Hauptseminar im SS 2006

Ausgewählte Kapitel eingebetteter Systeme (AKES)

Fehlertoleranz in eingebetteten Systemen

Vortragender:
Thomas Klöber

Betreuer:
Olaf Spinczyk

Inhalt

1. Einführung

Eingebettete Systeme finden in den verschiedensten Gebieten des täglichen Lebens Anwendung. Während es bei Dingen wie Handys, Fernbedienungen oder MP3-Playern zwar unerfreulich aber akzeptabel ist, dass diese irgendwann ausfallen, darf dies im Finanzwesen wie z.B. in Geldautomaten sowie bei sicherheitskritischen Anwendungen wie ABS, ESP oder Flugzeugsteuerungen nicht passieren. Während sich die Folgen im ersten Fall noch auf monetäre Verluste beschränken, drohen in den anderen Fällen Schäden für Leib und Leben. Die Systeme dürfen weder durch interne Fehler, noch durch Einwirkung von außen zu einem undefinierten Verhalten gebracht werden.

Die zu tolerierenden Fehler reichen von Ausfall einzelner Komponenten / Teilsysteme (Hardware-Fehler) über fehlerhafte Übertragung von Daten (Kommunikationsfehler) bis hin zu fehlerhaft programmierten Anwendungen (Software-Fehler).

Ob und in welchem Maß jede einzelne Fehlerklasse toleriert wird, muss für jedes System einzeln entschieden werden. Es ist nicht zweckmäßig, in jedem System möglichst viele Fehler zu tolerieren, da jeder zu tolerierende Fehler das System komplexer macht.

Damit eventuell auftretende Fehler toleriert werden können, müssen sie zunächst als solche erkannt werden. Dazu und zur späteren Korrektur der Ergebnisse wird Redundanz / Replikation eingesetzt:
- redundante Kodierung / Speicherung der Daten
- redundante Berechnung der Ergebnisse
- redundante Verwendung von Bauteilen

Im Folgenden wird nur auf Hardware-Fehler sowie Software-Fehler eingegangen, Kommunikationsfehler werden nur am Rande behandelt, auch diese erschöpfend zu behandeln würde zu weit führen.

2. Tolerieren von Hardware-Fehlern

Gegen den Ausfall eines Stücks Hardware besteht die Absicherung prinzipiell immer in der Replikation der darauf laufenden Prozesse (auf zusätzliche Hardware). Ein Satz Hardware, der benötigt wird, um einen replizierten Prozess laufen zu lassen, bildet aus Sicht der Fehlertoleranz eine Einheit und wird im Folgenden als Knoten bezeichnet. Es existieren verschiedene Strategien, wie Redundanz genutzt wird, um Fehler zu tolerieren, die sich im Kommunikationsaufwand, dem Zeitverlust durch einen Fehler und den Anforderungen an die einzelnen Prozesse bzw. die verwendete Hardware unterscheiden.

2.1. Aktive Replikation

Bei aktiver Replikation sind alle replizierten Prozesse gleichzeitig aktiv und erzeugen ihre Ergebnisse simultan. Diese werden von einem Voter verglichen und das mehrheitlich erzielte Ergebnis wird weitergeleitet. Somit ist ein Knoten fehlerhaft, wenn das erzeugte Ergebnis nicht mit

der Mehrheit übereinstimmt oder zu spät bzw. gar nicht erzeugt wird. Die beiden letzteren Fälle werden über Time-Outs erkannt.

Um dieses Verfahren verwenden zu können, müssen folgende Punkte erfüllt sein:
- Eingabe-Konsistenz: Jedes Replikat eines Prozesses muss die gleichen Nachrichten in der selben Reihenfolge erhalten.
- Replikations-Determinismus: Bei gleichem Startzustand müssen alle Replikate eines Prozesses, bei Verarbeitung eines konsistenten Satzes von Eingaben, einen identischen und identisch geordneten Satz von Ausgaben erzeugen.

Generell werden hier 2f+1 Replikate gebraucht, um f fehlerhafte Knoten tolerieren zu können. Der klassische Fall ist, dass ein Fehler erlaubt ist und somit 3 Replikate benötigt werden (triple modular redundancy, TMR).

Einen Sonderfall stellt der Betrieb von „fail-silent“-Knoten in aktiver Replikation dar. Ein Knoten ist „fail-silent“, wenn sichergestellt ist, dass nur korrekte Ergebnisse gesendet werden, eine Übertragung von fehlerhaften Ergebnissen wird verhindert.
Ist dies der Fall, so kann auf eine Auswertung der einzelnen Ergebnisse durch den Voter verzichtet werden, da jede gesendete Nachricht als korrekt angenommen werden kann.
Daher ist der minimale Replikationsgrad in diesem Fall 2, die Kommunikation ist vereinfacht und die Performance verbessert, da die Ergebnisse direkt nach der Erzeugung verteilt werden können, statt auf das Ende des Voting zu warten.

2.2. Passive Replikation

Im Gegensatz zur aktiven Replikation ist hier immer nur ein Knoten aktiv. Der aktive Knoten kopiert seinen Zustand zu bestimmten Zeitpunkten (Checkpoints) auf die anderen Knoten (Backups). Falls der aktive Knoten ausfällt, wird eines der Backups aktiviert und beginnt am aktuellsten Checkpoint zu arbeiten.
Um ein Ausfallen des aktiven Knoten zu erkennen, müssen alle Knoten „fail-silent“ sein.

Ein Problem bei dieser Art von Replikation kann das doppelte Versenden von Nachrichten aufgrund eines Fehlers sein.

Beispiel:
Der aktive Knoten hat seit Erstellung des aktuellsten Checkpoints 2 Nachrichten gesendet. Der Knoten fällt aus und es wird ein Backup aktiviert, das auf dem Stand des Checkpoints das Arbeiten beginnt. Dieses erzeugt nun die bereits gesendeten Nachrichten erneut.

Das Versenden von doppelten Nachrichten kann auf zweierlei Art verhindert werden, dem systematischen oder periodischen Erstellen der Checkpoints.

Systematische Checkpoints werden nach jedem Versenden von Information erstellt. Daher erfordert ein Rollback auf den aktuellsten Checkpoint niemals das erneute Senden einer Nachricht.

Die periodische Strategie reduziert die Anzahl der Checkpoints, indem sie z.B. nur alle n

Nachrichten erstellt werden. Im Falle einer Wiederherstellung werden alle von dem neu aktivierten Replikat erstellten Nachrichten mit einem Log der bereits gesendeten Nachrichten verglichen und nur versendet, falls keine gleichwertige Nachricht gefunden wird.
Es muss also Replikations-Determinismus herrschen und alle replizierten Knoten müssen die Nachrichten in der selben Reihenfolge erhalten, so dass die Ersatz-Replik die selben Nachrichten produziert, wie das Original vor dem Ausfall und erkennen kann, dass diese bereits gesendet wurden.

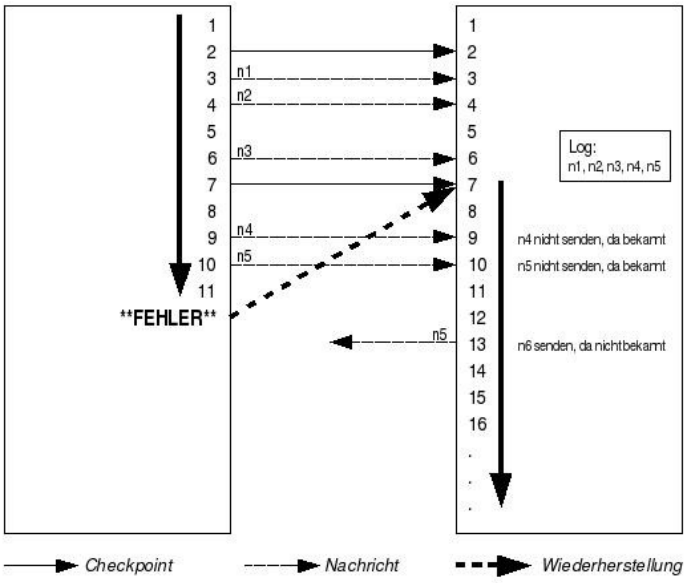

Illustration 1: periodische Checkpoints

Obwohl systematische Checkpoints mehr Overhead produzieren, als periodische, ist die Fähigkeit, nicht-deterministische Prozesse verwenden zu können, ein wichtiger Vorteil.

2.3. Leader/Follower Replikation

Die Leader/Follower Replikation stellt eine Mischform der beiden oben erläuterten Strategien dar.
Von den vorhandenen Knoten wird einer zum Leader bestimmt, die anderen sind Follower. Wie bei der aktiven Replikation sind immer alle Replikate eines Prozesses aktiv. Das Ergebnis wird jedoch nicht durch Voting ermittelt, sondern der Leader übermittelt sein Ergebnis, sobald es bereitsteht.

Er trifft alle Entscheidungen, die den Replikations-Determinismus betreffen und schickt sie mit Synchronisations-Nachrichten an seine Follower. Diese führen die selben Berechnungen durch, wobei sie die Entscheidungen des Leaders berücksichtigen und sind somit immer auf dem gleichen Stand wie der Leader. Fällt der Leader aus, wird einer der Follower zum neuen Leader bestimmt.

Somit lässt sich das Verhalten auch mit passiver Replikation vergleichen, nur dass die Verteilung des Zustandes an die anderen Knoten nicht durch Checkpoints sondern durch Berechnung erreicht wird.

Es werden zwei Formen von Synchronisations-Nachrichten verwendet, Input-Synchronisations-Nachrichten und Preemption-Synchronisations-Nachrichten.

In Echtzeit-Anwendungen ist es notwendig, den Server über den Vorrang einer Berechnung vor einer anderen benachrichtigen zu können. Daher müssen die Knoten fähig sein, ankommende Nachrichten in einer Reihenfolge, die den Vorrang einbezieht, zu konsumieren. Trotz alledem müssen alle Kopien eines Prozesses die gleichen Nachrichten in der gleichen Reihenfolge konsumieren. Deshalb erzeugt der Leader, sobald er (aus seinem lokalen Nachrichten-Pool) eine Nachricht ausgewählt hat, eine Synchronisations-Nachricht, die die Identität der zu konsumierenden Nachricht enthält. Die Follower wissen somit, welche Nachricht sie zu konsumieren haben.

Ein weiteres Problem liegt in der Tatsache, dass Prozesse in Echtzeitsystemen oftmals sehr schnell unterbrochen werden müssen, sobald ein bestimmtes Ereignis auftritt. Dies kann zu nicht-Replikations-deterministischem Verhalten führen, falls nicht jede Kopie des Prozesses an exakt der gleichen Stelle unterbrochen wird.

Im Leader/Follower-Modell sind deshalb in den Prozessen bestimmte „preemption points“ definiert, also Punkte, an denen der Prozess unterbrochen werden darf. Jedes mal, wenn der Leader einen „preemption point“ erreicht, erhöht er einen Zähler (pro Replikat). Wenn eine Nachricht beim Leader eintrifft, wird überprüft, ob diese Nachricht den Leader zu einer Unterbrechung veranlasst. Falls dies zutrifft, wird der „preemption point“, an dem das geschieht, ausgewählt (der aktuelle Zählerstand + 1) und eine Synchronisations-Nachricht erzeugt, die den Zählerstand sowie die Identifikation der Nachricht, die die Unterbrechung verursacht, enthält.

Damit dieser Mechanismus funktioniert, müssen sich die Follower in der Ausführung immer mindestens einen Schritt hinter dem Leader befinden.

2.4. Vergleich der Techniken

Aktive Replikation bietet den Vorteil, dass der angebotene Dienst im Fehlerfall nicht unterbrochen wird. Werden „fail-silent“ Knoten verwendet, entfällt auch der Voting-Prozess, und somit entsteht keine Verzögerung. Allerdings ist es unumgänglich, atomares Multicasting zu unterstützen, um die Eingabe-Konsistenz zu garantieren, was komplexe Kommunikaktionsprotokolle erfordert. Außerdem müssen sich die Prozesse replikations-konsistent verhalten. Falls die Replikate schnell auf äußere Ereignisse reagieren müssen, also mit Unterbrechungen umgehen, wird aktive Replikation sehr schwierig zu handhaben, Unterbrechungen sind schwer zu synchronisieren, da jedes Replikat an exakt der selben Stelle unterbrochen werden muss.

Bei passiver Replikation sind Unterbrechungen kein Problem, da jeweils nur ein Replikat aktiv ist. Zudem kommt passive Replikation mit einer relativ einfachen Kommunikation zurecht und die Prozesse müssen nicht deterministisch sein. Da jeweils nur eine Kopie aktiv ist, ist auch der Rechenaufwand minimal. Allerdings werden für passive Replikation „fail-silent“ Knoten benötigt und es entsteht eine Verzögerung beim Ausfall des primären Replikats, was eventuell mit den Echtzeitanforderungen an die Anwendung kollidieren könnte. Zudem entsteht im fehlerfreien Betrieb ein Overhead durch die Kommunikation zur Checkpoint-Erstellung.

Das Leader/Follower-Modell reduziert den Kommunikations-Overhead, da nur die Synchronisations-Nachrichten entstehen. Auch hier wird kein Voter benötigt. Es können nicht-deterministische Prozesse verwendet werden und Unterbrechungen stellen kein Problem dar.

Allerdings entsteht ebenfalls eine Verzögerung beim Ausfall des Leaders und es werden „fail-silent“ Knoten benötigt. Der Rechenaufwand gleicht dem bei aktiver Replikation, es wird jedoch trotz gleichzeitiger Aktivität aller Knoten Kommunikation zur Synchronisation benötigt.

3. Tolerieren von Software-Fehlern

Software-Fehler sollten sich durch Planung und Verifikation der Software zwar vermeiden lassen, jedoch sind sie nur in den seltensten Fällen wirklich auszuschließen. Was Software-Fehler jedoch anrichten können, wurde mittlerweile bereits bei einigen spektakulären Unfällen deutlich.

Damit eventuell vorhandene Software-Fehler nicht die Funktionalität des Systems beeinflussen, existieren, wie bei Hardware-Fehlern auch, verschiedene Techniken, die alle auf Redundanz beruhen.

3.1. Recovery-Blocks

Ein Recovery-Block ist ein Modul, in dem eine Anzahl von Alternativen, die unabhängig nach der selben Spezifikation entwickelt wurden, zusammen mit einem Akzeptanztest zu Fehlererkennung kombiniert sind. Wenn ein Recovery-Block ausgeführt wird, wird zunächst die erste Alternative ausgeführt, danach der Akzeptanztest. Falls der Akzeptanztest feststellt, dass die Alternative erfolgreich war, wird der Block verlassen.
Wird jedoch vom Akzeptanztest ein Fehler in der Ausführung festgestellt, wird der Zustand wiederhergestellt, der existierte, als der Block betreten wurde und eine andere Alternative wird ausgeführt. Dieses Prinzip wird fortgeführt, bis entweder der Akzeptanz-Test erfolgreich durchlaufen wird, oder keine weiteren Alternativen mehr übrig sind und somit der Recovery-Block als Ganzes fehlgeschlagen ist.

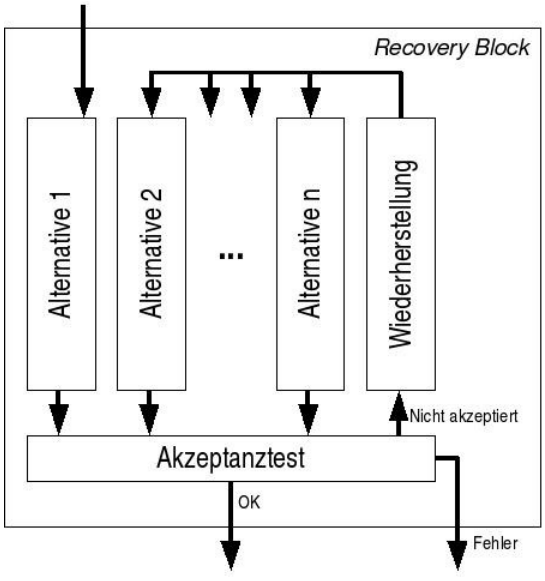

Illustration 2: Recovery Blocks

Recovery-Blocks lassen sich auch ineinander verschachteln, so dass das Versagen eines Recovery-Blocks die Wiederherstellung eines umschließenden Blocks auslösen kann.

Probleme können entstehen, wenn während der Ausführung eines Recovery-Blocks Nachrichten an andere Prozesse verschickt werden, jedoch der abschließende Akzeptanztest fehlschlägt. Somit wären alle verschickten Informationen ungültig und die beteiligten Prozesse, die diese Nachrichten konsumiert haben müssten als fehlgeschlagen angesehen werden.
Eine Möglichkeit, dieses Problem zu umgehen, wäre während der Ausführung eines Recovery-Blocks jegliche Inter-Prozess-Kommunikation zu verbieten. Das wäre jedoch für viele Anwendungen zu restriktiv, also wird eine alternative Methode benötigt.

Eine solche Alternative sind Dialoge. Dialoge sind Strukturen, die eine Anzahl von Prozessen und Datenstrukturen enthalten können und diesen Prozessen die Kommunikation untereinander ermöglichen, ohne die Möglichkeit, sich nach Fehlern wiederherzustellen, zu beeinträchtigen. Dialoge sorgen dafür, dass eine Wiederherstellung eines fehlgeschlagenen Prozesses an alle Prozesse, mit denen dieser kommuniziert hat, seit er sich im Dialog befindet, weitergeleitet wird. Recovery-Points sind demnach hier statt mit Recovery-Blocks mit Dialogen verbunden.
Der Dialog terminiert, wenn alle enthaltenen Prozesse die erfolgreiche Abarbeitung des im Dialog auszuführenden Codes bestätigen. Sollte einer der Prozesse wegen eines Fehlers nicht erfolgreich terminieren, müssen sich alle Prozesse des Dialogs wiederherstellen.

3.2. N-Version-Programming

Ebenso wie bei Recovery-Blocks werden bei N-Version-Programming mehrere Alternativen (Versionen) unabhängig nach der selben Spezifikation entwickelt. Beim Eintritt in das N-Version Modul werden alle Versionen nebenläufig ausgeführt und die Ergebnisse werden an einen Schiedsrichter weitergeleitet, der dann das endgültige Ergebnis konstruiert.

Ein Problem bei dieser Art der Software-Fehlertoleranz besteht im Umgang mit dem Fehlschlagen einer einzelnen Version. Falls das Modul nicht „zustandslos“ ist, befindet es sich nach einem solchen Fehlschlagen in einem inkonsistenten Zustand, also muss das ganze Modul als fehlgeschlagen angesehen werden.
Eine Möglichkeit, mit dem Versagen einer Version umzugehen, ist, eine neue Replik der fehlgeschlagenen Version zu klonen. Diesem Klon muss allerdings noch der Zustand mitgeteilt werden, in dem sich die ursprüngliche Version vor dem Versagen befunden hat. Teilweise kann dieser durch Untersuchung anderer Komponenten des Systems bestimmt werden, in den meisten Fällen jedoch wird man sich die benötigten Informationen von anderen, korrekten Versionen des Moduls holen müssen. Dazu muss eine Standard-Repräsentation des Zustands definiert werden, so dass eine oder mehrere Versionen ihren eigenen Zustand ausgeben können. Der neue Klon kann dann seinen Zustand daraus erzeugen.

4. Fehlertoleranz am Beispiel von Delta-4

4.1. Überblick über Delta-4

Delta-4 ist ein Gemeinschaftsprojekt, das im Rahmen des „European Strategic Programme for Research in Information Technology“ (ESPRIT) durchgeführt wurde. Sein Ziel war Definition und Design einer offenen, verlässlichen und verteilten Computersystem-Architektur.

Ein Delta-4-System besteht aus einer Anzahl (möglicherweise heterogener) Computer, die durch ein verlässliches Kommunikationssystem verbunden sind. Anwendungen sind als Softwarekomponenten, die über die Knoten des Systems verteilt sind, strukturiert. Jede Softwarekomponente darf repliziert werden, jedoch müssen die Kopien auf homogenen Knoten laufen. Jeder Knoten besteht aus einem Host-Computer und einem Network-Attachment-Controller (NAC).

Die NACs sind spezialisierte Kommunikationsprozessoren, die zusammen ein verlässliches Kommunikationssystem implementieren, das multi-point-Kommunikation erlaubt.

4.2. Hardware-Fehlertoleranz

Delta-4 unterstützt sowohl aktive als auch passive Replikation, sowie auch den Leader/Follower-Mechanismus.
Als Besonderheit ist anzumerken, dass Delta-4 bei aktiver Replikation nicht die Ergebnisse miteinander vergleicht, sondern sich auf Signaturen (eine Art Checksumme) verlässt und dass das Voting vom Kommunikationssystem durchgeführt wird, so dass kein seperater Voter nötig ist.

4.3. Software-Fehlertoleranz

Delta-4 unterstützt laut Spezifikation keine Technik zur Toleranz von Software-Fehlern, lässt sich jedoch leicht dahingehend erweitern, dass die beiden vorgestellten Techniken zum Einsatz kommen können.

Recovery-Blocks lassen sich problemlos implementieren, je nach verwendeter Hardware-Fehlertoleranz sind jedoch verschiedene Überlegungen notwendig:

aktive Replikation:

> Jedes Replikat führt die Recovery-Blocks parallel aus. Durch die Eigenschaft des Replikations-Determinismus ist sichergestellt, dass alle Replikate die selbe Alternative erfolgreich ausführen.

passive Replikation:

> Die Replikate führen die Recovery-Blocks nicht selbst aus, brauchen jedoch die Recovery-Points, für den Fall, dass ein Hardware-Fehler auftritt, während ein Recovery-Block ausgeführt wird und ein anschließender Software-Fehler eine Wiederherstellung nötig macht. Daher muss beim Eintritt in einen Recovery-Block eine Kopie der Recovery-Informationen an die Replikate gesendet werden, die nach Verlassen des Blocks wieder verworfen werden kann.
>
> Zu Beachten ist, dass Recovery-Points und Checkpoints unabhängig voneinander sind.

Während ein neuer Checkpoint automatisch den alten überschreibt, muss die Recovery-Information erhalten bleiben, bis ausdrücklich die Anweisung zum Verwerfen derselben gegeben wird.

Leader/Follower Replikation:

Das Verhalten beim Leader/Follower-Modell in Verbindung mit Recovery-Blocks kann optimiert werden, so dass die Follower niemals von einem Software-Fehler betroffen sind und daher auch keine Recovery-Points erstellen oder verschiedene Alternativen ausführen müssen. Dazu suspendieren sich die Follower, sobald ein Recovery-Block erreicht wird während der Leader den Block durchläuft. Der Leader schickt die Information, welche Alternative auszuführen ist an die Follower, die aufgrund des Replikations-Determinismus mit der gleichen Alternative ebenfalls erfolgreich sein müssen.

Dieses Schema wird durch die notwendige Möglichkeit verkompliziert, dass der Leader während der Ausführung eines Recovery-Blocks Synchronisations-Nachrichten an seine Follower schicken kann. Schickt der Leader während der Ausführung einer Alternative Nachrichten an seine Follower und schlägt die Alternative fehl, sind die Nachrichten ungültig. Da diese Nachrichten zwar versendet, aber noch nicht konsumiert sind, muss bei der Wiederherstellung eine weitere Nachricht versandt werden, die die Follower veranlasst, die Nachrichten zu verwerfen.

N-Version Programming lässt sich ebenfalls leicht in Delta-4 einbinden. Die einfachste Möglichkeit ist, ein komplettes N-Version Modul in ein Delta-4 Objekt zu verpacken, das dann repliziert wird. Der Overhead dieser Variante ist jedoch absehbar sehr groß (mehrere Versionen, die jeweils in mehreren Replikaten laufen).

Ein alternativer Ansatz ist, in jedem Replikat eines Objekts eine andere Version des N-Version Moduls laufen zu lassen. Die Ausgabe der verschiedenen Replikate wird an den Schiedsrichter gesendet, der wiederum ein Delta-4 Objekt ist (und somit repliziert werden kann). Der Schiedsrichter repräsentiert das N-Version Modul gegenüber dem Rest des Systems. Im Zuge der benötigten Symmetrie gehen Nachrichten an das Modul ebenfalls über den Schiedsrichter.

5. Fehlertoleranz am Beispiel der TTA

5.1. Überblick über die TTA

Die Time Triggered Architecture hat ihren Ursprung im MARS-Projekt der Technischen Universität Berlin sowie den verschiedenen Implementierungen der MARS-Architektur durch die Technische Universität Wien, da sich herausstellte, dass eine hardware-unterstützte, fehlertolerante Uhrensynchronisation einen fundamentalen Teil einer zeitgesteuerten Architektur darstellt. Im Rahmen des European PDCS (Parallel and Distributed Computing and Systems) Projekts entstand schließlich der erste Prototyp der TTA.

Das Prinzip hinter der TTA ist die Zerlegung einer großen Echtzeitanwendung in beinahe autonome Einzelteile, die auf einer gemeinsamen Zeitbasis laufen. Die gemeinsame Zeitbasis dient der Vereinfachung der Kommunikation, der Fehlererkennung und garantiert die Rechtzeitigkeit von Echtzeitanwendungen.

5.2. Fehlertolerante Zeitbasis in jedem Knoten

Computersysteme arbeiten prinzipbedingt auf einer diskreten Zeitbasis. Da zwei Uhren niemals perfekt synchronisiert werden können, besteht immer die Möglichkeit, des folgenden Fehlers:
Uhr im Knoten j tickt, Ereignis e tritt ein, Uhr in Knoten k tickt.
In diesem Fall würde das Ereignis e in den beiden Knoten verschiedene Timestamps erhalten.

Die TTA löst dieses Problem, indem es eine sogenannte „sparse time base“ verwendet. Diese besteht aus einer unendlichen Folge von sich abwechselnden Intervallen der Aktivität und der Stille. Alle Ereignisse, die während eines Intervalls der Aktivität auftreten, werden als gleichzeitig angesehen. Die Architektur muss sicherstellen, dass selbst erzeugte Ereignisse wie das Senden einer Nachricht nur während eines Aktivitäts-Intervalls auftreten. Externe Ereignisse werden durch ein Einigungsprotokoll einem Intervall der Aktivität zugeordnet. Somit ist sichergestellt, das jedes Ereignis im gesamten System die gleiche Timestamp erhält.

Illustration 3: "sparse time base"

5.3. Fault-Tolerant Units (FTUs)

Die TTA definiert die Verwendung von aktiver Replikation in Form von sogenannten Fault-Tolerant Units. Es wird sowohl die Verwendung von mehreren Knoten in Verbindung mit einem Voter definiert, als auch die Kombination zweier „fail-silent“ Knoten, hier „self-checking nodes“ genannt. Bei der Verwendung eines Voters muss jeder Knoten zusätzlich periodisch seinen internen Zustand ausgeben, der ebenfalls die Voting-Prozedur durchläuft. Dadurch kann sich ein neuer Knoten integrieren, indem er einen solchen Zustand annimmt.

5.4. Never-Give-Up (NGU) Strategie

Sollten zu viele Fehler auftreten, um sie tolerieren zu können, also der minimal benötigte Dienst nicht mehr erbracht werden können, so wird eine Never-Give-Up Strategie aktiviert. Diese hängt stark von der Anwendung ab, so könnte z.B. ein Fahrzeug sicher zum Stillstand gebracht werden, falls das Bremssystem die eigene Funktion nicht mehr garantieren kann.

6. Literaturverzeichnis

- Barret P. A. , Speirs N. A. 1993: Towards an integrated approach to fault tolerance in Delta-4 *IEE Distributed Systems Engineering, Volume 1, Issue 2*, pp 59-66, IOP Publishing Ltd.
- Kopetz H., Bauer G. 2003: The Time-Triggered Architecture *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software,* pp 112-126