Fehlertoleranz durch Active Replication

1 Grundprobleme

(1) **Konsistenter Normalbetrieb**

Im Normalbetrieb muß gewährleistet sein, daß "die Applikationen nicht auseinanderdriften".

(2) **Fault Detection**

Technik zur Fault Detection hängt von der Fehlerart ab (Fail Stop vs. Byzantine)

(3) **Recovery**

Bzgl. der Recovery ist folgendes zu berücksichtigen:

- Bei Byzantinischen Fehlern ist die fehlerhafte Komponente zu isolieren.

  Bei intern nicht erkannten Fehlern ist die fehlerhafte Komponente durch die anderen zu isolieren. Bei byzantinischen Fehlern erfordert dies ein spezielles Protokoll, nach dem sich die Komponenten auf "isolieren/nicht isolieren" einigen: alle fehlerfreien Komponenten müssen zur selben Entscheidung kommen. (Dies ist möglich, führt aber nicht bei jedem byzantinischen Fehler auf die Entscheidung "isolieren" .)
- Geeignete Rekonfigurationsmaßnahmen müssen getroffen werden, z.B. bzgl. I/O.
- Fehlerbehebung muß geeignet durchgeführt werden, etwa die Korrektur fehlerhafter Ausgaben.[1]

(4) **Reintegration**

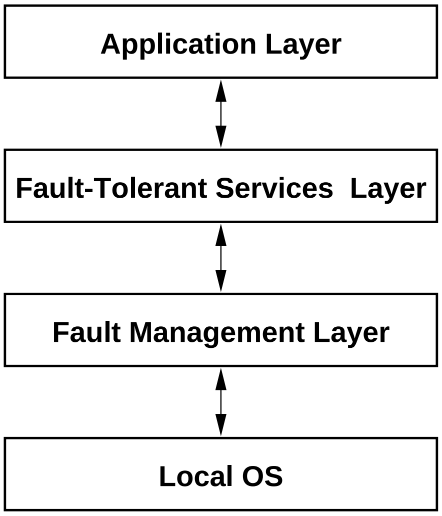
Bei der Reintegration ist folgendes zu beachten:

- Die reparierte Komponente muß in einen konsistenten Zustand gebracht werden.
- Ein geeigneter Aufsetzpunkt für weitere Inputs muß gefunden werden.

[1] diese Maßnahme ist u.U. zeitkritisch, z.B. wenn korrigierte Werte nur innerhalb eines bestimmten Zeitrahmens akzeptiert werden.

2 Standard-Architektur für fehlertolerante Systeme

Das folgende Bild gibt die Standardarchitektur für fehlertolerante Systeme wieder.



Das **Fault-Tolerant Service Layer** stellt OS Services zur Verfügung, die von den Applikationen wie Services des lokalen OS genutzt werden können, aber der Verwaltung des FML unterliegen, bevor sie auf die eigentlichen Services abgebildet werden. Das **Fault Management Layer** ist für die die Realisierung der Lösungen für die in Abschnitt 1 genannten Grundprobleme zuständig.

3 Anforderungen an die Applikation

(A.1) Die Applikation muß am Interface das Verhalten einer deterministischen Zustandsmaschine zeigen.[2]

Das bedeutet unter anderem:

- Keine Entscheidung ist abhängig von der lokalen Clock.
- Zufallsalgorithmen werden nicht verwendet.
- Outputs verwenden keine Werte der lokalen Clock

[2]Bei gewünschtem Nichtdeterminismus muß ein anderer Ansatz gewählt werden, z.B. die Verwendung von *self-stabilizing algorithms*

(A.2) Gleiche Inputfolgen bewirken gleiches I/O-Verhalten und gleiche Zustandsübergänge.

Dabei sind als "Inputs" zu verstehen u.a.

- messages vom Lane Interface,
- messages von anderen lokalen Applikationen,
- globale Daten,
- Time und Timeouts,
- exceptions,
- interrupts,
- Rückmeldungen von Betriebsmitteln.

Hinweis: Für einen systematischen Ansatz empfiehlt es sich, ein Objektmodell für die wesentlichen Inputs aufzubauen.

(A.3) Die Zustandsvariablen müssen wohldefiniert sein und für das FML erreichbar sein.

Dies wird z.B. für die Rekonfiguration während der Reintegration relevant sein.

(A.4) Die Verarbeitung muß transaktions-orientiert sein.

Insbesondere dürfen zwischen `EndOfTransaction` und `StartOfTransaction` keine I/O oder Zustandsänderungen erfolgen.

4 Anforderungen an das Fault Management Layer

(FML.1) Atomic Broadcast für alle Applikationsinputs (vgl. (A.2))

Motivation:

- gleiche Empfangsreihenfolge mehrerer multicast-Messages (muß durch ein geeignetes Multicast-Protokoll gewährleistet sein)
- Empfangsausfall, falls einer der redundanten Prozesse nicht empfangsbereit (hieraus ergeben sich auch zusätzliche Anforderungen auch an die Rekonfigurationsphase).

(FML.2) Fehlererkennung (Fail Stop und Byzantinisch)

Fehlerannahmen:

- Eine Komponente selbst kann eigene byzantinische Fehler nicht feststellen.
- Eine Komponente kann "böswillig" sein, d.h. den anderen Komponenten verschiedene Daten übermitteln.

Grundprinzip: Fehlererkennung durch Voting auf Basis der Redundanz der Daten.

Bei nicht-byzantinischen Fehlern genügt 3-fach-Redundanz, bei byzantinischen Fehlern 4-fach-Redundanz bei der zusätzlichen Fehlerannahme, daß nur eine Komponente "böswillig" ist.

(FML.3) Recovery (1): Isolierung der fehlerhaften Komponente

Dazu werden Operationen benötigt für

- Eigenfehlerüberwachung und Selbstisolierung
- Isolierung externer Komponenten[3]

(FML.4) Recovery (2): Rekonfiguration

Rekonfiguriert werden müssen u.a.:

- atomic broadcast ("runterschalten" auf die reduzierte Anzahl der Lanes)
- Voting Operationen für die Fehlererkennung, die u.U. auch abhängig sind von der Anzahl der Lanes
- I/O, z.B. wenn die eine Lane, die Daten versendet (übliches Vorgehen) ausfällt und Ausgabe auf eine andere Lane übertragen werden muss.

(FML.5) Reintegration (1): "Suche" nach Konsistenzpunkten

Hierzu gibt es verschiedene Ansätze:

- dynamisch, etwa bei Datenbanken, wenn keine der Transaktionen mehr offen ist
- zu festgelegten Zeitpunkten im Ablauf des Systems.

(FML.6) Reintegration (2): Kopie der Zustandsdaten

[3] für diese Operation müssen die Fehlerhypothesen berücksichtigt werden, etwa, daß bei einem 4-Lane Betrieb höchstens eine der Komponenten byzantinisch ist.

5 Anforderungen an die Fault Tolerant Services

(FTS.1) Konsistenter Clock-Service/Timeout-Service zwischen den Lanes.

Bsp.: Clock-Synchronisierung durch Broadcast, für das eine Komponente zuständig ist. Probleme hierbei:

- Byzantinische Fehler bei der Synchronisation müssen ausgeschlossen sein.
- Bei Wechsel des Clock-Servers im Fehlerfall werden Anforderungen an die Recovery-Zeit im Verhältnis zur Clock-Drift gestellt!

(FTS.2) ADTs für globale Daten

Die Operationen auf den Daten werden über das FML synchronisiert. Hierbei ist es wichtig, daß alle Operationen auf globalen Daten zusichern, daß auf allen Lanes äquivalente serialisierbare Schedules (für Lesezugriffe) gewährleistet sind.

(FTS.3) Dienste für "message send", "message receive"

Ziel: Konsistenz in allen Lanes

(FTS.4) Spezielle Dienste "StartOfTransaction", "EndOfTransaction"

z.B. für Broadcasts, Rekonfiguration, Reintegration, Sendewiederholungen im Zuge der Rekonfiguration.

6 Eine Fehlertolerante Lösung mit synchronem System

Definition: Ein System heißt **synchron**, wenn für alle sichtbaren Aktionen die Antwortzeit bekannt und beschränkt ist

In synchronen Systemen sind einige Dinge einfacher, z.B.:

- Erkennung von Konsistenzpunkten
- Fehlererkennung
- Uhrenbehandlungen

Synchrone Systeme werden mit Hilfe von synchronen Kommunikationsprotokollen realisiert. Diese Protokolle lösen das Reihenfolgeproblem direkt. Ein Beispiel für ein solches Protokoll ist das im folgenden vorgestellte **Frame Protokoll**.

Datenstrukturen zwischen den Layern

In einem **Frame** werden mehrere **Pakete** in sogenannten **Subframes** versendet. Die Anzahl der Subframes pro Frame ist eine feste Größe. Die Kommunikation findet über einen Bus statt.[4] Busprotokolle werden über die Pakettypen spezifiziert, die u.a. festlegen

- wer sendet,
- was der Inhalt der Nutzdaten ist.

In Bild 1 sind die im folgenden erläuterten Strukturen schematisiert dargestellt.

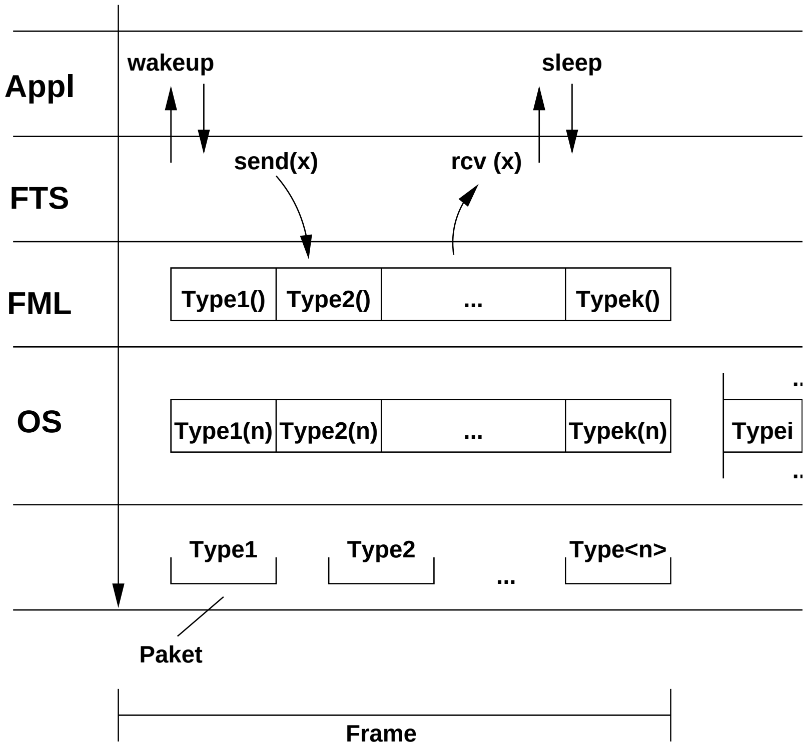


Abbildung 1: Datenstrukturen zwischen den Layern

- Die Busprotokolle legen fest, in welcher Reihenfolge die Pakete verschiedenen Typs (`Type1,...,Type<n>`) im Frame auftreten. Dies erlaubt den Anwendungen einen gezielten Zugriff auf die relevanten Daten.

[4] im Allgemeinen werden für diese Form von Systemen keine Punkt-zu-Punkt-Kommunikationen verwendet; Applikationen legen über die Services Daten auf den Bus, wo sie von anderen Applikationen auch wieder über die Services zugänglich sind.

- Im OS-Layer befindet sich neben dem Feld für die Pakete (`OS-Puffer`) auch eine Tabelle mit Zuordnungen wie auf die verschiedenen Typen zugegriffen werden soll (reiner Lesezugriff: `out`, reiner Schreibzugriff: `in`). Darüber wird geregelt, ob der Inhalt des betreffenden Feldes auf den Bus geschrieben werden soll und/oder der Businhalt in das Feld geschrieben werden soll.
- Datenaustausch zwischen `FML-Puffer` und `OS-Puffer` findet zu geeigneten Zeitpunkten statt (s.u.).
- Die Services `send()` und `rcv()` kopieren Applikationsdaten in den FML-Puffer resp. lesen sie aus dem Puffer aus

Zeitverhalten

Das Zeitverhalten des Systems orientiert sich an den Frames. Genauer wird der Framezyklus zur Synchronisierung der lokalen Clocks verwendet.[5] Im Folgenden wird das Zeitverhalten genauer analysiert; eine graphische Darstellung findet sich in Bild 2.

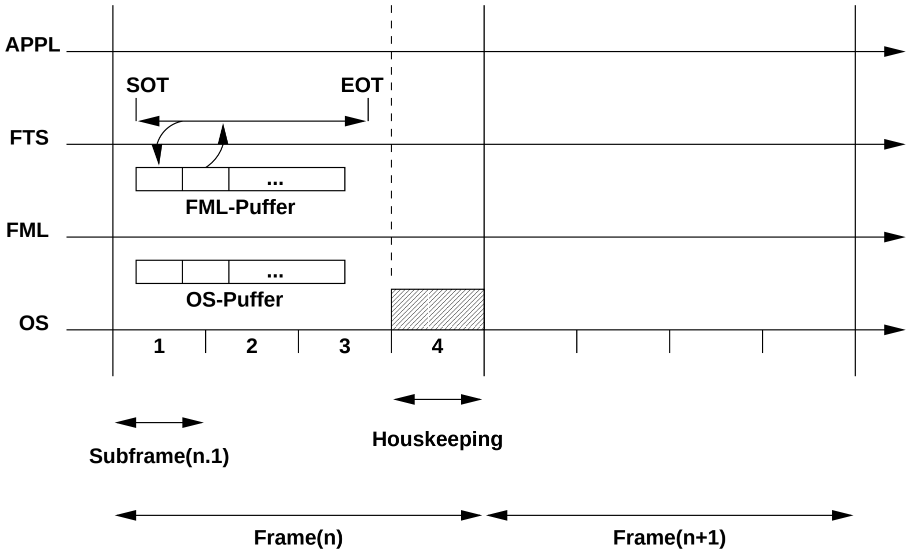


Abbildung 2: Timing Diagramm

- Während `Frame(n)` können die Applikationen nur auf die Daten zugreifen, die in `Frame(n-1)` auf dem Bus waren. Diese Daten liegen dann im FML-Puffer.

[5]u.U. ist die dadurch erreichte Zeitgranularität zu grob für Hard-Realtime Anwendungen!

- Austausch der Daten zwischen OS und Bus bzw zwischen FML und Applikation müssen vor dem letzten Subframe stattfinden. Insbesondere muß das Interval zwischen `StartOfTransaction` (`SOT`) und `EndOfTransaction` (`EOT`) innerhalb der Subframes ohne den letzten Subframe liegen.
- Die Anwendungsdaten x_m stammen aus `Frame(n-1)`.
- Das Umkopieren der Daten findet in dem "Housekeeping"-Subframe statt. Dadurch kommt es nicht zu Konflikten. Der Typ des Pakets bestimmt dabei die Aktion:
  - `in`: Kopie von OS-Puffer nach FML-Puffer
  - `out`: Kopie von FML-Puffer nach OS-Puffer
  - `in/out`: Swap zwischen OS-Puffer und FML-Puffer

7 Das Problem der Byzantinischen Generäle

Siehe:

L. Lamport, R. Shostak, and M. Pease, *The Byzantine Generals Problem*, In: ACM Transactions on Programming Languages and Systems, Vol.4, Nr. 3, (1982)