDF ist unvorhersehbar (Domino-Effekt möglich)

Response-Time Analysis

- Ist ein besserer Schedulability Test für FPS
- Utilization-based Tests sind:
 - simpel
 - nicht genau
 - nicht gut anwendbar auf generalisierte Tasks

⇒ Response-Time Analysis:

- Berechnet Worst Case Response Time R_i für jeden Task und berücksichtigt dabei die Interference I_i von Tasks mit höherer Priority

⇒ $R_i = C_i + I_i$, I_i ... die Zeit, die der Task durch andere mit höherer Priority verzögert wird

- Ziel: Checkt, ob Deadline eingehalten wird, also: $R_i \leq D_i$
- Um I_i abzuschätzen, muss bekannt sein, wie oft ein Task i durch einen anderen Task j mit höherer Priority unterbrochen wird
- Angenommen alle Tasks werden bei gleicher Zeit aktiviert, dann beträgt die max. Anzahl an Task Preemptions von Task i durch j :

$$\left\lceil \frac{R_i}{T_j} \right\rceil$$

⇒ wie oft geht die Periode von j in meine response time rein ⇒ so oft stört er mich ⇒ die Zeitkosten jeder Störung sind C_j (=Ausführungszeit j)

$$\Rightarrow I_j = \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j$$

- Summiert man über alle höherpriorisierten Tasks (=hp_i), hat man die worst case response-time von Task i:

$$R_i = C_i + \sum_{j \in hp_i} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j$$

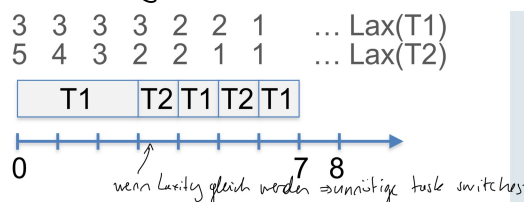
⇒ Rekursiv ⇒ Alternativ: $w_i^{n+1} = C_i + \sum_{j \in hp_i} \left\lceil \frac{w_i}{T_j} \right\rceil \cdot C_j$, $w_i^0 = C_i$

- Response-Time Analysis ist notwendig & hinreichend für FPS!

Least-Laxity First Scheduling (LLF)

- Dynamisch, preemptive
- Laxity: Differenz zwischen verbleibender Zeit & Deadline
- Task mit geringster Laxity hat höchste Priority
- Optimal für Uniprocessor Systeme

Task	Deadline	WCET
T1	8	5
T2	7	2

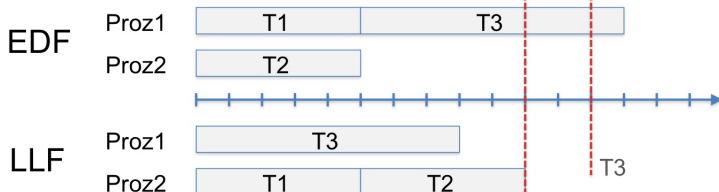


◦ Man kann bei einigen gleichen Laxity-Werten viele Task Switches haben ⇒ Modified LLF (MLLF) löst dieses Problem

Multiprocessor Scheduling

Task set

Task	Deadline	WCET
T1	10	5
T2	10	5
T3	12	8



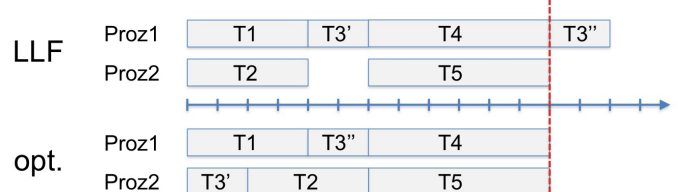
⇒ LLF besser als EDF bei Multiprocessor

T1, T2 Deadlines

Non-Optimality of LLF in Multiprocessor Sys.

Task set

Task	Arrival	Deadline	WCET
T1	0	4	4
T2	0	8	4
T3	0	12	4
T4	6	12	6
T5	6	12	6

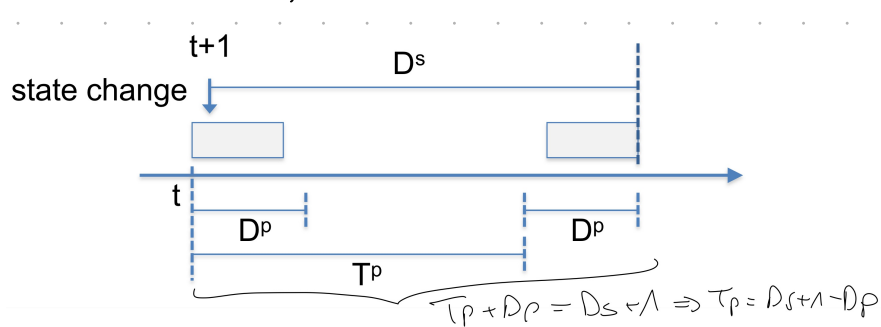


⇒ LLF ist auch nicht optimal

Deadline T3

Sporadic Task Scheduling

- Sporadischer Task wird quasi-periodisch gemacht
- Sporadic Task Parameter: $\min t^s, D^s, C^s$
- Quasi-periodic Task Parameter: $C^p = C^s$



$$D^p \leq D^s, \text{ z.B. } D^p = C^p = C^s$$

$$T^p = \min(\min t^s, D^s - D^p + 1)$$

Sporadic Server Task

- Sporadic-task transformation may yield poor processor utilization, especially if D^s is small compared to $\min t^s$.
- We can define a server task for the sporadic request that has a short latency
- The server is scheduled in every period, but is only executed if the sporadic request actually appears. Otherwise the other tasks are scheduled
- This will require a task set in which all the other tasks have a laxity of at least the execution time of the server task.

\Rightarrow ich reserviere zwar Zeit, aber wenn der sporadic task nicht kommt, fahre ich andere Tasks aus

Priority Inversion

- Problematik bei Tasks mit Mutual Exclusion constraints
- Priority Inversion tritt auf, wenn ein höherpriorisierter Task auf einen niedrigerpriorisierten Task warten muss (hp-Task ist geblockt)
 - direct blocking: niederprioriter Task, der in kritischem Abschnitt ist, blockiert höherprioriten Task
 - \Rightarrow nicht verhinderbar, weil der höherprioritäre Task die Ressource nicht preemptive bekommen darf
 - indirect blocking: mittelprioriter Task kommt dazu & unterbricht den niederprioriten (kann er, weil er keine Ressource braucht + höhere Priorität hat) \Rightarrow das blockiert den höherprioriten zusätzlich

⇒ **verhinderbar** mit Priority Inheritance

◦ Bsp indirect blocking:

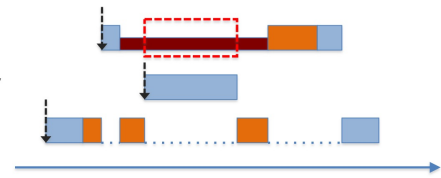
In the shown example the high-priority task is indirectly blocked by the medium-priority task (dashed box).

Task 1 and Task 3 use the same resource

Task 3, highest priority

Task 2, medium priority

Task 1, lowest priority



- ... task executes
- ... mutex resource use
- ... task is blocked, priority inversion
- task is preempted
- ... task indirectly blocked

30

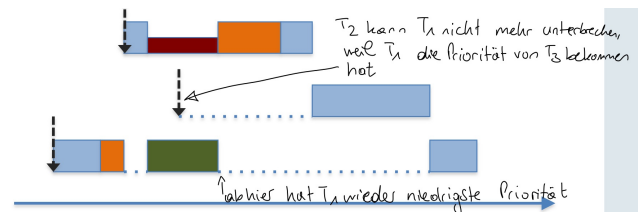
Priority Inheritance

◦ Wenn ein Low-Priority Task einen oder mehrere Tasks mit Higher-Priority blockiert, dann bekommt er **temporär** die höchste Priorität der von ihm blockierten Tasks

Task 3, highest priority

Task 2, medium priority

Task 1, lowest priority



- ... task executes
- ... mutex resource use
- ... task is blocked
- task is preempted
- ... task using mutex resource runs at inherited priority

31

◦ Priority Inheritance **verhindert nicht** Deadlocks!

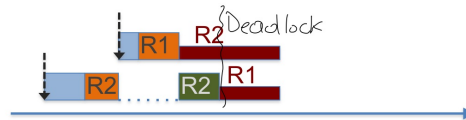
The priority-inheritance protocol does not prevent deadlocks

Example

1. Task 1 locks R2
2. Task 2 preempts Task 1 and locks R1
3. Task 2 tries to lock R2 but fails
4. Task 1 inherits priority from Task 2 but blocks when trying to lock R1

Task 2, high priority

Task 1, low priority



- ... task executes
- ... mutex resource use
- ... task is blocked
- task is preempted
- ... task using mutex resource runs at inherited priority

32

⇒ Deadlocks sind **verhinderbar** durch **Priority Ceiling Protocol**

Priority Ceiling Protocol

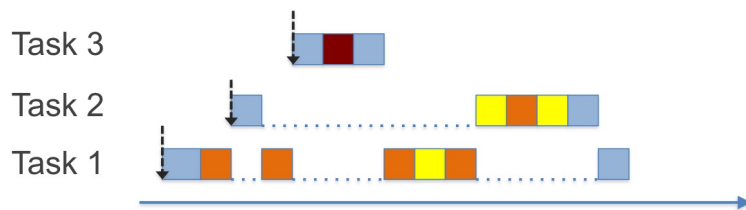
- Jeder Task hat eine **Default Priority**
- Jede Ressource (für Mutex) bekommt ein **Priority Ceiling** zugeordnet: Priority Ceiling = **Höchste** Priorität, die unter den Tasks auftritt, die diese Ressource **verwenden**
- Zusätzlich gibt es die **dynamische** Priorität für jeden Task: Dynamische Priorität = $\max(\text{eigene default Priority, Ceiling Value aller Ressourcen, die der Task gerade verwendet})$
- Ein Task bekommt eine Ressource



Die Priorität des Tasks ist **größer** als die Ceiling Values **aller** Ressourcen, die in dem Moment von anderen Tasks **geblockt** sind

Priority Ceiling Protocol – Example

Task 3: ... P(S1) ... V(S1) ... highest priority
 Task 2: ... P(S2) ... P(S3) ... V(S3) ... V(S2) ... medium priority
 Task 1: ... P(S3) ... P(S2) ... V(S2) ... V(S3) ... lowest priority



Critical section guarded by Sx (priority ceiling):

■ ... S1 (high) ■ ... S2 (medium) ■ ... S3 (medium)

↑ von T₃ verw. ↑ von T₂ verw. ↑ von T₁ verw.

Response-Time mit Blocking

- Berechneter worst case blocking time B_i

$$\Rightarrow R_i = C_i + B_i + \sum_{j \in hp_i} \left\lceil \frac{R_j}{T_j} \right\rceil \cdot C_j$$

Statisches vs Dynamisches Scheduling

	Statisch	Dynamisch
Vorteil	<ul style="list-style-type: none">◦ Berechnung vor Laufzeit◦ Wenig Komplexität zur Laufzeit◦ Formale Analyse implizit dabei◦ Keine Mutex & Synchr. nötig, da Schedule vorher erstellt wurde◦ Verwendet Precedence Graph	<ul style="list-style-type: none">◦ Performance kann mit Infos aus der Laufzeit verbessert werden◦ Aufteilung der Tasks fairer, da nicht immer Worst-Case-Annahme◦ Flexibler als statisches Scheduling
Nachteil	<ul style="list-style-type: none">◦ Kann schlechtere Average Time haben◦ Meist worst-case Annahme◦ Ressourcennutzung suboptimal	<ul style="list-style-type: none">◦ Formal quasi nicht analysierbar◦ Worst-case Zeit nicht berechenbar

Precedence Graph: Teilt Tasks in Mengen, die voneinander abhängen
⇒ Sucht im Suchraum durch Baum ⇒ wenn gefunden, ist Scheduling Test erfüllt

Kategorisierung der Schedulingverfahren

- Statisch: Cyclic Executive, FPS, RMS
- Dynamisch: EDF, LLF