ATTEMPTO:

Eine modulare portable Betriebssystem-erweiterung für Fehlertoleranz

Implementation und empirische Validation

Der Technischen Fakultät der
Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR - INGENIEUR

vorgelegt von

Willi Günter

geb. am 27.12.1954 in Fulda

Erlangen - 1992

Als Dissertation genehmigt von
der Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung :

Tag der Promotion :

Dekan :

Berichterstatter :

Inhaltsverzeichnis

TEIL I

TEIL II

Danksagung

Ich möchte hier ganz besonders Herrn Prof. Dr. Mario Dal Cin danken, sowohl für die Hauptbegutachtung wie auch für seine immer hilfreiche Unterstützung. Als Projektleiter des DFG-geförderten Projekts ATTEMPTO war er auch maßgeblich an der Grundkonzeption beteiligt. Dies gilt genauso für Herrn Dr. Rüdiger Brause, akademischer Oberrat an der Universität Frankfurt, dem besonders für zahlreiche Diskussionen und Anregungen zu danken ist.

Für die Zweitbegutachtung bin ich Herrn Prof. Dr. Fridolin Hofmann sehr dankbar.

Auch meiner Lebensgefährtin Inge Semmler, die mich mit viel Verständnis während des Entstehens dieser Arbeit unterstützt hat, möchte ich hiermit meinen Dank abstatten.

Erlangen, im November 1992

Überblick

Die vorliegende Arbeit stellt das Fehlertoleranzkonzept und die Implementation des fehlertoleranten Multiprozessorsystems ATTEMPTO vor und nimmt eine empirische Bewertung desselben vor anhand eines realisierten Prototypen. Das System zeichnet sich aus durch einen sehr hohen Freiheitsgrad in der Verwendbarkeit, sowohl hinsichtlich möglicher Standardanwendungen unter einem weitverbreiteten Betriebssystem wie auch in der Adaptierbarkeit seiner Mechanismen für Fehlertoleranz auf andere Betriebssysteme (auch für ganz andere Anwendungsbereiche) und Standard-Hardwarekomponenten. Die Fehlertoleranz ist dabei eine erweiterte Betriebssystemfunktion und damit transparent für den Systembenutzer, der zudem noch die Möglichkeit zur graduellen Differenzierung seiner Zuverlässigkeitsanforderungen erhält, die das System durch eine automatische Redundanzverwaltung für ihn erfüllt. Im besonderen kann der Benutzer Parallelität und Fehlertoleranz als gleichberechtigte Systemeigenschaften gegeneinander aushandeln: er trifft die Wahl zwischen hoher Systemleistung oder hoher Systemzuverlässigkeit. Die Transparenz bezieht sich auf unveränderte Binärprogramme des Monoprozessors, die ebenso im Fehlertoleranzbetrieb lauffähig sind. Diese Eigenschaft und daß zugleich mit der graduellen Abstufung des zugrundeliegenden Fehlermodells hinsichtlich Mehrfachfehler eine relativ fein abgestufte System-Zuverlässigkeit bis hin zu hohen Zuverlässigkeiten möglich wird, sind besondere Eigenschaften dieses Systems im Vergleich mit anderen bekannten Konzepten.

Die Arbeit besteht aus zwei Teilen. Im ersten werden einige allgemeine Probleme fehlertoleranter Systeme untersucht, um Hintergründe und Grundlagen aufzuzeigen, die für die Realisierung des ATTEMPTO-Systems relevant sind. Dabei wird ein Grundwissen über die Grundlagen fehlertoleranter Systeme bereits vorausgesetzt. Der zweite Teil beschäftigt sich mit Konzept, Implementierung und Bewertung des eigentlichen Systems.

Zunächst konstatiert und untersucht eine Einleitung einen Entwicklungstrend bei kommerziellen fehlertoleranten Rechensystemen hin zu einer Mehrzweckfehlertoleranz. Eigenschaften einer idealen Allzweckfehlertoleranz werden aufgeführt und praktische Einschränkungen für die Architektur realer Systeme bedingt durch den Anwendungscharakter und seine Zuverlässigkeitsanforderungen werden herausgestellt. Die für weite Anwendungsgebiete gute Verwendbarkeit von software-implementierten Fehlertoleranzmechanismen auf der Basis von einer allgemeinen Hardwarearchitektur, wie sie durch lose gekoppelte Mehrrechnersysteme gegeben ist, wird motiviert. Im weiteren Verlauf geht der erste Teiles auf Redundanzkonzepte für fehlertolerante Systeme ein; dabei werden Grundkonzepte mit ihren Zuverlässigkeitszielen und dem erforderlichen Realisierungsaufwand vorgestellt, Alternativen für ein Fehlertoleranz-Design verglichen, und die Eignung einer software-implementierten Fehlermaskierung, die auf einer allgemeinen verteilten Systemarchitektur aufsetzt, für eine variable Mehrzweck-Fehlertoleranz begründet. Auf einige allgemeine Probleme der Fehlermaskierung wird eingegangen, hauptsächlich auf solche, die mit der Synchronisierung der Replikate zusammenhängen. Vergleichstestverfahren als Basis für Fehlerunterdrückung werden untersucht; dabei werden Diagnosemodelle klassifiziert und -algorithmen vorgestellt.

Der zweite Teil erläutert nach einer kurzen Vorstellung der Projekthistorie und des weiteren Projektumfeldes das Systemkonzept mit seinen Prämissen und die Systemarchitektur. Auf einige Implementationsdetails - zuverlässiger Multicast zur Interprozessorkommunikation, Ver-

bindung von neuen und alten Betriebssystemfunktionen, nachrichtengesteuerte Mechanismen für Fehlertoleranz - wird näher eingegangen, wie auch auf spezielle Probleme der Implementierung. Das Fehlermodell und die Fehlerbehandlung werden präzisiert.

Die Systembewertung wird in fünf Abschnitten vorgenommen: die Eignung der Standard-Hardware wird beurteilt, die Systemeigenschaften bzgl. der geforderten Modularität und der Nutzbarkeit validiert. Betrachtungen über die Zuverlässigkeit werden durchgeführt, und das Leistungsverhalten wird quantitativ analysiert.

Eine Zusammenfassung schließt die Arbeit ab mit einem Überblick über das Projekt und die erreichten Projektziele.

1. Einleitung

Konzepte für fehlertolerante Rechensysteme sind schon viele vorgestellt worden, dagegen sind aber weitaus weniger fertige Systemtypen im Einsatz. Trotzdem gilt es als unumstritten, daß Fehlertoleranztechniken mit dem Vordringen des Rechnereinsatzes in neue Anwendungsgebiete und auch wegen der zunehmenden Komplexität der Rechner immer mehr an Bedeutung gewinnen werden. Während auf Kodierungsredundanz basierende Fehlertoleranztechniken schon heute in fast allen Rechnern zur Anwendung kommen - man denke nur an fehlerkorrigierende Codes im Arbeitsspeicher und bei der Datenübertragung - sind dem Einsatz von massiver Redundanz aus praktischen Wirtschaftlichkeitsüberlegungen heraus Grenzen gesetzt. Dagegen findet man sehr viel häufiger die Vervielfachung ganzer Rechner, z.B. organisiert als verteiltes oder regulär strukturiertes massiv paralleles System, allein zur Erhöhung des Leistungsvermögens, nicht selten wohl auch deshalb, weil ein erhöhtes Angebot an Rechenleistung in der Regel einen gestiegenen Bedarf daran erzeugt. Solche Systeme besitzen damit inhärent, d.h. zumindest prinzipiell, ein für Fehlertoleranz nutzbares Reservoir an struktureller Redundanz. Meist fehlen nur die Mechanismen zum kontrollierten Einsatz für diesen Zweck. Wenn es gelänge, solche Mechanismen nachträglich einzubringen, wären diese Systeme auch für fehlertolerante Anwendungen offen. Gleichzeitig könnten mit diesem erweiterten Dienstleistungsangebot auch neue Anwendungsgebiete für Fehlertoleranz erschlossen werden. Wichtig ist dabei, daß dem Benutzer die Wahl zwischen Parallelität und Fehlertoleranz bleibt, und zwar mit der Möglichkeit der graduellen Differenzierung. Diese Wahlfreiheit bei nicht vorher festgelegter Zweckgebundenheit des Rechensystems geht einen Schritt in die Richtung auf ein parallelverarbeitendes Allzweck-System. Dieses Charakteristikum verlangt in erster Linie nach einer geeigneten Architektur für leistungsfähiges Parallelrechnen. In Bezug auf Fehlertoleranz stehen hier anpassungsfähige Zuverlässigkeitseigenschaften durch die variable Nutzbarkeit der Fehlertoleranzmechanismen im Vordergrund. Einer ***Allzweck-Rechnerarchitektur*** und einer ***Allzweck-Fehlertoleranz*** sind verschiedene Teilaspekte gemeinsam.

1.1 Aspekte einer Allzweck-Fehlertoleranz

Eine Allzweck-Fehlertoleranz charakterisiert ein Konzept für Fehlertoleranz, das von der jeweils aktuellen Systemkonfiguration und auch der Anwendung abstrahiert. Interessant ist natürlich die Fragestellung, ob ein solches Universalkonzept für Fehlertoleranz überhaupt definiert werden kann, oder ob dies wenigstens innerhalb gewisser Grenzen möglich ist. Dieser "***General Purpose***"-Aspekt der Fehlertoleranz ist schon verschiedentlich untersucht worden bzw. hat Einzug gefunden in praktischen Systemrealisierungen [MAN90], hier jedoch in der Regel nur begrenzt auf bestimmte traditionelle Anwendungsgebiete und bei statischer Konfigurierung innerhalb gewisser Zuverlässigkeitsklassen.

Ähnlich wie bei der Frage nach einer "General Purpose"-Parrallelrechner-Architektur sind auch bei der "General Purpose"-Fehlertoleranz unterschiedliche sich teilweise widersprechende Anfordungsprofile zu vereinigen und somit praktische Grenzen gesetzt (Abb. 1). Als wesentliches Merkmal ist beiden gemeinsam die ***Variabilität*** in der Systemnutzbarkeit, die die eigentliche Zweckungebundenheit ausmacht. Daneben ist von größter Wichtigkeit die leichte ***Handhabbarkeit***. Sie bestimmt wesentlich die Akzeptanz und damit Erfolg und Popularität des Konzepts. Dahinter steckt weniger der Gedanke an eine leistungsfähige Benutzeroberfläche, die natürlich bei interaktiven Systemen ebenfalls von großer Bedeutung ist, als vielmehr die Klarheit der Programmierung von Anwendungen. Implizite Mechanismen, die sich für den An-

wendungsprogrammierer transparent darstellen, sind hier gefragt

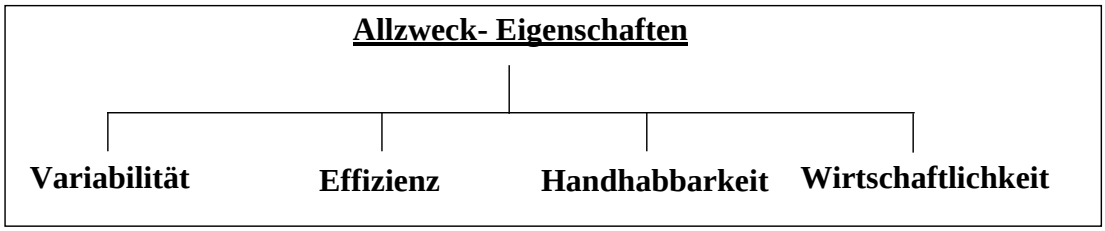


Abb. 1 Allgemeine Merkmale eines Allzwecksystems

Der Aspekt der ***Zeit-Effizienz*** ist nicht zu vernachlässigen, da er die Benutzerakzeptanz sehr stark dominiert. Während Effizienz für parallelverarbeitende Systeme das eigentliche Entwicklungsziel ist, stehen bei fehlertoleranten Systemen natürlich in erster Linie Zuverlässigkeitseigenschaften im Vordergrund. In einer wichtigen Domäne der Anwendung von Fehlertoleranz, im Bereich der Prozeßautomatisierung, können aber häufig nur effiziente Fehlertoleranz-Mechanismen die geforderten Rechtzeitigkeitsbedingungen erfüllen. Zuverlässigkeit und Leistungsverhalten eines fehlertoleranten Systems stehen in einem engen Verhältnis zueinander. Es ist i.d.R. ein diametrales, da die für Redundanz gebundene Parallelität zur Leistungssteigerung entfällt. Effizienz kann zum Beispiel auch im Gegensatz stehen zur Transparenz; für bestimmte - aus der Menge der möglichen sogar die meisten - Anwendungsfälle mag es vorteilhaft sein, vom vorgefertigten impliziten Kontrollmechanismus abzuweichen zugunsten eines speziell angepaßten expliziten Ansatzes [DC88]. .

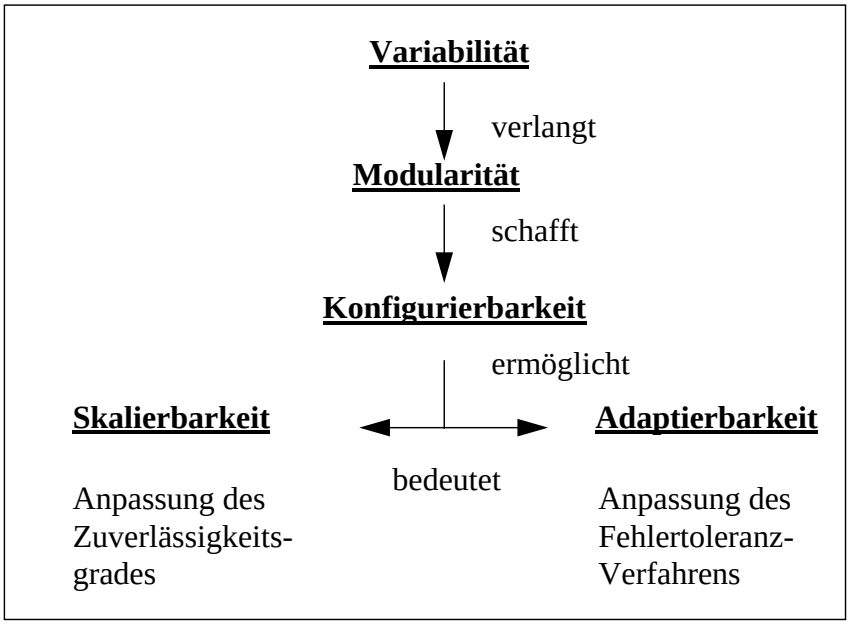


Abb. 2 Strukturmerkmale für Variabilität in Bezug auf eine Allzweck-Fehlertoleranz

Hinter der Variabilität verbirgt sich im einzelnen die graduelle und strukturelle Anpassungsfähigkeit der Systemarchitektur. ***Adaptierbarkei***t ist die Anpassungsfähigkeit der Hardware-Architektur an die Anwendungsstruktur; damit ursächlich verbunden ist die Frage nach einer geeigneten Vernetzung der System-Komponenten. Für Parallelität ist die Strukturierung eine geeignete Aufteilung des Anwenderprogramms auf parallel arbeitende Funktionskomponenten, um optimal schnelles Rechnen zu ermöglichen. Für die Fehlertoleranz ist damit die Verschal-

tung von replizierten Funktionskomponenten zusammen mit koordinierenden Funktionseinheiten gemeint, um Fehlerdiagnose und -behandlung durchführen zu können. Die graduelle Anpassungsfähigkeit ist gekennzeichnet durch die ***Skalierbarkeit***, die sich auf die Ausbaufähigkeit des Systems bezieht: das System soll mit den Aufgaben wachsen können, dabei aber weder in Leistungsvermögen noch Zuverlässigkeit degradieren. faßt zusammen, was die Architektureigenschaften Skalierbarkeit und Adaptierbarkeit in Hinblick auf eine Allzweck-Fehlertoleranz bedeuten.

Skalierbarkeit und Adaptierbarkeit ist wie überhaupt jede Art von ***Konfigurierbarkeit*** ohne ***Modularität*** nicht denkbar. Eine planvolle Kooperation der Einzelmodule setzt eine geeignete Vernetzung dieser voraus. Im allgemeinen wird unter Konfigurierbarkeit statische, d.h. Off-Line-Modifizierbarkeit verstanden. Dies reicht aber für General-Purpose-Systeme nicht aus, eben aus Gründen der leichten Handhabbarkeit. Das beste Allzwecksystem ist das, welches sich automatisch an die Erfordernisse der Anwendung anpaßt. Wegen der großen Variabilitätsbandbreite sind an die Grund-Vernetzung extreme Anforderungen gestellt. Handelt es sich um homogene Systeme, ist im Idealfall eine vollständige Vernetzung notwendig, da nur sie die Abbildung der logischen Anwendungsstruktur auf die realen physikalischen Verhältnisse nicht behindern kann. Bei hohen Zuverlässigkeitanforderungen, insbesondere wenn auch noch hohe Leistung erzielt werden soll, reicht eine vollständige Vernetzung allein auf logischer Ebene, d.h. verdeckt realisiert über Routingstationen, nicht aus, z.T. aus Gründen der Fehlerisolation und vor allem wegen eingeschränkter Rekonfigurationsmöglichkeiten. Vollständige physikalische Vernetzung kostet aber hohe Aufwendungen, insbesondere bei großer Komponentenzahl. Rein konstruktiv wird sie auch Erweiterungen erschweren.

Im Zusammenhang mit Adaptierbarkeit stehend, wird sich eine Allzweck-Fehlertoleranz daran messen lassen müssen, inwieweit unterschiedliche Zuverlässigkeitsanforderungen mit ein- und derselben Basis-Systemarchitektur zu erfüllen sind. Zuverlässigkeit als Allgemeinbegriff bedarf weiterer qualitativer wie auch quantitativer Präzisierung. So verlangen die unterschiedlichen Projektierungsziele

- hochwahrscheinliches Überleben im projektierten Einsatzzeitraum,
- lange Lebensdauer,
- hohe Verfügbarkeit und/oder
- hohe Sicherheit

unterschiedliche Berücksichtigung in der Systemarchitektur vor allem hinsichtlich des Redundanzeinsatzes. Statische und dynamische Redundanz schafft die Voraussetzung für Fehlerunterdrückung bzw. -erholung. Neben der reinen Wirkredundanz - das sind die ggf. (identisch oder diversitär) replizierten Funktionskomponenten, die den eigentlichen Systemdienst erbringen können- gehören je nach verwendetem Fehlertoleranzverfahren verschiedene Zusatzkomponenten als Prüfredundanz. Dies bezeichnet Instanzen, die die Wirkkomponenten koordinieren bzw. kontrollieren, und so etwa für Fehlererkennung, Rekonfiguration oder Maskierung zuständig sind. Ein Universal-Architekturkonzept muß daher eine gewisse Menge an Grundeinheiten definieren, die flexibel miteinander kombiniert werden können. Die eigentliche Systemrealisierungsaufgabe besteht dann darin, die notwendigen Funktionskomponenten auf diese Grundmenge der Basisarchitektur mit einem geeigneten Automatismus abzubilden. Die Abbildung wird dann aber möglicherweise nur eine Untermenge erfassen, d.h. es werden Komponenten brachliegen.

Betrachtet man alle Komponenten als in Hardware ausgeführt - in diesem Fall spricht man von

hardwareimplementierter Fehlertoleranz - , widerspricht eine solche u.U. ineffiziente Ausnutzung jedoch dem ***Wirtschaftlichkeitsprinzip***, das bei einem General-Purpose-Konzept für eine allgemeine Akzeptanz ebenfalls von großer Bedeutung ist. Hierfür wäre eine möglichst einfache Hardware-Grundarchitektur von Vorteil, auf der nicht nur die Anwendung - wie z.B. beim klassischen Von-Neumann-Allzweckrechner - programmgesteuert abläuft, sondern auch die Fehlertoleranzmechanismen selbst. Dies entspräche einer ***Softwareimplementation der Fehlertoleranz***. Um die zugrunde liegende Hardwarearchitektur für die geforderte Variabilität sowohl der Parallelität als auch der Fehlertoleranz nutzen zu können, ist eine flexible Betriebssoftware nötig. Geht man von makroskopischen Funktionskomponenten aus - z.B. ganzen Rechnern als kleinste ersetzbare Einheiten bzw. Erweiterungsmodule -, und verzichtet man auf zuverlässigkeitskritische Zentralisierung, realisiert diese Software ein verteiltes Betriebssystem. So kann auch die Betriebssoftware modular mitwachsen. Der Verteilungscharakter trägt mit zur Robustheit bzw. sogar Fehlertoleranz der eigentlichen Fehlertoleranzmechanismen bei, wenn die Redundanz entsprechend genutzt wird. Damit das Allzweckkonzept nur auf einer allgemeinen Hardwarearchitektur fußt, nicht aber auf einer speziellen Hardware selbst, muß auf die ***Portabilität der Betriebssoftware*** geachtet werden.

1.2 Praktische Einschränkungen für den Allzweck-Charakter

Betrachten wir exemplarisch verschiedene Anwendungsbereiche der Fehlertoleranz, z.B.

- Fehlertoleranz in einer verteilten Anwendung in einem inhomogenen Netz von autonomen Rechnern. Hier geht es vor allem um hohe Verfügbarkeit von speziellen Systemdiensten (z.B. Plattenserver) und um Datenintegrität,
- Fehlertoleranz bei massiver Parallelität in regulären MIMD-Strukturen (Hochleistungs-Parallel-Rechnern). Im Vordergrund steht hier die Absicherung des Programmlaufs bei langlaufenden rechenintensiven Programmen (z.B. Simulationen),
- Fehlertoleranz und Echtzeit, wo hohe Zuverlässigkeit, Verfügbarkeit und/oder Sicherheit für Steuerungsaufgaben gefordert wird,

so wird deutlich, welche Probleme die zuvor beschriebenen Ansätze *Universalarchitektur* und *Softwareimplementation der Fehlertoleranz* aufwerfen. Die anwendungsbezogen Grundarchitekturen nämlich differieren allein schon so stark, daß an ein einheitliches Einbringen von redundanten Komponenten nicht zu denken ist. Darüberhinaus zeigt das letzte Beispiel, daß auch Softwareimplementation allgemein den Anforderungen der Anwendung trotz ihrer Flexibilität nicht unbedingt genügen wird, in diesem Fall aus Effizienzgründen.

Somit führen also beide Architekturansätze für eine Allzweckfehlertoleranz nicht zum Ziel: die automatisch konfigurierbare Hardwareimplementation scheidet aus aus Gründen der Wirtschaftlichtkeit und Komplexität, die Softwareimplementation genügt vielen Anwendungen nicht, da sie strukturelle (Wirk-)Redundanz nur einsetzen, nicht aber ersetzen kann.

Deshalb schränkt man die Allzweck-Fehlertoleranz auf nurmehr eine Mehrzweck-Fehlertoleranz ein durch

- Einschränkung auf bestimmte Anwendungsgebiete (Echtzeit-, Transaktionssysteme, ...),
- Festlegung auf ein bestimmtes Redundanzgrundkonzept (statisch, dynamisch),

- Verzicht auf transparente Redundanzverwaltung (explizite, auch manuelle (Re-)Konfiguration) unter Bereitstellung der Fehlertoleranz-Mechanismen für den Anwender,
- Beschränkung der Systemgröße,
- Ausrichtung auf ein bestimmtes Zuverlässigkeitsverhalten (entweder hochsicher oder hochverfügbar, ...),
- grobabgestufte Zuverlässigkeitsklassen (wahlweise z.B.: nicht-fehlertolerant, zuverlässig, oder hochzuverlässig, ...),
- vereinfachte Fehlermodelle.

Diese Aufgliederung ist weder vollständig noch sind die Gliederungpunkte voneinander scharf abgegrenzt. Insbesondere sind die Einflüsse des Fehlermodells zu beachten, das jeder Klassifizierung zugrundeliegt. So besitzt z.B. eine Echtzeitanwendung im Vergleich zu nicht zeitkritischen Anwendungen ein erweitertes Fehlermodell, das auch nicht-zeitgerechtes Ergebnisverhalten miteinbezieht. Ein weiteres Beispiel: verzichtet man auf strukturelle Redundanz, sind permanente Fehler nicht tolerierbar.

1.3 Entwicklungstrends zur Mehrzweck-Fehlertoleranz

Eine gewisse Flexibilität als wichtigstes Grundmerkmal einer Mehrzweck-Fehlertoleranz findet man bei vielen Systemkonzepten für kommerzielle oder experimentelle fehlertolerante Systeme. Bespielsweise gibt es modulkonfigurierbare hardwareimplementierte Systeme mit dediziertem Anwendungsbereich (FTMP [HSL78], JPL-STAR [Av71]). [SIM90],[MPR87] sind Prozeßsteuerungssysteme, die mithilfe eines Baukastenprinzips unterschiedliches Zuverlässigkeitsverhalten projektierbar machen. SIFT [Wen78] ist ein Pionierprojekt für softwareimplementierte Fehlertoleranz und benutzt Standardkomponenten, komplette Rechner, die jedoch in einer relativ starren Kommunikationsstruktur verschaltet sind und so die Erweiterbarkeit behindern. STRATUS [Frei82] vernetzt an sich schon fehlertolerante Rechnerkomponenten für verteilte Transaktions-Applikationen, während für das gleiche Anwendungsgebiet die Systeme TANDEM Non-Stop [Bart81] und AURAGEN [BBG83],[Glaz84] die verteilte Systemumgebung selbst als dynamischen Redudanz-Pool nutzen. Die beiden letztgenannten verwenden Formen von Checkpointing, also das Prinzip der Fehlerbehebung. TANDEM garantiert dabei Transparenz nur für die eigentliche Datenbankanwendung. AURAGEN (bzw. der Nachfolgetyp TARGON32 [Borg89]) geht in Hinblick auf Mehrzweck-Fehlertoleranz erhebliche Schritte weiter. Fehlertoleranz ist hier ein Betriebssystemdienst und das System bietet dem Benutzer eine Standard-Betriebssystemschnittstelle (UNIX- Kompatibilität). Auch die Systeme Sequoia [Bern88] und das neuere TANDEM -System Integrity S2 [Jew91] bieten dies an. SEQUOIA verwendet als Fehlertoleranzverfahren Verdoppelung mit selbsttestenden Einheiten, Integrity S2 ist ein fehlermaskierendes System. Am Beispiel TANDEM kann man erkennen, daß offenbar die Kosten für massive strukturelle Redundanz (Verdreifachung von Prozessoreinheiten, Verdoppelung von Speichern) zunehmend in Kauf genommen werden zugunsten der Vorteile Transparenz und Standardanwendung. Eine Analyse fehlertoleranter Systeme [Ser89] zeigt überhaupt sehr deutlich, daß Hardwareaufwendungen im Vergleich zu den sehr viel höheren Entwicklungskosten für (oft redundanzsparende) Software-FT-Mechanismen eine untergeordnete Rolle spielen. Verglichen mit dem Marktführer TANDEM, der mit seinem Non-Stop-System sehr früh auf dem Markt present war, hatten fast alle Mitbewerber nur geringen wirtschaftlichen Erfolg zu verzeichnen, abgesehen von STRATUS, wo die Entwickler einen hohen Redundanzeinsatz für eine hardwareimplementierte Fehlertoleranz nicht scheuten. Interessant ist, daß dies - obwohl oft proklamiert - sich nicht negativ auf die preisliche Konkurrenzfähig-

keit ausgewirkt hat. Dagegen war dem AURAGEN-Konzept - wenngleich elegant und redundanzsparend - offenbar wegen erheblicher Schwierigkeiten bei der Software-Entwicklung, im ursprünglichen wie auch im Nachfolgesystem TARGON32, kein Erfolg beschieden. Dem hohen Entwicklungsaufwand für eine Softwareimplementation für Fehlertoleranz steht allerdings unbestreitbar ihr Flexibilitätsvorteil entgegen. Ein Universalkonzept, das sich leicht auf unterschiedliche Systemplattformen anwenden läßt, wäre hierfür also sehr erwünscht.

All die hier aufgeführten Systeme sind nicht beliebig skalierbar[1]. Häufig findet sich ein zweistufiges Ausbaukonzept. Die Systeme sind in einzelne Cluster unterteilt, die ihrerseits weiter vernetzt werden können. Aus Fehlertoleranzgründen werden dazu redundante Busse verwendet. Die Cluster selbst sind von kleiner bis mittlerer Gößе (2-32 Knoten) und stellen als abgeschlossene Fehlerbehandlungsbereiche eine Beschränkung für die Redundanzverwaltung dar, d.h. über Clustergrenzen hinaus können ausgefallene Rechnereinheiten nicht durch andere ersetzt werden. Innerhalb des Fehlerbehandlungsbereiches gilt außerdem in der Regel eine 1-Fehler-Annahme: mehr als ein Knotenfehler kann nicht toleriert werden. Existierende fehlertolerante Systeme sind also nicht sehr flexibel, was die Wahl des Fehlertoleranzverfahrens angeht, und begrenzt in der Mächtigkeit ihres Fehlermodells.

Über diesen Ansatz hinaus versucht das ATTEMPTO-Fehlertoleranzkonzept dem Benutzer einen höheren Freiheitsgrad zu bieten. Ähnlich wie bei den zuvor beschriebenen Systemen besteht jedoch eine Größenbeschränkung, und das Fehlertoleranzverfahren ist festgelegt. Das gewählte maskierende Grundkonzept gestattet jedoch eine variable Fehlertoleranz mit relativ feiner Abstufung: ein Fehlertoleranzgrad kann vom Benutzer eingegeben werden, worauf das System eine automatische Redundanzverwaltung vornimmt und die Fehlertoleranzmechanismen transparent für den Anwender ablaufen. Mit dem Fehlertoleranzgrad ist ein variabler Replikationsgrad der Redundanz-Grundeinheiten (Benutzerjobs auf disjunkten Rechnerknoten) und damit ein variables Fehlermodell verbunden, das es nun zuläßt, auch Mehrfachfehler im System zu tolerieren. Die Grundmechanismen sind relativ einfach gehalten und in einem eigenständigen, modularen, von restlichen Betriebssystemfunktionen abgekapselten Softwaresystem realisiert, sodaß das Fehlertoleranzkonzept gut auf unterschiedliche Basis-Systeme angepaßt werden kann. Die Prototyp-Realisierung ist ein General-Purpose-Arbeitsplatzrechner für interaktive Anwendungen. Es dient zu experimentellen Zwecken und als Machbarkeitsbeispiel. Prinzipiell ist die Eignung des Konzeptes aber auch für andere Anwendungsgebiete gegeben. Denkbar wäre z.B. der Einsatz als fehlertoleranter (Platten-)Server in einem verteilten System, als fehlertolerante Diagnoseeinheit in einem Rechnernetz, und auch - vorausgesetzt das Basissystem erfüllt bestimmte Anforderungen an das Zeitverhalten (Echtzeitfähigkeit des Betriebssystems und deterministische Gleichlaufabweichungen parallel arbeitender redundanter Prozeßreplikate) - als hochzuverlässiger Steuer- oder Leitrechner im Bereich der Industrie-Automation. Für den interaktiven Einsatz ist es als System besonders hoher Zuverlässigkeit für die Dauer typischer und auch langlaufender Benutzerjobs angelegt, wie etwa Simulationsläufe, nicht jedoch für den wartungsfreien Dauerbetrieb. Hohe Systemverfügbarkeit ist also nicht das primäre Zuverlässigkeitsziel. Sie wird jedoch durch bestimmte Eigenschaften der Wartbarkeit unterstützt. In erster Linie ist dafür eine Online-Fehlerdiagnose, die unmittelbar der Fehlermaskierung zugrundeliegt, verantwortlich. Die Reparaturzeiten können darüberhinaus durch ein wissensbasiertes Werkzeug zur Offline-Hardwarefehlerdiagnose [Phil91] verkürzt werden.

1. Beliebige Skalierbarkeit im Sinne von unbegrenzter Komponentenzahl ist in physikalischen Systemen ohnehin unmöglich. In diesem Zusammenhang ist damit gemeint, daß sich Beschränkungen in der linearen Skalierbarkeit schon bei relativ kleiner Komponentenzahl einstellen, oder aber das Erweiterungskonzept selbst in Stufen variiert.

2. Redundanzkonzepte für Fehlertoleranz

Fehlerbehandlungsstrategien können auf zwei Grundmechanismen zurückgeführt werden: Fehlerunterdrückung und Fehlererholung. Für den ersten Fall wird statische Redundanz benötigt, im zweiten vor allem dynamische. Statische Wirkredundanz bindet einerseits Rechenleistung, andererseits jedoch ist für eine vorgegebene Anwendung das Leistungsverhalten i.a. besser, insbesondere im Fehlerfall, da unterbrechungsfreier Betrieb möglich ist. Dies prädestiniert solche Systeme für Echtzeitanwendungen. Bei reinrassig maskierenden Systemen beruht dieser Leistungvorteil vor allem darauf, daß die sonst üblichen Schritte der Fehlererdiagnose (Fehlererkennung und -lokalisierung) nicht explizit ausgeführt werden, sondern mit der Fehlerbehandlung selbst zusammenfallen. Es läßt sich zeigen (Kapitel 6.5), daß solche Systeme hochzuverlässig für begrenzte Einsatzdauer sind und auch hochsicher, zum einen, weil der Redundanz-Einsatz zu einem hohen synergetischen Effekt hinsichtlich der Zuverlässigkeit führt, und zum anderen, weil ihre impliziten Fehlererkennungseigenschaften, die auf Relativtests beruhen, sehr wirkungsvoll sind. Sie sind aber nicht für lange Lebensdauer und hohe Verfügbarkeit geeignet wegen der fehlenden Defektkomponentenausgrenzung und Modul-Rekonfiguration. Im Gegensatz zur Maskierung geht die Fehlererholungsstrategie sparsamer mit struktureller Redundanz um, i.d.R. jedoch auf Kosten von Zeitredundanz. Im Falle von ausschließlich betrachteten transienten Fehlern ist gar keine zusätzliche statische Wirkredundanz vonnöten. Charakteristisch für solche Systeme ist, daß Ersatzkomponenten erst nach dem Auftreten von Fehlern zur Verfügung gestellt werden müssen, um den geforderten Systemdienst beibehalten zu können. Diese Fehler müssen aber sicher erkannt und lokalisiert werden, um sie unschädlich machen zu können. Vor dem Fehlerfall können die Ersatz-Module zur gesteigerten Gesamtleistung beitragen. Bei Ausfall von Modulen degradiert das System dann nicht nur im Zuverlässigkeits- sondern auch im Leistungsverhalten. Solche Eigenschaften stören nicht sehr in Mehrbenutzer-Systemen für interaktive Anwendungen, z.B. in Verteilten Systemen (Rechnernetze mit kooperativem Verteilungscharakter oder Vermittlungssysteme mit verteilter Rechnersteuerung, usw.). Kann die Leistungsdegradierung nicht hingenommen werden, muß statische Wirkredundanz eingesetzt werden in Form von nicht fremdgenutzten Reservekomponenten. Mit zunehmendem Anteil an statischer Redundanz führt der Weg zu hybriden Systemen.

Ist hohe Verfügbarkeit das vorherrschende Entwicklungsziel, spielt die Reparierbarkeit des Gesamtsystems eine wesentliche Rolle neben der allgemeinen Zuverlässigkeit. Für lange Lebensdauer sind dann viele Ersatzkomponenten (Spare-Redundanz) gefragt oder die Reparatur der defekten Komponenten und deren Reintegrierbarkeit während des Betriebs (manuelle On-Line-Reparierbarkeit). Für sehr hohe Verfügbarkeit sind offenbar fehlererholende Systeme zu favorisieren, da deren Kernstück zur Fehlerbehebung, die Rekonfiguration, schon eine (automatische) Reparatur des Systems ist. Ihr Problem liegt jedoch in der Zuverlässigkeit der Fehlererkennung. Ein Maß dafür ist die Fehlerüberdeckung, die die idealen 100% nennenswert unterschreiten kann und damit die erreichbare Maximalzuverlässigkeit maßgeblich verschlechtert. Sie richtet sich nach dem Aufwand an Prüfredundanz oder auch zusätzlicher Wirkredundanz, z.B. in Form von Verdoppelung. Letztere wird zu Vergleichstests herangezogen und wirkt damit im Grunde ähnlich statisch wie ein maskierendes System. Dies treibt den Aufwand erheblich in die Höhe, sodaß dieser Vorteil gegenüber maskierenden Systemen mit Mehrheitsentscheid verschwinden, ja sich sogar ins Gegenteil verkehren kann, wie die Quadruplex-Technik zeigt, die auch die Ersatzkomponente doppelt auslegt (sogennante *Pair-and-Spare*-Methode [Glaz84]). Dann liegt es vielleicht oft näher, ein fehlermaskierendes Grundverfahren zu verwenden, und dieses um zusätzliche Diagnose- und Rekonfigurationsmöglichkeiten zu

erweitern (klassische Hybridredundanz).

Auf alle Fälle ist zu erwarten, daß quasi-ideale Zuverlässigkeitsanforderungen, d.h. hohe Zuverlässigkeit im langen Einsatz bei stoßfreiem Betrieb im Fehlerfall, zu einer Nivellierung der Architekturmerkmale führen. Dann nämlich sind diese Systeme hybrid und mit einem hohen Anteil an statischer Redundanz versehen sowohl zur Fehlererkennung als auch für Ersatzkomponenten und automatisch reparierbar im Sinne von rekonfigurierbar. Soll aktive Spare-Redundanz gespart werden, kommt - für viele Anwendungen - noch manuelle On-line-Reparierbarbarkeit (durch Ersatz von Defektkomponenten) in Betracht. Abb. 3 illustriert grob diese Tendenz in der Vereinheitlichung des Redundanzcharakters. Hierbei ist hohe Zuverlässigkeit (im Sinne von Überlebenswahrscheinlichkeit) als typisch für Fehlermaskierung mithilfe von statischer Redundanz etwas vereinfacht als orthogonale Eigenschaft zu Verfügbarkeit und langer Lebensdauer betrachtet, um den Grundcharakter zu kennzeichnen.

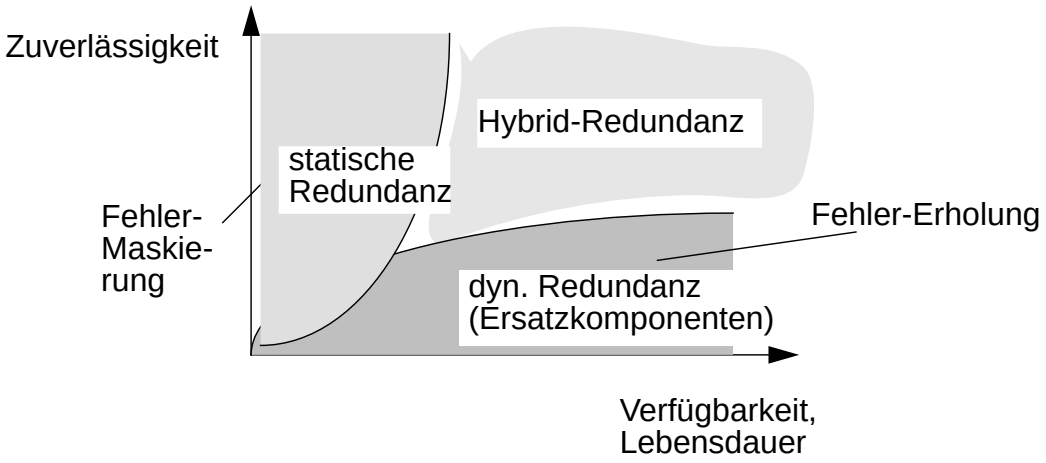


Abb. 3 Angleich des Redundanzcharakters bei Extremanforderungen

2.1 Fehlermaskierung als ausbaufähiger Basismechanismus

Die Forderung nach Skalierbarkeit eines Rechnersystems läßt sich leichter erfüllen, wenn allen Ausbaustufen ein einheitlicher Grundmechanismus zur Fehlertoleranz zugrundeliegt.

Fehlermaskierung hat gegenüber Fehlererholung folgende Vorteile:

- Bei fehlertolerantem Einsatz im Kurzzeitbereich ist ein fehlermaskierendes System zuverlässiger.
- Wie bereits angedeutet, ist die Fehler-Behandlung sehr schnell (in taktsynchronen Systemen unmittelbar), womit die Eignung für Echtzeitsysteme gegeben ist.
- Die Fehler werden nach außen, d.h. an der Schnittstelle zur Systemumwelt, nicht sichtbar, was hochsichere Systeme ermöglicht. Diese Eigenschaft der *Integrität* kann auch ausgenutzt werden, um Systemkomponenten mit *fail- stop*-Verhalten zu bilden, die als Grundelemente für andere Fehlertoleranzverfahren dienen. Dieses Fehlermodell charakterisiert das Ausfallverhalten als fehlerisolierend und vereinfacht den Entwurf und die Verifizierung fehlererholender Systeme, in denen nur solche fail-stop-Komponenten verwendet werden.

- Erweiterungskonzepte hin zu gesteigerter Zuverlässigkeit (Skalierbarkeit) gestalten sich einfacher, auch hinsichtlich des zugrundeliegenden Fehlermodells.
- Daß keine sofortige Fehlkomponentenausgrenzung vorgenommen wird, ist in einem Fehlermodell von Vorteil, in dem transiente Fehler als sehr viel häufiger als permanente angesehen werden. Temporär fehlerhafte Komponenten erhalten so die Möglichkeit zur "Selbstheilung". Diese Chance kann eingeräumt werden, da gleichzeitig auch die Integrität des Systems gewährleistet bleibt. Tatsächlich sind transiente Fehler in der Realität durchschnittlich etwa zehnmal häufiger als permanente [Siew82]; ein solches Fehlermodell erscheint also gerechtfertigt.
- Die Anwendertransparenz der Fehlertoleranzmaßnahmen ist zugleich mit hoher Fehlerüberdeckung leicht gegeben, da die Fehlererkennung auf Relativtests beruht. Fehlererholende Systeme müssen sich dagegen auf Absoluttests abstützen, die bei allgemeinem Fehlermodell zur Tolerierung beliebigen Fehlverhaltens ohne Berücksichtigung der Semantik des jeweiligen Anwenderprogramms durch spezifische Akzeptanzkriterien nur geringere Fehlerüberdeckung ermöglichen. Bei Fehlermaskierung muß allerdings eine Beschränkung der Fehleranzahl und die Isolierung der Fehler in einzelnen unabhängigen Fehlerbereichen im Modell garantiert sein (*Fehlerbereichsannahme* [Echt90]).
- Wegen der einfachen Fehlererkennung und der - bei "reinrassiger" Maskierung - zunächst nicht erforderlichen Fehlerlokalisation und -behebung gestaltet sich die Implementierung der Fehlertoleranz-Mechanismen relativ einfach.

Eine Mehrzweck-Fehlertoleranz kann daher sinnvollerweise auf einem maskierenden Grundverfahren aufsetzen, es müssen jedoch gleichzeitig auch Mechanismen zur Fehlererholung bereitgestellt werden, um den Forderungen nach hoher Verfügbarkeit und längeren Einsatzzeiträumen gerecht zu werden.

2.2 Fehlertoleranz-Design: Vom verteilten System zur Fehlertoleranz

Beim Design eines fehlertoleranten Systems kann man zwei hauptsächliche Vorgehensweisen unterscheiden: zum einen wird die Architektur des Systems den speziellen Zuverlässigkeitserfordernissen der jeweiligen Anwendung angepaßt und so eine Optimierung des Redundanzeinsatzes und möglicherweise auch des System-Leistungsverhaltens erreicht. Zum anderen versieht man existierende Syteme bzw. solche mit allgemeinerer Architektur zusätzlich mit Fehlertoleranzeigenschaften. Während der erste Ansatz dem Sytemdesigner den höchsten Freiheitsgrad läßt, die Systemnutzbarkeit aber auf die jeweilige Anwendung bzw. Anwendungsklasse beschränkt bleibt, steht beim zweiten die Erweiterung des Systemdienstangebotes für eine flexiblere Systemnutzbarkeit im Vordergrund, bei jedoch reduzierter Flexibilität für den Designer. Für den Allgemeinansatz sind bestimmte Systemarchitekturen besonders gut geeignet als Basis für ein fehlertolerantes System. Dazu gehören alle modularen Systeme, die homogene Grundstruktur besitzen, d.h. aus mehreren Wirkkomponenten gleichen Typs bestehen. Damit ist inhärent die für Fehlertoleranz unentbehrliche Redundanz gegeben. Die Grundeinheiten müssen in geeigneter Weise miteinander verbunden sein, um ein Zusammenwirken für Fehlertoleranzzwecke zu ermöglichen. Für Rechensysteme sind insbesondere Parallelarchitekturen vom Typ MIMD hervorzuheben, deren Komponenten a priori einen höheren Grad an Autonomie besitzen als z.B. bei SIMD-Architekturen. Autonomie fördert die Robustheit, weil es damit möglich wird, zuverlässigkeitskritische singuläre Funktionseinheiten zu vermeiden. Innerhalb der Klasse MIMD sind daher auch Mehrrechnerstrukturen zentralspeicher-gekoppelten Multiprozessoren vorzuziehen. Fehlertoleranzmechanismen sind viel einfacher zu implementieren (und auch einfacher zu verifizieren), wenn Fehlerrückwirkungen ausgeschlossen

werden können. Auch das spricht für autonome, lose gekoppelte Mehrrechner, deren physikalische Schnittstellen auf das nötigste - das ist i.d.R. das für die Kommunikation Erforderliche - begrenzt sein sollten. So finden sich auch relativ zahlreiche fehlertolerante Systemrealisierungen oder Realisierungskonzepte, die auf verteilten Systemen aufbauen, z.B. Stratus [Frei82], [TW89], Tandem [Bart81],[HM84], Nixdorf [Borg89], SIFT [Wen78], BBC-Alphorn [AK88], REBUS [ACD82], Fehlertolerantes MACH [Bab90]. Allen gemeinsam ist ihre Ausbaufähigkeit für Fehlertoleranz im geplanten Anwendungsbereich.

2.3 Designbeispiele fehlertoleranter Systeme mit Mehrzweck-Eigenschaften

Ein klassisches Beispiel für ein konfigurierbares Multiprozessor-System mit allerdings statischem Redundanzkonzept und dediziertem Anwendungsbereich ist das FTMP-System [HSL78], bei dem sich aus Grundeinheiten (Speicher-, Prozessor-, Busmodule) sogenannte Triads für eine hardwareimplementierte Fehlermaskierung durch einen 2-von3-Mehrheitsentscheid zusammenfügen lassen. Direkt vergleichbar im Anwendungsbereich und Grundprinzip, jedoch flexibler mit seiner allgemeinen Hardware-Architektur ist SIFT [Wen78], ein Pionierkonzept für eine Softwarerealisierung von Fehlermaskierung, das auch im Prozeß-Steuerungs-System AUGUST 300 eine kommerzielle Verbreitung gefunden hat [Wen81]. Ein entscheidender Schritt hin zu einer Allzweck-Fehlertoleranz ist die variable Redundanznutzung mit verschiedenen Taskreplikationsgraden (theoretisch auch über Verdreifachung hinaus), die jedoch statisch konfiguriert werden muß. Hauptsächlich im Bereich der industriellen Prozeßautomatisierung finden sich gelegentlich softwareimplementierte fehlertolerante Systeme, um bestehende und bewährte nicht-fehlertolerante Grundhardware für erhöhte Zuverlässigkeitsansprüche verwenden zu können [SIM90],[MPR87]. Häufig sind die Grundkomponenten ganze Steuerrechner, und mit geringem zusätzlichen Hardwareaufwand verkoppelt.

In einer solchen relativ allgemeinen verteilten Systemumgebung sind auf disjunkten Rechnern replizierte Tasks die eigentlichen Verwaltungseinheiten der Fehlertoleranz-Betriebssoftware. Zuverlässigkeitsklassen können dann zweckmäßigerweise als Attribut bei der Taskgenerierung unterschiedliche Fehlertoleranzverfahren kennzeichnen (sogenannte *taskspezifische Fehlertoleranz* [Fär81]). Dies führt zu einer sehr effektiven Redundanznutzung allerdings bei einer recht groben Abstufung der Zuverlässigkeitseigenschaften und fixiertem Fehlermodell. Zudem bezieht sich die Fehlertoleranz zunächst nur auf die Steuerung, was in der Regel nicht ausreicht, um ein projektiertes Zuverlässigkeitsziel für das gesamte technische System zu erreichen. Im Prozeßautomatisierungsbereich z.B. muß die Prozeßstrecke mit einbezogen werden, da sich hier die dominant zuverlässigkeitskritischen Komponenten befinden (Aktoren und Sensoren sind meist mechanisch und arbeiten direkt in der “ungesunden” Produktionsumwelt). Hierzu kann man *Fehlertoleranzklassen nach funktionalen Gesichtspunkten* definieren, wie etwa in [Män86].

Generell ist es wegen der Integrität des Ausgabeverhaltens und für eine hohe Fehlerüberdekkung von Bedeutung, möglichst weit außen liegende System-Schnittstellen als Referenzort für Fehlererkennungs- bzw. -behandlungsmaßnahmen zu wählen. Dies spricht für eine sehr grobe Granularität der redundanten Komponenten. Andererseits läßt sich bei feinerer Granularität Aufwand sparen. In der Regel benötigt man aber detaillierte Kenntnis des Systems, um diese punktuelle Redundanz an den zuverlässigkeitskritischen Stellen plazieren zu können, sodaß automatische Mechanismen ausscheiden. Diese Methode eignet sich besser für Verfahren der expliziten software-implementierten Fehlertoleranz. Besonders gewinnbringend lassen sich sol-

che Verfahren in Parallel-Programm-Entwicklungsumgebungen für MIMD-Rechnerstrukturen integrieren, insbesondere dann, wenn die (logische) Granularität der Parallelität nicht zu fein und der der Fehlertoleranz angepaßt ist [Trib92], [Kram91], [CGR88]. In Software-Architekturen , die dem Client-Server-Prinzip für verteilte Systeme und/oder dem Objekt-Orientierungs-Paradigma folgen, sind die Grundeinheiten i.a. mittelkörnig, d.h. es handelt sich um Prozedur- bzw. Modul-Instanzen, von denen gleichzeitig mehrere auf zumeist makroskopische Hardwarekomponenten, z.B. Rechnerknoten in einem Netz, verteilt sind. Natürliche Systemschnittstellen zum Ansatz für Fehlertoleranzmaßnahmen liegen dann bei der Dienstbeauftragung bzw. Objektinteraktion.

Auf der Basis von Client/Server-Systemen für allgemeine Anwendungen gibt es auch Ansätze für eine implizite Fehlertoleranz: ft-Mach ist ein solches Beispiel [Bab90], ein Konzept für Fehlertoleranz im verteilten System Mach. Das LOCUS-Konzept bietet nahezu transparent Fehlertoleranz für ein verteiltes Dateisystem in einem Netz von UNIX-Rechnern [Popek81].

In massiv-parallelen, regulären Rechnerstrukturen erweist sich ein Mehrzweck-Fehlertoleranzkonzept als problematisch. Solche Hochleistungs-Parallel-Rechner sind allerdings auch nicht für allgemeine Anwendungen konzipiert. Dennoch wäre auch hier eine transparente Fehlertoleranz von Vorteil, um die i.a. ohnehin komplexe Programmierung nicht noch weiter zu erschweren. In der Regel scheut man jedoch den dazu erforderlichen hohen Redundanzeinsatz, und beschränkt sich darauf, Redundanz gezielt an dafür geeigneten Stellen der Grundstruktur einzusetzen. Solche Stellen in regulären Strukturen zu finden, ist jedoch meist nicht einfach. Vorzuziehen sind hier hierarchisch aufgebaute Strukturen, wie z.B. Mehrebenenfelder mit Pyramidengrundstruktur, wie sie im Erlanger Projekt MEMSY [DC89] verwendet werden.

3. Probleme der Fehlermaskierung

[1]Wenn man sehr eingeschränkte Fehlermodelle außer Acht läßt, z.B. solche, in denen man nur selbstheilende transiente Fehler annimmt, liegt der Fehlermaskierung immer statische strukturelle Redundanz zugrunde: ein- und dieselbe Aufgabe wird nebenläufig auf (mindestens drei) gleichen Komponentenexemplaren ausgeführt, die Ergebnisse eingesammelt, und einem Votierungsvorgang unterworfen. Zum Votieren gehört die Fehlererkennung, die auf Relativtests beruht: weichen zwei Ergebnisse voneinander ab, liegt ein Fehler vor. Das richtige Ergebnis wird durch einen ***Maskierungsentscheid*** [Echt90] ermittelt, der die Strategie des Votierens kennzeichnet. Bei deterministischem Verhalten der Komponentenexemplare kommt z.B. eine Einstimmigkeits-, Mehrheits-, oder Paarentscheidung in Betracht. Maskierung über nichtdeterministische Daten ist ebenfalls möglich, allerdings nur unter gewissen Randbedingungen. Die Votierungsstrategie kann sich dann nicht mehr auf reinen Datenvergleich stützen, sondern muß bestimmte Ähnlichkeitskriterien abprüfen, denen die korrekten Daten auch genügen müssen. Mögliche Maskierungsentscheidungen sind dann die Medianentscheidung oder Intervallentscheidung [DaWa78], [Echt90]. Den Maskierungsentscheid nimmt der Maskierer vor. Er kann auch verteilt realisiert sein, um keinen zuverlässigkeitskritischen Punkt im System darzustellen. In diesem Fall sind die Maskierer ebenfalls repliziert, und müssen alle die gleiche Ergebnismenge aufsammeln, um eine lokale Maskierungsentscheidung treffen zu können. Die redundanten Maskierer gehören in der Regel zum ***Maskierungsbereich***, der die replizierten Komponenten in isolierten ***Einzelfehlerbereichen*** enthält. Das Fehlermodell schreibt als besondere Eigenschaft bei Fehlermaskierung die Begrenzung der Maximalzahl der tolerierbaren Fehler vor (sogenannte *k-Fehler-Annahme* mit k Anzahl der zugelassenen Fehler), und daß diese *unabhängig* voneinander sind, in dem Sinne, daß Fehler nicht zum gleichen falschen Ausgabeverhalten bei verschiedenen Komponenten führen, sodaß die Fehlererkennung durch Relativtests versagt. Im folgenden wird - für den Fall, daß Zweifel bestehen - die Konfiguration mit redundanten Maskierern vorausgesetzt (in [Echt90] auch *m-fach-n-von-m-Maskierungssystem* genannt, mit m=n+k Replikationsgrad, das ist die Gesamtzahl der redundanten Komponenten), die einen Maskierungsentscheid aufgrund der Annahme eines deterministischen Verhaltens der replizierten Komponenten treffen (im weiteren gelegentlich als “deterministische Fehlermaskierung” vereinfacht) . Die redundanten Grundkomponenten sind Prozesse, die sich zusammen mit den redundanten Maskierern auf getrennten Hardwarekomponenten befinden, den Rechnereinheiten oder Rechnerknoten, die so die Enzelfehlerbereiche physikalisch abgrenzen.

Bei Fehlermaskierung gibt es einige grundsätzliche Probleme, die z.T. schon angesprochen wurden und im folgenden näher untersucht werden sollen. Zunächst muß für eine zeitliche Zusammenführung der Ausgabewerte der einzelnen Replikate gesorgt werden, damit ein sinnvoller Maskierungsentscheid überhaut möglich wird. Der Gleichlauf im Programmfluß der Prozeßexemplare kann darüberhinaus durch spezifische Ungleichheiten in der Umwelt der replizierten Prozesse, z.B. durch eine indetermistische Erfassung von Eingabedaten, gefährdet sein.

1. Das im folgenden kurz vorgestellte Prinzip der Fehlermaskierung wird unter Zuhilfenahme einer Terminologie von [Echt90] erläutert. Das Buch [Echt90] kann als Standardwerk über Fehlertoleranzverfahren dienen, das eine deutsche Fehlertoleranz-Begriffswelt treffend aus dem originär englischsprachigen Umfeld der Fachliteratur übersetzt. Gleichzeitig schafft es eine ausgezeichnete Basis für den relevanten Vergleich unterschiedlicher Fehlertoleranzverfahren.

3.1 Synchronisation

Eine Fehlermaskierung beruht letztendlich auf dem Vergleich von Ausgabewerten der einzelnen replizierten Komponenten. Wieviel Werte gleichzeitig für den Maskierungsentscheid herangezogen werden müssen, hängt von der gewählten Votierungsstrategie ab. Der eigentlichen Maskierungsentscheidung geht eine Phase des Aufsammelns der zu vergleichenden Ausgabewerte voraus. Im allgemeinen wird ein System eine Folge von Ausgabewerten produzieren; damit verbunden ist eine Folge von Votierungsvorgängen. Aufeinanderfolgende Ausgabewerte einer Komponente müssen voneinander unterscheidbar sein, um sie zusammen mit denen der anderen Replikate für den Vergleich sinnvoll gruppieren zu können. Dies kann entweder dadurch erreicht werden, daß die Produktion von Ausgabewerten in synchronen Runden erfolgt, innerhalb derer nur jeweils ein Ausgabewert erzeugt werden kann, oder aber indem bei unkoordiniertem Gleichlauf die Ausgabewerte eine Kennzeichnung tragen, die es ermöglicht, zusammengehörige Werte verschiedener Komponentenexemplare zu identifizieren.

Maskierungssysteme lassen sich nach Grad der Synchronität klassifizieren [DaWa78],[Wen78] in

- ***Taktsynchrone Systeme*** (starre Kopplung: FTMP, STRATUS): Die Komponentenexemplare werden von einem zentralen Grundtakt (z.B. Prozessortakt) versorgt. Der Gleichlauf ist damit schon auf unterster Ebene, und damit auch für dem Maskierungsentscheid, sichergestellt. Dieses Verfahren schafft die Grundlage für hardwareimplementierte Fehlermaskierung. Nachteilig ist, neben der geringen Flexibilität des Fehlertoleranzverfahrens, daß i.d.R. besondere Maßnahmen zur Erhöhung der Zuverlässigkeit des zentralen Taktgebers erforderlich sind, z.B. durch Anwendung dafür geeigneter gesonderter Fehlertoleranzverfahren.
- ***Rahmensynchrone Systeme*** (SIFT, SAFE [YST85], FUTURE [DeRi82]): Zeitscheiben, gebildet durch die Zusammenfassung von Haupttaktzyklen oder durch periodische Ereignisfolgen, schaffen einen Grundrahmen für eine globale Synchronisation gröberer Körnung als bei Taksynchronität, innerhalb dessen die einzelnen Replikate unabhängig voneinander arbeiten. Neben einer Phase, in der die Anwendung rechnet, gibt es noch eine Phase im Grundrahmen, in der eine globale Abstimmung der lokalen Betriebssysteme, u.a. auch für die Fehlermaskierung, erfolgt. Dieses Rahmensynchronisationsverfahren läßt sich durch Software implementieren. Die geforderte Grundsynchronität spiegelt sich jedoch in der Anwendung wider, falls periodische Ereignisse den Rahmen setzen, und ist damit nicht voll transparent. Zur zusätzlichen Erläuterung kann ein Beispiel in Kapitel 3.2.3 dienen.
- ***Asynchrone Systeme*** (Ereignissynchronisierung, geregelte Synchronisation (Puffersynchronisation) [Echt90], [Gunn83], [DaWa78], ATTEMPTO): Die Komponentenexemplare arbeiten zunächst frei und autonom, und synchronisieren sich nur zu besonderen Anlässen. Dieses Verfahren ist besonders gut geeignet in einem System redundanter Prozesse, die sich mithilfe von Nachrichtenaustausch synchronisieren. Die Anlässe zur Synchronisation sind durch bestimmte Ereignisse im Programmfluß der Prozeßreplikate vorgegeben, z.B. durch das Schreiben von Daten auf eine den Replikaten gemeinsame Schnittstelle (globale Ressource), beim Verschicken von Interprozeß-Nachrichten, o.ä.. Bei den asynchronen Synchronisationssystemen kann noch nach dem Grad des Auseinanderlaufens der Ausgabefolgen der Replikate differenziert werden. Im Maskierer werden die eintreffenden Ausgabewerte in den einzelnen Prozeßexemplaren zugeordneten

Puffern solange zwischengespeichert, bis eine genügende Anzahl zusammengehörender Werte von unterschiedlichen Exemplaren vorliegt. Mit steigender Puffertiefe kann ein größerer Vorlauf einzelner Prozeßreplikate vor den anderen erlaubt werden. Die Kopplung zwischen den Replikaten wird damit zunehmend loser und die Nebenläufigkeit steigt, was in Summe zu einer besseren Auslastung der Rechenkapazität im System führen kann. Nicht immer sind jedoch die Belastungsverhältnisse soweit kontrollierbar, daß gewährleistet werden kann, daß der Maximalfüllstand der Puffer nicht überschritten wird; in solchen Fällen ist ein Mechanismus zur Fluß-Steuerung notwendig, um die Ankunftsrate der Ausgabewerte zu drosseln. Bei starker Belastung geht dann der Vorteil der Entkopplung verloren. Ein weiteres Problem ist in diesem Zusammenhang von Bedeutung. Um auch dem Nichteintreffen von Ergebniswerten Rechnung zu tragen, z.B. bei Ausfall einer replizierten Komponente, kann das Aufsammeln zusammengehörender Ergebniswerte durch eine Zeitschranke begrenzt werden. Um eine solche Zeitschranke überhaupt definieren zu können, muß gesichert sein, daß die Gleichlaufdifferenz im System einen Maximalwert nicht überschreitet. Um auf eine globale Systemzeit verzichten zu können - dies ist vor allem relevant bei verteilten Maskierern - wird man relative Zeitmessung vorziehen, d.h. eine Zeitüberwachung wird lokal gestartet, wenn z.B. der erste (oder die Mehrheit) von den vergleichbaren Ergebniswerten vorliegt[1]. Bei Puffersynchronisierung tritt nun das Problem der ***Akkumulation von Zeitschranken*** auf (*Zeitgrenzenproblem* [Echt90]): beim Einordnen des ersten Ausgabewertes auf den k-ten Pufferplatz kann die Wartezeit bis zum Eintreffen des letzten auf das k-fache des Wertes der Zeitschranke im ungepufferten Fall ansteigen. Die Zeitüberwachung wird also mit zunehmendem Füllstand immer ineffizienter, was vor allem bei Echzeitanforderungen stört, und allgemein die Zeitüberwachungsfunktionen verkompliziert.

Neben der hier vorgestellten Klassifizierung von Synchronisationsverfahren kann eine weitere in Abhängigkeit vom Synchronisationsmechanismus vorgenommen werden. [Echt90] unterscheidet 1. gesteuerte, 2. geregelte und 3. implizite Synchronisation. Die 1. Klasse faßt solche mit festem global vorgegebenen Zeitschema zusammen, wie z.B. die o.a. Takt- und Zeitscheibensynchronisationsverfahren, die 2. kennzeichnet Rückkopplungsmechanismen wie die Ereignissynchronisation, und die 3. Klasse charakterisiert Sonderfälle, in denen die Beachtung zusätzlicher Randbedingungen und Systemvorgaben aus rein lokaler Sicht die Synchronität ohne globale Koordination impliziert. Ist z.B. die Produktionsrate von Vergleichswerten naturgemäß viel geringer als die Bedienrate des Maskierers, besteht nie die Gefahr des Überschneidens von Synchronisationsrunden. Ein implizites Synchronisationsverfahren gemäß diesem Modell kommt im ATTEMPTO-Broadcast-Kommunikationssystem zum Einsatz (Kapitel 5.4.6).

3.2 Gleichlaufgefährdung durch Indeterminismus

Grundlage für jede Fehlerbehandlung, die mit zeitlich wiederholten oder zeitgleich agierenden Prozeßkopien arbeitet, und auch für jede Fehlererkennung, die auf Relativtests beruht, ist der deterministische Programmlauf der intakten Prozeßexemplare. Determinismus bezogen auf ein einzelnes Prozeßexemplar bedeutet, daß eine gegebene Eingabesequenz bei wiederholter Programmausführung immer die gleiche Ausgabesequenz erzeugt. Wird nur statische Redundanz

1. Bei Einstimmigkeitsentscheid würde man z.B. so vorgehen. Häufig möchte man auch nicht nur maskieren, sondern außerdem, und möglichst unmittelbar, diagnostizieren. In einem solchem Fall ist man z.B. auch bei Mehrheitsentscheid an allen, und nicht nur an einer Mehrheit von gleichen Ergebniswerten interessiert.

eingesetzt, d.h. arbeiten die Kopien zeitgleich, wie z.B. bei der reinrassigen Fehlermaskierung, ist Determinismus dann gegeben, wenn eine Eingabesequenz aus der Systemumwelt, die allen Replikaten identisch zugeleitet wird, bei diesen zu identischen Ausgabesequenzen *beim aktuellen Programmlauf* führt (im fehlerfreien Fall). Die Eingaben aus der Umwelt sind z.B. je nach Anwendung Eingabedaten zur Ist-Zustandsübermittlung des technischen Prozesses, die Eingabe des mit dem Programm interagierenden Benutzers, o.ä.. Probleme sind immer dann zu erwarten, wenn zur Dezentralisierung replizierte Eingabesensoren vorhanden sind, die den singulären Eingangsdatenstrom zu den einzelnen Prozeßreplikaten gesondert führen. Abweichungen können sich ergeben bei der Wertermittlung oder der zeitlichen Erfassung: z.B. bei der Analog/Digital-Wandlung aufgrund naturgegebener physikalischer Streuungen in Übergangsbereichen zwischen Diskretisierungsstufen oder nicht exakt synchronisierten (bzw. synchronisierbaren) Abtastzeitpunkten. Das Problem, die redundanten Prozeßexemplare mit identischen Eingabedaten zu versorgen, um einen determisistischen Programmlauf zu gewährleisten, wird gelegentlich als ***Problem der Quellkongruen***z (*source congruency* [BS84]) bezeichnet.

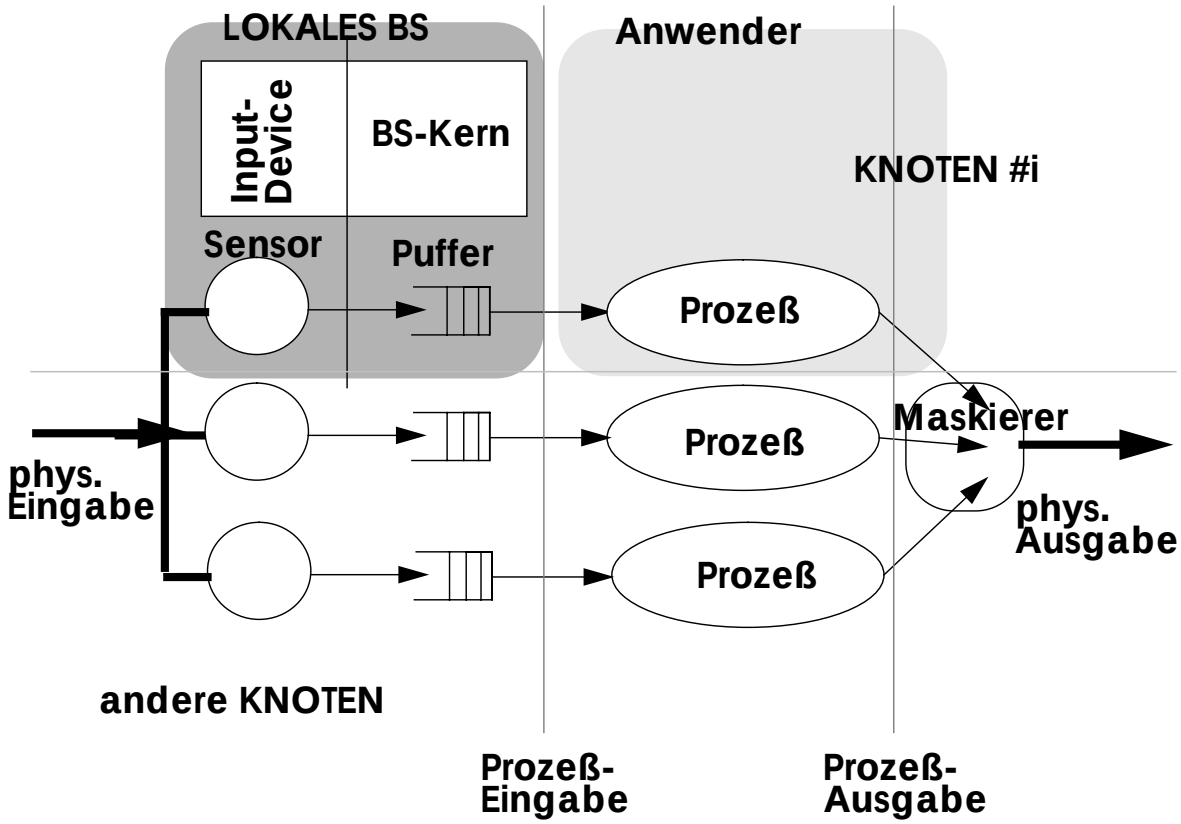


Abb. 4 Zum deterministischen Verhalten bei makroskopischen Replikaten in einem fehlermaskierenden System

Bei einer “Über-Alles”-Fehlermaskierung (entsprechend einer *End-to-End*-Strategie) auf Basis lose synchroniserter Prozessorknoten sind die Eingabesensoren den Benutzerprozeß-Replikaten zugeordnet und bilden zusammen mit den lokalen Betriebssystemfunktionen die makroskopischen replizierten Grundeinheiten, deren Synchronlauf garantiert werden muß (siehe Abb. 4). Dieses Modell soll als Grundlage für die weiteren Betrachtungen dienen.

Der ereignisbezogene Gleichlauf im Programmfluß der Anwenderprozesse auf den einzelnen

Replikaten ist gefährdet durch die Abhängigkeit von lokalen, global nicht synchronisierten (Betriebs-) Systemfunktionen. Unterschiede ergeben sich bei

- **synchronen Systemaufrufen**, die Informationsdienste zur Verfügung stellen, die aber nur lokale Gültigkeit besitzen, z.B. Zeitabfragen, Verlangen von prozeßbezogenen oder speicherverwaltungsbezogenen Daten.
- **asynchronerEin-/Ausgabe** von Nutzdaten von/zur Prozeß- bzw. Systemumwelt. Lokale Abweichungen ergeben sich hier hinsichtlich des Inhalts (in Bezug auf Eingabedaten von globalen Systemschnittstellen sogenannte *Replikationsfehler* [Echt90]) und des Umfanges der transferierten Daten (read bzw. write count). Neben dem bereits oben angeführten Problem der Prozeßeingabeverteilung in lokalen asynchron arbeitenden Systeminstanzen besteht auch ein Problem bei der Ausgabe von Daten über betriebssystemeigene Puffer, deren Verfügbarkeit von lokalen laufzeitbestimmten Randbedingungen abhängt. Probleme der asynchronen Ein-/Ausgabe treten vor allem bei nichtblockierendem Betrieb ("*non-blocking I/O*") auf. Nichtblockierendes Lesen ermöglicht das Pollen auf Eingabekanälen, nichtblockierende Ausgabe verhindert Warten auf freie Puffer.
- **Hardware-Interrupts** als von Natur aus asynchron den Programmfluß unterbrechende Ereignisse. In der Regel können diese jedoch nicht direkt den Programmfluß des Anwenderprozesses beeinflussen. Dies trifft weniger auf Echtzeitsysteme als auf (z.B. auf UNIX-basierte) Allzwecksysteme zu. Es gibt allerdings auch hier solche, die sogenannte *Asynchrone System-Traps* [Egan88] zur Verfügung stellen. Häufiger wird der Umweg über
- **Signale** gehen, die letztlich ebenfalls asynchrone Unterbrechungen bedeuten. Diese können z.B. auch aus Eingabedaten abgeleitet sein. Sie stellen auch eine Methode zur Interprozeßkommunikation dar. Neben einer vordefinierten Grundfunktion, z.B. Prozeßabbruch, sind der Anwendung auch eigene Maßnahmen zur expliziten Beeinflussung des Programmflusses erlaubt.

Während bei den zuerst aufgeführten synchronen Ereignissen nur ein Datenabgleich zwischen den Prozeßreplikaten erforderlich ist, muß bei asynchronen Ereignissen für eine zustandskonsistente Einwirkung gesorgt werden. Im Falle der asynchronen E/A sind - wie bei synchronen Systemaufrufen auch - natürliche Synchronisationspunkte bei der Rückkehr der Prozeßexemplare aus dem Kernmodus in den Anwendermodus gegeben, bei asynchronen Unterbrechungen hingegen muß für eine Angleichung der Prozeßzustände erst künstlich gesorgt werden. Dieses Problem wird in der Literatur auch als *Anhalteproblem* [Echt90] bezeichnet. Es ist um so schwieriger zu lösen, je loser die Replikate synchronisiert sind.

Im folgenden werden einige Lösungsansätze für die einzelnen Teilprobleme vorgestellt.

3.2.1 Synchrone Systemaufrufe

Erstreckt sich die Autonomie der Systemkomponenten auch auf lokale Betriebssystemfunktionen, etwa in der Art, daß die Prozeß- und Speicherverwaltung auf einem Rechnerknoten ausschließlich im lokalen Zuständigkeitsbereich liegen, können alle Anwendungen, die in irgendeiner Weise Bezug auf lokale System-Zustandsdaten nehmen, zu abweichendem Programmfluß bzw. Ergebnisverhalten der Prozeßreplikate führen. Betroffen sind also Systemaufrufe, die System-Informationen vom Betriebssystem anfordern. Ist vollständige Kompatibilität der Betriebssystemschnittstelle des fehlertoleranten Systems zu der des lokalen (Standard-) Betriebs-

systems als Entwicklungsziel gefordert, kann der Aufwand zur Globalisierung solcher Systemdaten beträchtlichen Aufwand bedeuten. Dazu gehören Objektbezeichner, wie z.B. die Prozeß-ID oder solche, die das Dateisystem betreffen (Dateideskriptoren und -namen). Aus Aufwandsgründen kann es vorteilhaft sein, diese Bezeichner nicht schon bei der Vergabe zu vereinheitlichen, sondern erst bei der Anforderung durch den fehlertolerant ablaufenden Anwendungsprozeß solche lokalen Daten auf globale abzubilden (Mapping). In jedem Fall ist die Funktion eines globalen Namensdienstes zu realisieren, vorteilhafterweise natürlich verteilt.

Aus der Abhängigkeit vom Ausführungsort resultierender Indeterminismus ist auch bei der wichtigen Systemfunktion Zeiterfassung zu erwarten, wenn eine globale Systemzeit fehlt. Selbst bei deren Vorhandensein würden unsynchronisierte Abfragezeitpunkte zu differierenden Ergebnissen der lokalen Systemaufrufe führen. Deshalb ist also sowohl die Zeit (der Rückgabewert des *time*- Systemaufrufs) wie auch der Programmlauf der Replikate zu synchronisieren[1]. Da in der Regel die Zeit nur sehr grobkörnig gemessen wird, sind aber die Anforderungen an den Prozeßgleichlauf nicht besonders hoch. Für den Abgleich der Zeitangabe ist jedoch ein Übereinstimmungsprotokoll erforderlich. Die Synchronisation verteilter Uhren ist ein allgemeines Problem der interaktiven Konsistenz in verteilten Systemen. Ein Überblick über die hierzu geleistete Arbeit gibt z.B. [Sim90]. Grundlage ist i.a. die implizite Annahme von Schranken für mögliche Laufzeitdifferenzen und Nachrichtentransferdauern. Kommt es weniger auf absolute Zeitgenauigkeit an, reicht es für die Forderung nach deterministischem Programmlauf für die Maskierung, eine Einigung über eine wahrscheinliche gemeinsame Systemzeit, z.B. durch Medianbildung über der geordneten Menge der lokalen Systemzeiten, zu erzielen. Diese Einigung ist nicht zyklisch notwendig, sondern nur bei der aktuellen Abfrage erforderlich.

3.2.2 Asynchrone Ein-/Ausgabe

Die Kommunikation mit der Außenwelt (bezogen auf den lokalen Rechnerkern) vollzieht sich über Geräteschnittstellen, deren Steuerung Aufgabe des Betriebssystems ist. Sehr häufig geschieht die physikalische Ein-/Ausgabe asynchron zu den die Daten produzierenden bzw. konsumierenden Prozessen. Dazu ist eine Zwischenlagerung in betriessystemeigenen Puffern nötig. Bei nicht global synchronisierter Speicherverwaltung kann die unterschiedliche Verfügbarkeit von Pufferkapazität allein schon zu Indeterminismus führen. So kann z.B. bei Puffermangel das Schreiben von Daten vom BS abgelehnt werden, falls nichtblockierender Betrieb vorliegt. Allgemein (so auch z.B. in UNIX) muß nicht a priori garantiert sein, daß die von der Anwendung gewünschte Anzahl von Daten auch wirklich geschrieben wird. Sind keine expliziten Synchronisationsmechanismen für eine globale Speicherverwaltung vorgesehen, muß wenigstens eine möglichst große Homogenität im replizierten System gewährleistet sein, z.B. bezüglich der Prozeßlandschaft (gleiche Prozeßkonstellationen für vergleichbare Last) und der Füllzustände auf externen Datenträgern im lokalen Zuständigkeitsbereich. Vollständige Sicherheit vor Indeterminismus kann aber vor allem im Grenzlastbereich nicht mehr garantiert werden.

Bzgl. der globalen Systemschnittstellen zu der Systemumwelt ist die Ausgabe unkritisch, da sie (bei einem Über-Alles-Konzept) durch den Votierungsvorgang synchronisiert wird. Die Eingabe jedoch ist der Gefahr der Replikationsfehler ausgesetzt. Bei asynchroner Abtastung

1. Auf Fehlererholung anstelle Fehlermaskierung beruhende Fehlertoleranzverfahren sind hier schutzlos dem Indeterminismus bei wiederholtem Programmlauf des (Ersatz-) Prozesses ausgeliefert, falls nicht die Systemuhr neu gestellt wird.

unstrukturierter (z.B. analoger) Eingangssignale ist auch bei intakten Sensoren Indeterminismus unvermeidlich. Selbst bei zeichenorientierter Eingabe über digitale Leitungen mit hohen Störabständen und deshalb geringer Fehlerwahrscheinlichkeit stellt die möglicherweise unterschiedliche Anzahl zum Lesezeitpunkt asynchron bereits eingetroffener Zeichen ein Problem dar, wenn das nichtblockierende Lesen einer unbestimmten Anzahl von Zeichen erlaubt ist, z.B. um einen Eingabekanal “abzupollen”. I.a. wird davon direkt die Zahl der Poll-Zyklen abhängen und damit die Prozeß-Kern-Interaktion beeinflußt, sodaß zwangsläufig die Synchronisation verloren geht. In diesem Fall muß zumindest diese Anzahl der zurückgegebenen Zeichen bei jeder Rückkehr des read-Systemaufrufes in den Anwendermodus abgeglichen werden.

3.2.3 Unterbrechungen

Unterbrechungen sind unkorreliert mit dem Programmfluß eintreffende, diesen aber möglicherweise beeinflussende äußere Ereignisse. Indeterministischer Programmlauf kann nur verhindert werden, wenn entweder die Unterbrechung selbst eine Synchronisation bewirkt, oder aber periodische Synchronisation für Unterbrechungen nur definierte Einwirkungszeitpunkte zuläßt. Auf alle Fälle ist ein expliziter Abgleich erforderlich.

.

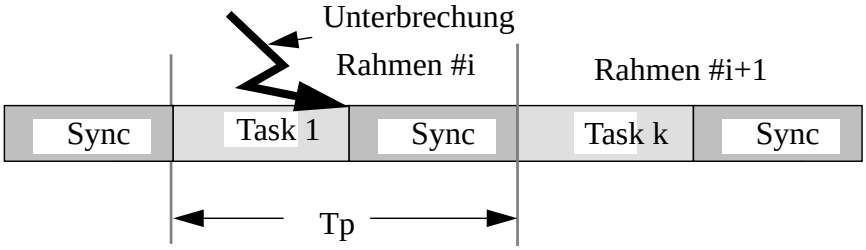


Abb. 5 Signaleinphasung bei Rahmensynchronisation

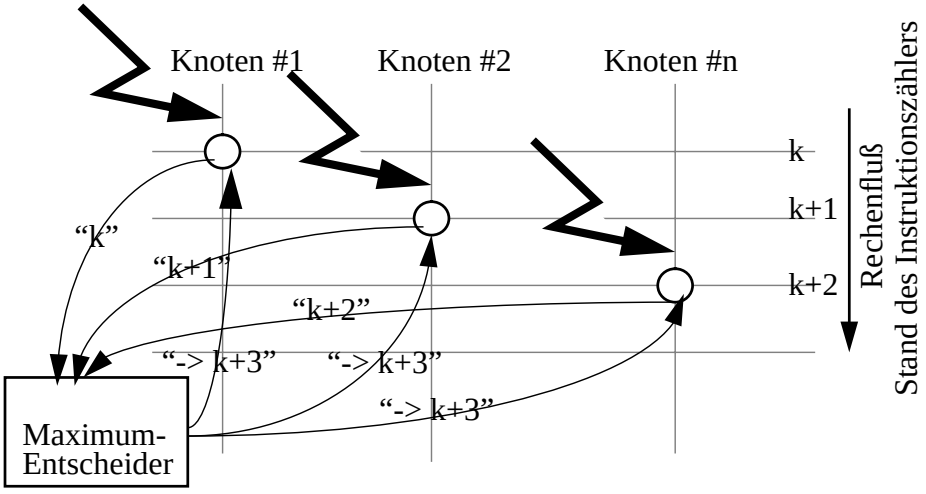


Abb. 6 Ereignissynchronisierung mit Vorlaufausgleich

Im einzelnen ergeben sich folgende Möglichkeiten zur Einsynchronisierung von asynchronen Unterbrechungen:

- a) **taktsynchron**; anwendbar bei taktstarrer Hardwareimplementation der Fehlertoleranz. Das asynchrone Ereignis wird mit Hardwaremitteln (i.d.R. mehrstufige Latches) in den gemeinsamen Taktfluß eingephast, um die erforderlichen zeitlichen Bedingungen (Setup- und Holdzeit) bei der Bewertung durch eine Synchrontaktflanke zu erfüllen.
- b) Rechenfluß-synchron bei **Rahmensynchronisierung**; gestattet ausschließliche Softwareimplementation. Diese Synchronisationsvariante ist im Vergleich zur starren Taktsynchronität loser, indem der Hardwaremechanismus a) auf die Softwareebene abgebildet wird. Sie basiert auf einem globalen Scheduling von Zeitscheiben, deren Dauer durch synchronisierte periodische Ereignisse des Programmflusses bestimmt ist. Diese Ereignisse kennzeichnen einen expliziten Wechsel in den Kernmodus. Hier wechseln also Phasen der Taskbearbeitung zyklisch mit Betriebssystemaktivitäten, siehe Abb. 5. Während der letzeren wird mit einem Verfahren zur interaktiven Konsistenz eine gemeinsame Systemsicht über dieses Intervall betreffende Unterbrechungen hergestellt, die dann auf die Anwendertask zum bereits synchronisierten Kerneintrittspunkt wirken. Dieses Verfahren wurde z.B. im System SIFT [FrWe82] angewendet. Dort war eine iterative Programmstruktur der Anwendertasks vorausgesetzt, was im geplanten Anwendungsfeld Echtzeit-Prozeßsteuerung i.a. nicht hinderlich ist wegen der hier ohnehin notwendigen Periodizität der Bearbeitung bei der Istwerterfassung und Stellgrößenermittlung. Für den allgemeinen Fall ist damit allerdings eine Einschränkung der Softwaretransparenz gegeben. In SIFT konnten mit diesem Verfahren ohne Einsatz von Synchronisations-Hardware die während eines Rahmens sich akkumulierenden Laufzeitunterschiede auf maximal 100µs angeglichen werden.
- c) **Ereignissynchronisierung** mit Vorlaufausgleich, siehe Abb. 6. Hier werden zunächst Unterbrechungen lokal erfaßt und ihr Einwirkungszeitpunkt relativ zum Instruktionsfluß ermittelt. In Zusammenarbeit mit einer (möglicherweise verteilt ausgeführten) Entscheidungsinstanz wird danach ermittelt, welches Prozeßexemplar bereits am weitesten fortgeschritten ist, um die Nachzügler daran anzugleichen. Nachteil an diesem Verfahren ist, daß ein global eindeutiges Maß für das Fortschreiten im Programmfluß benötigt wird. Der absolute Wert des Instruktionszeigers (*program counter*) kann diese Aufgabe nicht erfüllen; vielmehr wird ein Zähler benötigt, der nach jeder Instruktion inkrementiert wird. Wegen der hohen Zählfrequenz ist unbedingt eine Hardwareunterstützung dazu erforderlich, um den sonst erheblichen Verlust an Rechenleistung zu begrenzen. Zudem sind Mittel erforderlich, um den Programmverzug auszugleichen. Da ohne eine Grundsynchronität wie in den Verfahren a) oder b) sich keine verläßliche Aussage über Laufzeitabweichungen der Prozeßreplikate machen läßt[1], ist dieses Verfahren der Ereignissynchronisierung in dieser Form für Echtzeitanwendungen nicht geeignet. Weiterhin nachteilig ist die mangelnde Robustheit dieses Verfahrens, das sich auf die Glaubwürdigkeit des Maximalwertes verläßt; besser geeignet wäre z.B eine Medianentscheidung im Abstimmungsverfahren, die allerdings sowohl ein Fortschreiten wie auch ein Rücksetzen im Instruktionsfluß nötig macht.Wählt man ein Verfahren, das ähnlich wie das zuletzt aufgeführte nur auf der Übereinstimmung auf einem einheitlich zu setzenden Prozeßzustand beruht, wird zwar der Instruktionszähler unnötig, der ausgehandelte Prozeßzustand muß aber umfassend definiert sein und ist daher teuer in der Erfassung: dazu gehören ein kompletter Speicherabzug (Daten- und Stacksegment), das vollständige Registerabbild der CPU, wie auch Daten aus der der Prozeßumgebung. Das Erstellen bzw. Setzen solcher Checkpoints verlangt zusätzliche Mechanismen, die in einem fehlermaskierenden System eigentlich artfremd sind. Diese Mechanismen passen viel besser zu fehlererho-

1. Dies ist das allgemeine "*Halteproblem*" der Logik und Berechenbarkeit [Strehl].

lenden Systemen (*Rollback, Backward Recovery* [AnLe81]), wo sie bereits Grundbestandteil des Fehlertoleranzverfahrens sind. Beispiel für ein fehlertolerantes System dieser Art, daß diese Unterbrechungssynchronisierung tatsächlich verwendet, ist TARGON32 [Borg89]. Hier wird bei jedem Auftreten eines Signales ein vollständiger Rücksetzpunkt erstellt, um im Fehlerfall das (sonst nicht aktiv rechnende) Prozeßreplikat zustandskonform aufzusetzen.

All diese Verfahren werden noch dadurch erschwert, daß bei multiplen Ereigniskanälen zusätzlich das *Reihenfolgeproblem* gelöst werden muß, d.h. daß allen Replikate die äußeren Ereignisse in identischer Reihenfolge zugeführt werden müssen. Zum eigentlichen Abstimmungsverfahren kommt also noch eine Arbitrationsphase hinzu, die auf einer Prioritätsvergabe von Ereignissen beruht.

3.3 Verfahren zur interaktiven Konsistenzsicherung

Alle bisher aufgeführten Quellen für Indeterminismus waren systembedingt, d.h. sie treten schon im fehlerfreien Betriebsfall als Folge von Asynchronität auf. Die Erscheinungsformen sind jedoch i.a. die gleichen, wenn nicht das Systemmodell, sondern das Fehlermodell dafür kausal zuständig ist. Das Problem der zuverlässigen Abstimmung nichtdeterministischer Eingabedaten ist identisch mit der Ausgrenzung “bösartig” verfälschter Daten; solche Fehler heißen “*byzantinische Fehler*”. Um sie tolerieren zu können, wurden Algorithmen zur interaktiven Konsistenz auf der Basis abstrakter Modelle definiert und untersucht [Lam82], [Dol82], [DolStro85]. Sie erlauben die Erkennung und Tolerierung solcher Fehler, indem zwischen den funktionierenden Einheiten durch Informationsabtausch eine Übereinstimmung über die allgemeine Systemsicht erzielt wird, die es ermöglicht, gemeinsame Entscheidungen zu treffen. Diese *byzantinischen Übereinstimmungsprotokolle* sind genauso geeignet, systembedingten Indeterminismus zu eliminieren.

Zustimmungsproblem:
(*agreement problem, reliable broadcast problem*)
- ein einziger Knoten verfügt über die Eingabe
- alle korrekten müssen zustimmen (gleiche Ausgabe)

Übereinstimmungsproblem:
(*consensus problem*)
- jeder hat Eingabe
- gleiche Ausgabe bei allen korrekten Einheiten

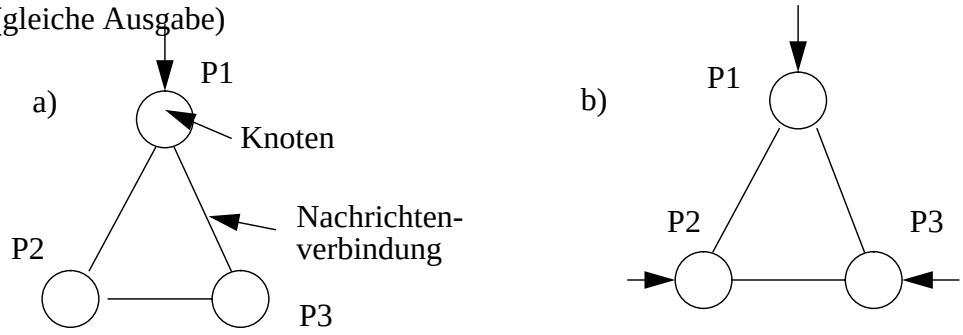


Abb. 7 Zustimmung und Übereinstimmung

Abb. 7 stellt das Grundszenario für das allgemeine Übereinstimmungsproblem in Abhängigkeit von der Eingabeverteilung im System dar. Ziel ist, daß alle korrekten Beteiligten (Rechnerknoten, Prozessoren) zu einer konsistenten Ausgabe (gemeinsames Ergebnis) gelangen. Ist

nur ein zentraler Eingabesensor unter Kontrolle eines einzigen Knotens vorhanden, muß man eher von einem Zustimmungsproblem (*agreement problem*) sprechen. Diese Problemstellung entspricht der eines zuverlässigen Rundspruchs. Der allgemeinere Fall des Übereinstimmungsproblems (*consensus problem*) geht von replizierten Eingabesensoren aus. Beide Problemkonfigurationen sind jedoch für Untersuchungen zur Lösbarkeit ineinander überführbar, zumindest in synchronen Systemen.

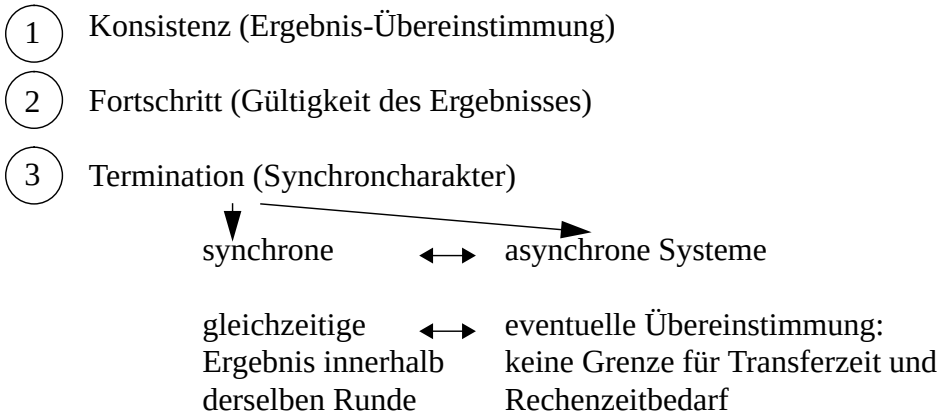


Abb. 8 Problemklassifizierungskriterien

Unterschiedliche Korrektheitskriterien für die Übereinstimmung haben zu speziellen Problemdefinitionen geführt. Abb. 8 listet solche Problemklassifizierungsmerkmale auf. Die Kriterien 1 und 2 sind die ***Bedingungen der interaktiven Konsistenz***. *Ergebniskonsistenz* bezieht sich darauf, daß alle intakten Beteiligten zu der gleichen (oder möglicherweise einer ähnlichen) Ausgabe gelangen, Fortschritt bzw. *Gültigkeit* heißt, daß die konsistente Ausgabe eine Funktion der korrekten Eingabewerte ist (die Triviallösung konstante Ausgabe entfällt damit). Schließlich läßt sich noch eine Unterscheidung nach Synchroncharakter treffen: i.a. wird davon ausgegangen, daß das Ergebnis durch eine komplexe Protokollabwicklung in mehreren synchronen Nachrichtenaustausch-Runden gleichzeitig ermittelt wird. Besteht hingegen keine Grenze für die Nachrichtenübertragungszeit und für den Rechenzeitbedarf des Protokolls, kommt es nur eventuell zu Übereinstimmungen.

Es ist bekannt, daß das Übereinstimmungsproblem in solchen vollständig asynchronen Systemen sogar bei Annahme von nur eimem einzigen Fehler keine deterministische Lösung besitzt [FLP85]. Die meisten realen Systeme gehören jedoch zu den synchronen, wobei darunter verstanden wird, daß sich die erwähnten Grenzen angeben lassen[1]. Im weiteren Verlauf werden nur in diesem Sinne synchrone Systeme betrachtet.

Manche der im weiteren aufgeführten Arbeiten, die sich mit interaktiver Konsistenz beschäftigen, treffen noch weitere Grundannahmen über die Systemumgebung und spezielle Eigenschaften des Fehlermodells. So kann häufig nicht zwischen Prozessor- oder Verbindungsfehlern unterschieden werden - die Nachrichtenübertragung wird damit als ideal fehlerfrei angenommen - und zweifelsfreie Identifikation des Absenders wird vorausgesetzt, was für die Praxis Punkt-zu-Punkt-Verbindungen zwischen den Knoten bedeutet. Diese Verbindungstopologie ist auch deshalb von Interesse, weil Netztrennungen ausgeschlossen werden sollen. Bei nichtvollständiger Vernetzung sind bestimmte Anforderungen an die *Konnektivität* (s.u..) in

1. Zur Unterscheidung von Synchronität in diesem Sinne und perfekter Synchronisation (absolut zuverlässiger Zentraltakt) findet man gelegentlich die Bezeichnung “partiell asynchron” [Sim90].

Abhängigkeit vom zugundeliegenden Fehlermodell zu stellen.

Folgende spezifische Problemstellungen haben sich herausgebildet:

- **Problem der Byzantinischen Generäle** [Lam82](synchrones Zustimmungsproblem - *synchronous agreement*).

  Konsistenz: alle korrekten Knoten kommen zum gleichen Ergebnis.
  Gültigkeit: ist der die Eingabe verwaltende Knoten intakt, folgen ihm alle intakten Knoten (die Ausgabe entspricht dann der Eingabe).
- **Übereinstimmungsproblem** (synchroner Konsens - *consensus problem*).

  Konsistenz: wie oben.
  Gültigkeit: wenn alle Eingaben der intakten Knoten gleich sind, ergibt sich Ausgabe = Eingabe.
- **Abgeschwächte Übereinstimmung** [Lam83] (Konsens oder Zustimmung - *weak consensus, weak byzantine generals problem*)

  Konsistenz: wie oben.
  Gültigkeit: Nur wenn das gesamte System fehlerfrei ist, muß die Ausgabe der Eingabe entsprechen.
- **Kreuzfahrer-Zustimmung** [Dol82] (*crusader agreement*).

  Konsistenz: Übereinstimmendes Ergebnis der intakten Knoten nur gefordert, wenn der Eingabeverwalter intakt ist.
- **Annähernder Konsens** [D83] (*approximate agreement*).

  Konsistenz: anstelle eines festen Ergebnisses gilt ein Entscheidungsbereich für die Ausgabewerte.
  Gültigkeit: Der Ausgabebereich ist als Unterbereich abgeleitet vom Eingabebereich. Nur im fehlerfreien Fall müssen beide übereinstimmen.

Die obenangeführten Literaturangaben sind nur repräsentative Beispiele. Sie untersuchen die theoretischen Lösbarkeitsvoraussetzungen für die einzelnen Problemstellungen[1] und stellen Agorithmen dafür vor. Diese Voraussetzungen sind Eigenschaften der Systemumgebung und beziehen sich auf Zusammenhänge zwischen

- **Replikationsgrad** n = Anzahl der redundanten Knotenexemplare,
- **Fehlertoleranzgrad** t (*resiliency*) = Maximalzahl der erlaubten fehlerhaften Knoten,
- **Konnektivität** C (Zusammenhangsgrad) = minimale Zahl der Knoten, bei deren Wegnahme das Netz zerfällt,
- **Komplexität der Kommunikation** = Gesamtanzahl der ausgetauschten Nachrichten,
- **Zeitkomplexität** m, gemessen in Anzahl der Nachrichtenaustauschrunden, und der
- **Rechenkomplexität** als Rechenaufwand für eine Runde.

Es zeigt sich, daß bei vereinfachter Fehlervorgabe die notwendigen Bedingungen an die

1. Zur Beweisführung bei byzantinischen Fehlern für die meisten der o.a. speziellen Übereinstimmungsprobleme wird der geschlossene Ansatz [FLM85] empfohlen.

Systemumgebung erheblich reduziert werden können. Dies bezieht sich vor allem auf den Replikationsgrad. Die wichtigsten Ergebnisse sind in der Tabelle Abb. 9 festgehalten.

Drei Fehlerklassen sind von wesentlicher Bedeutung (siehe auch Kapitel 5.4.5):

- Unterlassungsfehler (persistent oder nicht persistent),
- durch Authentifikation mithilfe von Signaturbildung erkennbare byzantinische Fehler und
- allgemeine byzantinische Fehler.

Die Signaturbildung ist eine sehr wirkungsvolle Zusatzmaßnahme, die gegen Replikationsfehler bei der Weiterleitung von Nachrichten schützt. Ohne sie gilt die - zunächst überraschende - Feststellung, daß im allgemeinen (byzantinischen Fehler-) Fall eine Fehlermaskierung in einer TMR-Konfiguration nicht ausreicht, um einen einzigen Fehler tolerieren zu können, sondern daß dazu wenigstens 4 Knoten erforderlich sind. Die Systemkonfiguration in Abb. 7 ist also gar nicht konsensfähig. Die Anforderungen an die Konnektivität sind ebenfalls hoch. Erst wenn der Zusammenhangsgrad mindestens C=2t+1 beträgt, kann ein Netzwerk ein vollständig verbundenes simulieren, da sich immer eine Mehrheit von t+1 korrekten Prozessoren zur Weiterleitung findet.

	Byzantinische F.	BF mit Auth.	Unterlassungsfehler
Fehler-Knoten-relation	n >= 3t+1	n>=2t+1	n>=t+1
Konnektivität	C>=2t+1	C>=t+1	C>=t+1
Minimale Rundenzahl	m=t+1	m=t+1	m=t+1

Abb. 9 Notwendige Bedingungen für die Lösung synchroner Übereinstimmungsprobleme

Die Bedingung für die Zeitkomplexität m=t+1 bedeutet bei großen Netzen und dort wünschenswert hoher tolerierbarer Fehlerkomponentenzahl t großen Aufwand. Sie stellt jedoch ein Optimum dar, und zwar unabhängig von der gewählten Fehlerklasse [DolStro82]. Eine besondere, unerwünschte Eigenschaft ist, daß bei synchronem Rundenentscheid selbst bei geringerer Fehlerzahl F<t das Protokoll nicht schneller terminiert [DRS82].

Die Nachrichtenkomplexität hängt stark vom jeweiligen Algorithmus und der Fehlervorgabe ab. [DR85] haben jedoch auch hierfür untere Schranken angegeben. Man hat sie bisher jedoch nur für bestimmte t im byzantinischen Fehlermodell erreicht. Danach benötigt ein Protokoll zur byzantinischen Übereinstimmung, das t Fehler toleriert, bei allgemeinem byzantinischen Fehlermodell im schlimmsten Fall mindestens O(nt), mit Authentifikation $O(n+t^2)$ Nachrichten, bei letzterem mit O(nt) Unterschriften.

Bisher war nur die Rede von deterministischen Algorithmen. Tatsächlich gibt es probabilistische Ansätze [Tou84], [Ben90], die effizienter als die deterministischen zu korrekter Übereinstimmung führen können, und die sich auch in asynchronen Systeme einsetzen lassen [Rei87].

In der Tabelle Abb. 9 fällt auf, daß sogar für das einfachste Fehlermodell, daß nur Unterlassungsfehler berücksichtigt, die Zeitkomplexität nicht geringer ist als in den beiden byzantinischen Modellen. Tatsächlich werden die Verhältnisse jedoch einfacher, wenn man noch gewisse Atomaritätseigenschaften für den Nachrichtentransport zur Eingabeverteilung voraussetzen kann. Wird für das Konsensproblem Abb. 7 b), das sich ja aus dem Zustimmungsproblem a) durch n-fache Überlagerung ergibt, nicht nur die Zuverlässigkeitseigenschaft des Rundspruchs gefordert, sondern darüberhinaus auch noch die Unteilbarkeit - alle intakten Knoten erhalten entweder die gleichen (korrekten) Eingabewerte eines Prozessors mitgeteilt oder gar keine, d.h. daß sich Unterlassungsfehler im System homogen auswirken, kann durch ein einfaches Verteilungsprotokoll [Echt90] auch bei Nichtdeterminismus der Eingabedaten in nur einer Runde ein Konsens erzielt werden. Replikationsfehler müssen dann nicht berücksichtigt werden, alle Knoten erhalten dieselbe Eingabemenge und können lokal eine einmütige Entscheidung fällen. Ein solches Verteilungsprotokoll maskiert “Fehler” (aus dem Rahmen fallende Eingabewerte), z.B. durch einen Entscheid auf der Basis des Mittelwerts oder Medians, bei einer Nachrichtenkomplexität von $O(n^2)$ - jeder sendet jedem seinen Eingabewert zu (Abb. 44 in Kapitel 5.8.2. kann dazu als Illustration dienen). Ist die Befähigung zum atomaren Rundspruch im Kommunikationssystem bereits verankert, reduziert sich dieser Aufwand auf $O(n)$[1] (Broadcast-) Nachrichten. Dieser Ansatz wird im System ATTEMPTO verfolgt.

Ganz allgemein läßt sich zeigen, daß zur Tolerierung von Replikationsfehlern in der Eingabe ein mehrstufiges Maskierungsverfahren geeignet ist, das zunächst in der (den) ersten Stufen den Indeterminismus im System durch ein Übereinstimmungsprotokoll beseitigt und daraufhin durch eine “deterministische” Maskierung zu einem eindeutigen Ergebnis gelangt [DaWa78].

1. Dies gilt auf der höheren Betrachtungsebene. Das Kommunikationssystem kann allerdings selbst wieder auf einem Übereinstimmungprotokoll aufbauen, falls nicht die Hardware schon die geforderten Rundsprucheigenschaften aufweist.

4. Verteilte Systemdiagnose zur Fehlermaskierung

Ein Verfahren zur Fehlermaskierung hat immer zum Ziel, fehlerhafte Komponenten zu disqualifizieren, d.h. ihren Einfluß auf die Erbringung der geforderten Systemdienstleistung - einer Ausgabe an die Systemumwelt - zu unterdrücken. Dazu ist in erster Linie Fehlererkennung erforderlich, um korrektes Verhalten von fehlerhaftem unterscheiden zu können. Eine zusätzliche Fehlerlokalisierung ist allgemein leicht möglich; sie ist jedoch nur dann von Interesse, wenn daraufhin eine Fehlerbehandlung vorgesehen ist, im klassischen Verfahren also gar nicht. Die Fehlererkennung in deterministischen Systemen beruht auf einem Vergleich der Ausgaben replizierter, also gleichartiger Komponenten, wobei sich z.B. nur die Ausgabe durchsetzt, für die eine mehrheitliche Übereinstimmung besteht. In softwareimplementierten Maskierern können Vergleiche nur sukzessive paarweise durchgeführt werden; erst die geschlossene Auswertung aller Vergleichsergebnisse erlaubt einen Maskierungsentscheid.

Von anderer Motivation geprägt, sich jedoch ähnlicher Mittel bedienend, sind Verfahren zur Fehlerdiagnose auf Systemebene. Hier ist Fehlerlokalisierung die haupsächliche Aufgabe, die auf dezentralen Tests zur Erkennung fehlerhafter Komponenten aufbaut, und die einzelnen Testergebnisse (zusammengefaßt im Fehlersyndrom) für ein konsistentes Diagnosebild auswertet. Wie bei der Fehlermaskierung wird zur Fehlererkennung eine Maximalfehlervorgabe verlangt: für eine korrekte Diagnose darf nur eine begrenzte Menge von Komponenten defekt sein. Die Ähnlichkeit in der Fehlererkennungsmethode wird noch deutlicher, wenn die Systemdiagnose in homogenen Systemen auf Vergleichstestmodellen basiert. Verteilte Maskierer entsprechen verteilten Diagnoseeinheiten. Im Gegensatz zur Fehlermaskierung, die i.a. alle Ergebnisse für einen Maskierungsentscheid bei allen verteilten Instanzen benötigt, bemühen sich Verfahren zur Fehlerdiagnose auf Systemebene um insgesamt minimalen Aufwand an Ergebnisvergleichen (und damit Testaufwand), indem die dezentralen Diagnoseeinheiten in begrenzten Testnachbarschaften arbeitsteilig vorgehen. Diesen Vorteil kann man sich für eine Fehlermaskierung zunutze machen. Wegen der für Vergleichstests erforderlichen Eigenschaft der *Fehlersymptomverschiedenheit* im Fehlermodell basiert die Maskierung auf einem Paarentscheid (2-von-N-Maskierungs-System). Auf die Diagnoseaufgabe der Lokalisierung von Defektkomponenten kann verzichtet werden. Stattdessen ist die Auswahl eines korrekten Ergebnisses und dessen Ausgabe an die Systemumwelt die Hauptaufgabe, die sich an die Fehlererkennung anschließt.

Diesen Ansatz verfolgt das ATTEMPTO-System. Der Mechanismus zur Fehlermaskierung beruht unmittelbar auf einer verteilten Systemdiagnose. Um die Möglichkeiten und Grenzen aufzuzeigen, schließt sich nun ein kurzer Überblick über Modelle und Verfahren der Selbstdiagnose auf Systemebene an.

4.1 Diagnosemodelle

Das Themengebiet wurde wesentlich beeinflußt durch die graphentheoretischen Arbeiten von Preparata, Metze und Chien [PMC67]. Das ***PMC-Modell*** benutzt Diagnosegraphen, in denen Prozessoreinheiten durch Knoten und die Testbeziehungen zwischen diesen durch gerichtete Kanten dargestellt sind. Es schafft die Grundlage zu Betrachtungen des *Analyseproblems*: “Welche Diagnostizierbarkeitseigenschaften hat ein gegebener Graph ?”, und des *Syntheseproblems*: “Wie müssen solche Graphen gestaltet sein, damit sich optimale Diagnostizierbarkeit ergibt ?”.

Das PMC-Modell macht folgende Voraussetzungen:

- homogene Systeme (Komponenten gleichen Typs: Prozessoreinheiten)
- begrenzte Anzahl von Fehlern (Maximalfehlervorgabe)
- ideale Fehlerüberdeckung (Testvollständigkeit)
- separate Testverbindungen zwischen Tester und Getestetem
- ideale Kommunikation (fehlerfreie Nachrichtenübertragung) zur Testresultatübermittlung
- statisches Fehlermuster während der Abarbeitung des Diagnosealgorithmus (Fehleratomarität), d.h. weder zusätzliche Ausfälle noch Reparaturen
- Testergebnisse fehlerhafter Tester sind unzuverlässig (wahr oder falsch = symmetrische Testinvalidation, siehe Abb. 10)
- Fremdtests
- zentrale Diagnoseinstanz im Perfektionskern des Systems mit separaten Verbindungen zu allen Testern.

Diese Annahmen sind in den meisten praktischen Systemen nicht realistisch; sie vereinfachen jedoch sehr wirkungsvoll die Betrachtungen.

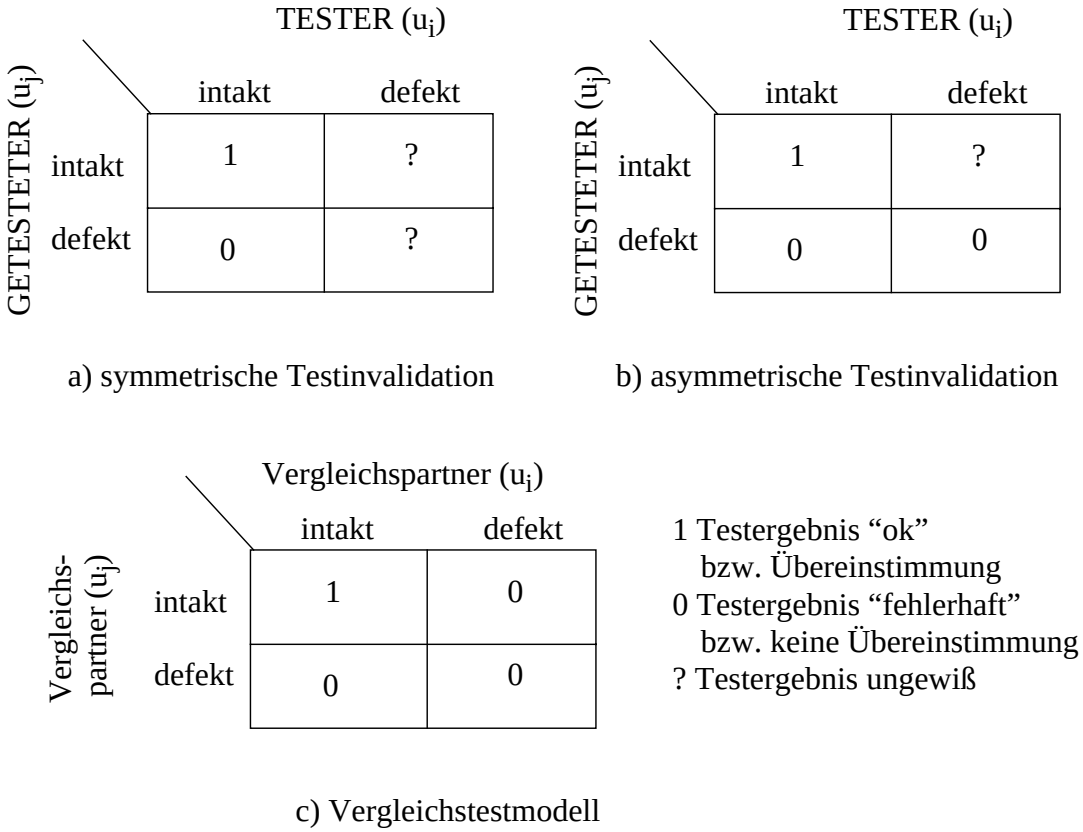


Abb. 10 Testresultatannahmen im PMC-Modell (a), bei asymmetrischer Testinvalidation (b) und im Vergleichstestmodell (c)

Darüberhinaus sind noch stärkere Vereinfachungen getroffen worden. In [BGM76] wird *asym-*

metrische Testinvalidation eingeführt. In diesem veränderten Fehlermodell wird vorausgesetzt, daß eine defekte Einheit immer korrekt als fehlerhaft erkannt wird, auch wenn der Tester selbst fehlerhaft ist. Das Vergleichstestmodell [Malek80], [DC81], [CHak81], in dem anstelle von Fremdtests Relativtests verwendet werden, geht noch weiter: nur wenn beide Vergleichspartner intakt sind, kommt es zu einer Übereinstimmung der Resultate, das Ergebnis “0” zeigt an, daß wenigstens einer, möglicherweise aber auch alle beide defekt sind. Dabei ist *Fehlersymptomverschiedenheit* vorausgesetzt: zwei defekte Einheiten gelangen nie zum gleichen Ergebnis. Eine solche Annahme ist dann realistisch, wenn bei geringer Wahrscheinlichkeit für die zufällige Berechnung eines korrekten Ergebnisses durch einen fehlerhaften Prozessor der Wertebereich für Ergebnisse sehr groß ist[1].

[Amm82] untersucht Diagnostizierbarkeitseigenschaften auch in anderen Vergleichstestmodellen, in denen Fehlersymptomverschiedenheit nicht gilt.

Das PMC- wie auch das Vergleichstestmodell leidet in seiner ursprünglichen einfachen Form insbesondere an den folgenden Unzulänglichkeiten:

- Reale Tests decken nicht alle Fehler in den getesteten Einheiten ab. Dies gilt für alle Arten von Tests. Die Fremdtests des PMC-Modells müssen alle Funktionen der benachbarten Einheit und auch die Testverbindung von außen testen, i.d.R. ist jedoch der Zugriff von außen physikalisch eingeschränkt. Im Vergleichstestmodell werden dagegen Resultate von lokalen Berechnungen verglichen. Damit eine Aussage über die Gesamtfehlerfreiheit der Rechnereinheit möglich ist, müssen diese Berechnungen alle Ressourcen (Subkomponenten) der Einheit nutzen. Die Berechnung besteht also in idealer Weise in der Durchführung eines vollständigen Selbsttests. Für Selbsttests ist aber immer ein fehlerfrei angenommener Systemkern nötig (*Hardcore, Perfektionskern*). Der Vollständigkeit stehen sowohl diese praktischen Einschränkungen wie auch der hohe Zeitbedarf entgegen. Die Methode des *Job-Result-Comparison* [Malek82], [ADC81], [DaSK85] im Vergleichstestmodell benutzt normale Jobausgaben für den Vergleich. Die Testunvollständigkeit führt dazu, daß Fehler nur mit einer bestimmten Wahrscheinlichkeit aufgedeckt werden. Insofern besteht eine Ähnlichkeit mit intermittierenden Fehlern [DaSK85]. In maskierenden Systemen können die unentdeckten, latenten Fehler während der Jobausführung geduldet werden, solange die Maximalfehlervorgabe nicht überschritten ist.

  Die Modellierung nichtidealer Fehlerüberdeckung ist durch probabilistische Ansätze versucht worden [Blount77], [FuRa88], [LShin90].

- Insbesondere bei großer Komponentenzahl ist nichtvollständige reguläre Vernetzung die Regel und deshalb eine direkte Verbindung zwischen Tester und Getestetem oft nicht gegeben. Häufig sind jedoch mehrere Umwege vorhanden, die es gestatten, fehlerhafte Nachrichtenverbindungen und im Wege liegende Defektkomponenten zu tolerieren. Ganz allgemein gibt es einen Zusammenhang zwischen dem Zusammenhangsgrad eines Netzes (siehe auch Kapitel 3.3) und dem Grad der Diagnostizierbarkeit (maximal diagnostizierbare Fehler, *Diagnosemaß*), wie unten noch gezeigt wird. Die Weitervermittlung führt zur möglichen Beeinflussung der Testergebnisse durch Dritte. Damit besteht auch die Gefahr der *Replikationsfehler*: bei verteilter Selbstdiagnose können Diagnoseinstanzen mit unterschiedlichen Teilsyndromen versorgt werden, sodaß ein konsistentes System-Diagnosebild unmöglich wird. Ein Ansatz zur Lösung ist darin zu sehen, daß Infor-

1. Siehe dazu eine abschätzende Rechnung anhand einer Bernoulli-Modellierung für fehlerhafte Komponenten in [DaSK85].

mation nur dann weitergeleitet wird, wenn die Nachricht von einem bereits als korrekt erkannten Knoten kommt (Nachrichteninvalidation [KuRe80]). Gegen Replikationsfehler hilft *Authentifikation* durch Signaturen. Praktisch bedeutet dies das Vorhandensein eines zuverlässigen Rundspruchs im System.

- Ein reales System besteht inhärent aus verschiedenartigen Komponenten. Neben den Prozessoreinheiten sind wenigstens noch Verbindungen als separate Komponenten vorhanden. Die Komponenten können über einen unterschiedlichen Grad an Funktionalität verfügen, der sie zum Tester befähigt. Unterschiedliche Komponenten sind außerdem meist unterschiedlich zuverlässig. Die Diagnose fehlerhafter Verbindungen kann z.B. in einem zweiten Schritt auf die Diagnose von Prozessoreinheiten folgen, um zweifelsfrei Verbindungs- von Prozessorfehlern unterscheiden zu können [DiAm84], [LSM82]. Im sogenannten *K-Graphen-Modell* [Mae82] wird ein geschlossener Modellierungsansatz für inhomogene Systeme versucht.
- Ein statisches Fehlermuster ist nur bei kurzer Laufzeit des Diagnosealgorithmus realisitsch. Je höher die Zahl der Knoten, umso langwieriger ist die Diagnose. Auch der Speicherplatzbedarf nimmt unter bestimmten Bedingungen große Ausmaße an [Malek82], z.B. wenn das ganze Netz das komplexe Diagnoseobjekt ist und umfangreiche Syndrominformation gesammelt wird. Dadurch besteht eine höhere Wahrscheinlichkeit für zusätzliche Fehler während der Diagnose. Der hohen Komplexität begegnet man durch Parallelität in der Abarbeitung des Diagnosealgorithmus, indem man Diagnosezuständigkeiten für nur begrenzte Testnachbarschaften im Netz aufteilt (Partitionierung in Subnetze, die t-diagnostizierber sind [Malek82], [DC82]).

  Da auch angenommen wird, daß Fehler während der Diagnosedauer nicht wieder verschwinden, ist das PMC-Modell praktisch nur bei permanenten Fehlern anwendbar.
- Vor allem Dahbura, Mallela und Masson [DM83],[MM78] haben in das Fehlermodell intermittierende Fehler aufgenommen. Die Bedeutung liegt darin, daß das Auftreten temporärer Fehler in realen Systemem weitaus häufiger ist als das permanenter Fehler. Die Diagnostizierbarkeitseigenschaften der Graphenmodelle werden dadurch beeinträchtigt. Wiederholtes Testen muß angewandt werden, um die Fehlerfreiheit einer Einheit zu erhärten. Grundsätzlich können sich natürlich intermittierende Fehler vor dem Tester verbergen. Eine *Hybridfehler-Diagnostizierbarkeit* zielt deshalb nur darauf ab, immer korrekt, aber möglicherweise unvollständig zu diagnostizieren. Außerdem ist es interessant, solche Testanordnungen zu finden, die ganz sicher zumindest die permanenten Fehler lokalisierbar machen [YaMa], was bei gleichzeitiger Berücksichtigung intermittierender Fehler erschwert ist.

Darüberhinaus wurde im PMC-Modell eine ideale zentrale Diagnoseinstanz (*golden unit*) vorausgesetzt, die alle Testergebnisse in einem *Fehlersyndrom* aufsammelt und daraus durch Anwendung eines Diagnosealgorithmus das Fehlermuster (Diagnosebild) ableitet. In der Praxis sind jedoch robustere Ansätze vorzuziehen, d.h. die Diagnosefunktion wird auf die Prozessoreinheiten (Knoten) im System verteilt. Gleichzeitig ist damit der Vorteil der schnelleren Diagnose durch Parallelarbeit verbunden (Partitionierung, s.o.). Da jede Einheit nur eine begrenzte Anzahl von Testverbindungen und damit nur eine eingeschränkte Systemsicht hat, müssen in einer weiteren Phase nach der Teildiagnose (oder der dezentralen Fehlersyndromermittlung) Zwischenergebnisse per Rundspruch im System verbreitet werden und in jeder Einheit zusammengefaßt (im Gesamt-Fehlersyndrom) der Auswertung unterzogen werden. Im weiteren werden wir vorzugsweise verteilte Diagnose im Vergleichsttestmodell betrachten

Wir definieren zunächst unser Vergleichstestmodell.

Die Testanordnung in einem zu diagnostizierenden homogenen System S wird repräsentiert durch einen Diagnose-Graphen G = (U,E). n Prozessoreinheiten u_i (i=1,..,n) bilden die Menge der Knoten U. Je ein Paar von diesen Knoten u_i und u_j (Testpaar) berechnet den gleichen Job J[1]. Nach dem Ende der Berechnung werden die Ergebnisse von u_i und u_j miteinander verglichen und zwar separat sowohl in u_i als auch in u_j. Für das Vergleichsergebnis gilt die Wahrheitstafel Abb. 10c. Die Kommunikation wird als ideal betrachtet. Alle Vergleichsergebnisse als geordnetes Tupel zusammengenommen bilden das Fehler-Syndrom SYN. Alle Testverbindungen bilden die Kantenmenge $E=\{(u_i,u_j) : (u_i,u_j)$ ist ein Testpaar, $i\neq j\}$. Wegen des gegenseitigen Vergleichs sind Kanten ungerichtet, $(u_iu_j) = (u_ju_i)$. Die Nachbarschaft eines Knotens u ist $C(u)=\{u_i : (u,u_i)\in E\}$. Die Kardinalität $|C(u)|$ gibt die Anzahl der Testkanten an, die von u abgehen.

Wichtig für die graphentheoretischen Untersuchungen ist der Begriff der t-Diagnostizierbarkeit.

Definition: Es sei vorausgesetzt, nicht mehr als t Knoten des Diagnosegraphen G produzieren unkorrekte Ergebnisse. G ist dann ***dezentral t-diagnostizierbar***, wenn alle intakten Einheiten u bei allen Syndromen SYN das gleiche korrekte Diagnosebild des Gesamtsystems S erlangen können.

Wie die Testanordnung für t-Diagnostizierbarkeit beschaffen sein muß, sagt das folgende

Grund-Theorem für t-Diagnostizierbarkeit : Ein Diagnosegraph G mit n Knoten ist im oben beschriebenen Vergleichstestmodell dann und nur dann dezentral t-diagnostizierbar, wenn

1.) $t \leq n-2$,

2.) $|C(u)| \geq t$ für alle $u\in U$, d.h. jeder Knoten muß sich mit wenigstens t anderen vergleichen,

3.) für jedes Testpaar u_i,u_j mit $(u_i,u_j) \in E$ und $|C(u_i)| = |C(u_j)| = t$ es wenigstens einen weiteren Knoten $u \in U$ gibt, für den gilt

$(u,u_j) \in E$, $(u,u_i) \notin E$ und $C(u) \neq C(u_j)$, oder
$(u,u_i) \in E$, $(u,u_j) \notin E$ und $C(u) \neq C(u_i)$,

d.h. der sich mit nur einem der beiden vergleicht, dabei aber nicht genau nur dieselben Testpartner wie der andere hat.

Zum Beweis siehe [Amm82].

Im allgemeinen wird man bemüht sein, die Anzahl von Vergleichstests möglichst gering zu halten, um Aufwand zu sparen. Darüberhinaus ist es auch interessant, die Belastung durch Vergleichsaufgaben möglichst gleichmäßig im System zu verteilen. Testgraphen, bei denen dies gelungen ist, heißen t-optimal.

1. I.a. brechnen alle Knoten den gleichen Job, sodaß das Ergebnis einer einmaligen lokalen Berechnung mit dem aller Partner verglichen werden kann.

Definition: t-optimal sind t-diagnostizierbare Testgraphen, die im Vergleich mit allen anderen gleicher Knotenzahl n minimales v=|E| und minimales $d_G=\max\{|C(u_i)| : i=(1,..,n)\}$ besitzen.

Alle t-diagnostizierbaren Testgraphen haben t<=n-2 und v>= $\lceil tn/2 \rceil$. Weiterhin kann gezeigt werden, daß v= $\lceil tn/2 \rceil$ bei gleichzeitiger t-Diagnostizierbarkeit t-Optimalität impliziert.

In [ADC81] werden allgemeine Bildungsregeln für t-optimale Diagnosegraphen angegeben. Solche t-optimale Diagnosegraphen werden auch im System ATTEMPTO verwendet. Der Grad t der Diagnostizierbarkeit wird hier Fehlertoleranzgrad (FT-Grad) genannt. Abb. 11 zeigt als Beispiel die Testanordnungen für FT-Grade 1 <= t <=4. Wie man anhand des obigen Theorems leicht nachprüfen kann, sind sie (n-2)-optimal.

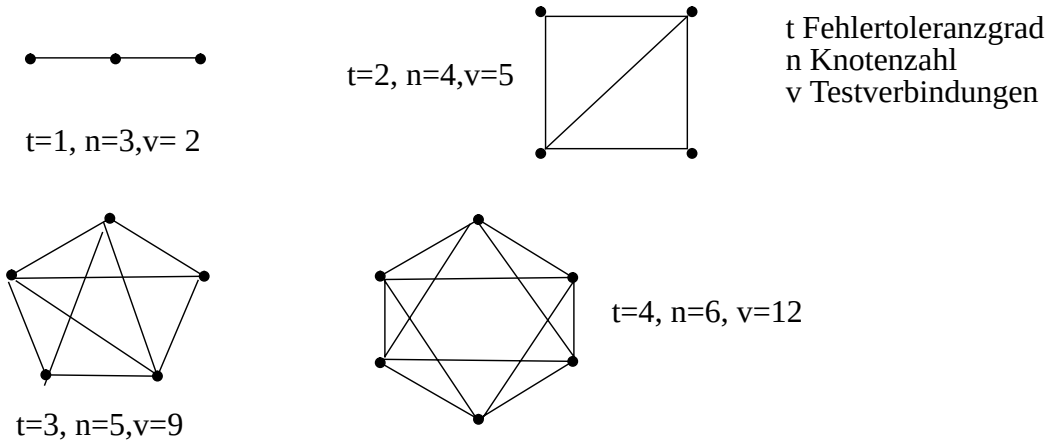


Abb. 11 Fehlertoleranzgradabhängige Testanordnungen in ATTEMPTO

Die Diagnostizierbarkeit im bisherigen Sinn erlaubt die Fehlerlokalisation im Gesamtsystem in einem geschlossenen Diagnosevorgang. Der Vollständigkeit halber sei erwähnt, daß es neben dieser “einstufigen” (*one-step-diagnosis*) auch eine *sequentielle t-Diagnostizierbarkeit* gibt, bei der fehlerhafte Komponenten sukzessive ermittelt werden. Hier folgen auf einen Diagnoseschritt weitere, nachdem zuvor entdeckte Defektkomponenten repariert wurden. Diese Vorgehensweise ermöglicht die weitere Vereinfachung von Testanordnungen [DC79] bezogen auf die Einzelschritte.

4.2 Diagnosealgorithmen

Ein Diagnosealgorithmus ist eine Strategie s zur Fehleridentifikation, die das Syndrom SYN auf ein Fehlermuster F abbildet, **s: SYN -> F**. Eine zentrale Diagnoseinstanz wird zunächst die Testergebnisse in einem globalen Syndrom vereinigen und darauf den Diagnosealgorithmus anwenden[1]. Bei verteilter Diagnose geht es darum, ein übereinstimmendes Systemdiagnosebild F bei allen intakten Diagnoseeinheiten zu erlangen. In der Regel besitzen die Diagnoseeinheiten jedoch nur begrenzte Testnachbarschaften und verfügen deshalb nicht über das vollständige Fehlersyndrom. Sie sind dadurch gezwungen, den Testergebnissen (Teilsyndromen) ihrer Diagnosepartner zu glauben, oder aber sich direkt auf deren fertiges Diagnoseurteil (Teil-

1. Falls es sich um ein einstufiges Verfahren handelt.

Fehlermuster) über andere Einheiten außerhalb des eigenen Zugriffs zu verlassen. I.a. wird vom Teil-Fehlermuster ausgegangen, da es sich - vor allem, wenn jeder Tester gleichzeitig Diagnoseeinheit ist, die sich immer selbst für korrekt hält - von einem Teilsyndrom nicht unterscheidet. Die Verläßlichkeit des Diagnosepartners muß allerdings erhärtet werden, wenn sie nicht von vornherein durch eine implizite Fehlermodellvorgabe feststeht, z.B. Fail-Stop-Eigenschaft und ideale Kommunikation. Wie bei der Testauswertung in einer zentralen Instanz eignet sich hierfür Invalidation: eine Diagnoseeinheit wird dem Urteil einer benachbarten nur trauen, nachdem sie sich von deren Korrektheit durch einen selbstdurchgeführten Test überzeugt hat. Ansonsten werden alle Nachrichten dieser Nachbarn verworfen und insbesondere auch nicht weitergeleitet. Dieses Prinzip der ***Nachrichteninvalidation*** wird in fast allen verteilten Diagnosealgorithmen verwendet. Der (logische) Nachrichtenfluß zur Übermittlung von Diagnoseurteilen fließt also im Diagnosegraphen auf denselben Kanten, die auch die Testbeziehungen ausweisen, im PMC-Modell mit gerichteten Kanten allerdings entgegengesetzt der Testrichtung.

Die Praktikabilität eines Diagnosealgorithmus hängt ab von:

- Aufwand (Rechenkomplexität, benötigter Speicherplatz, produzierter Nachrichtenverkehr)
- Verwendungsmöglichkeiten (funktionsbegleitend *on-line* oder separat *off-line*)
- Zuverlässigkeit des Diagnoseergebnisses (Realitätskonformität des zugrundeliegenden Diagnosemodells, Treffsicherheit des Algorithmus, Robustheit des Algorithmus).

Der Aufwand für eine verteilte Sytemdiagnose kann sehr unterschiedlich sein, dies hängt sehr stark von den Vorraussetzungen des Fehlermodells ab. Unbestimmte Testergebnisse defekter Tester, Syndromverfälschungen durch fremde fehlerhafte Knoten, unvollständiges Testen, temporäres Fehlerauftreten erschweren die Aufgabe. I.a. wird es zu existierenden Fehlermustern verschiedene Syndrome geben, und auch die Abbildung **SYN->F** kann mehrdeutig sein. Es läßt sich i.d.R. leicht nachweisen (in polynomialer Zeit), ob eine Lösung korrekt ist, schwieriger jedoch, ob sie die beste ist. Korrekt ist sie, wenn das Syndrom konsistent zum diagnostizierten Fehlermuster ist, die beste ist sie, wenn sie das höchstwahrscheinliche Diagnosebild erzeugt; das ist - bei hinreichend guter Verfügbarkeit der Einzelkomponenten - meist dasjenige mit minimaler Fehlerzahl im Fehlermuster. Diese Aufgabe gehört zu den NP-vollständigen Problemen [DC79], [Lee90]. Tatsächlich machen alle bekanntgewordenen Diagnosealgorithmen - auch die probabilistischen - Einschränkungen im Diagnose- bzw. Fehlermodell und sind so durchaus effizient. Deterministische Diagnoseverfahren sind solche, die unter den Modellvoraussetzungen sicher funktionieren, dagegen werden Verfahren, die mit gewisser hoher Wahrscheinlichkeit vertrauenswürdige Ergebnisse erzielen, als probabilistisch bezeichnet (obwohl sie im Algorithmus selbst nicht mit Zufälligkeiten operieren).

Die “Treffsicherheit” des Algorithmus und die Realitätskonformität des Diagnosemodells bestimmen jedoch nicht allein die Zuverlässigkeit des Diagnoseergebnisses; der Diagnosealgorithmus muß auch mit Fehlersituationen während seiner Abarbeitung umgehen können, d.h. er sollte selbst Robustheits- bzw. Fehlertoleranzeigenschaften besitzen. Damit ist zunächst noch nicht gemeint, daß während des Diagnosevorgangs bereits Veränderungen im Fehlermuster auftreten dürfen, sondern, ob über die Grundforderungen hinaus Verletzungen der Vorgaben des Fehlermodells geduldet werden können. Dazu gehört z.B. die Behandlung redundanter Nachrichten, der Umgang mit Unterlassungs-, Zeit- und oder byzantischen Fehlern im Kommunikationssystem und ein sinnvolles Verhalten in Fehler-Grenzbereichen, etwa bei Verlet-

zung der Maximalfehlervorgabe. Viele verteilte Diagnoseverfahren arbeiten in synchronisierten Phasen, meist jedoch, ohne Aufschluß über den Synchronisationsmechanismus zwischen den i.a. lose gekoppelten Einheiten zu geben. Berechnungsergebnisse für einen Relativtest, erwartete Teilsyndrome von Diagnosepartnern können ausbleiben, wodurch die Gefahr von Verklemmungen besteht. In asynchronen Systemen mit kalkulierbarem Laufzeitverhalten (siehe Kapitel 3.) muß der Algorithmus wenigstens Zeitschranken setzen, um Unterlassungsfehler aufgrund von Knotenausfällen tolerieren zu können.

Diagnosealgorithmen unterscheiden sich auch darin, ob sie im laufenden Betrieb oder nur zur *off-line*-Diagnose verwendet werden sollen bzw. können. Die off-line-Diagnose gestattet hohe Freiheitsgrade. Für eine *on-line*-Diagnose wird verlangt, daß

- die Diagnose entsprechend **effizient** ist, sodaß der Rechenbetrieb nicht zulange unterbrochen wird. Insbesondere ist dies für die Teilaufgabe des Testens zu fordern. Dies spricht gegen umfangreiches Selbsttesten, begünstigt aber die Methode des Jobergebnisvergleichs und besonders nebenläufiges Testen mit Hardwareunterstützung.
- der Diagnosealgorithmus **periodisch** durchgeführt oder aber nach erkanntem Fehler erneut gestartet werden kann. Ein einfacher Mechanismus ist z.B. die Überwachung auf periodische Lebenszeichen (*Heart Beat*), oder das zyklische Fremdtesten zur dezentralen Fehlererkennung. Die bekanntgewordenen on-line-Diagnosealgorithmen versuchen alle, relative Veränderungen am bestehenden Diagnosebild im gesamten System zu verbreiten, anstatt wieder eine vollständige neue Selbstdiagnose auf Systemebene anzustoßen. So können auch synchrone Runden vermieden werden, da die ursächliche Fehlererkennung eine zeitliche Sequenz kausal abhängiger Folgetests bewirkt ([KuRe80], [KuRe81]).
- sich das Fehlermuster schon während der Abarbeitung des Diagnosealgorithmus ändern darf (**dynamisches Fehlermuster)**. Diese Idealvorstellung ist nur bedingt realisierbar. In der Praxis wird verlangt, daß die Änderungen sich “gutmütig” auswirken, und daß Nachrichten Fehler- und Reparaturereignisse eindeutig identifizieren, sodaß es zu keiner Syndromdurchmischung kommen kann. In [Mo83] wird dies durch Zeitstempel erreicht.
- der Algorithmus **adaptive Eigenschaften** aufweist, um auch Rekonfigurationen gerecht zu werden. Dies verlangt einen Mechanismus zur Einigung auf einen geeigneten jeweils neuen Diagnosegraphen. Das probabilistische Verfahren [SBB92] hat diese Eigenschaft. Auch [Mae82], [Mo83] können auf Rekonfigurationen reagieren.

Eine vergleichende Gegenüberstellung verschiedener Diagnosealgorithmen wird in [Mo83] vorgenommen, ebenso auch in [Alt93], wo auch neuere probabilistische Verfahren aufgeführt und feinere Unterscheidungen getroffen werden. In diesem Zusammenhang soll auf eine detaillierte Beschreibung einzelner Verfahren verzichtet werden, abgesehen von einem vergleichstestbasierten verteilten Diagnoseverfahren mit besonderen Eigenschaften, das für das ATTEMPTO-System Bedeutung hat.

Das Diagnoseverfahren [DC84], [DCF85], [DC87] basiert auf t-optimalen Diagnosegraphen im Vergleichstestmodell. Ein lokaler Diagnoseschritt dient in erster Linie dazu, jedem Knoten Aufschluß über den Fehlerstatus seiner unmittelbaren Vergleichstestnachbarn zu geben. Eine Besonderheit dieses Verfahrens ist, daß auch jeder Knoten sich selbst diagnostiziert; im Gegensatz dazu gehen fast alle anderen verteilten Verfahren davon aus, daß die Diagnoseeinheit sich selbst a priori für intakt hält. Diese Annahme ist in den meisten Fällen auch zulässig, solange das Diagnoseziel - die Übereinstimmung der intakten Einheiten - dadurch nicht gefährdet ist.

Eine lokale Selbstdiagnose hat Vorteile für die Fehlerisolation, da ein Fail-Stop-Verhalten unterstützt wird. Wichtig für die Wirksamkeit ist, daß eine hohe Wahrscheinlichkeit für die Fähigkeit zur Selbstbeurteilung besteht. Dies ist in einem Fehlermodell gegeben, in dem primär temporäre Fehler angenommen sind. Die Fehlerisolation begünstigt auch den Einsatz in sicheren Systemen, in denen es vor allem auf die Integrität der Ausgaben ankommt. Aus diesem Grund ist das Verfahren auch gut als Basis für ein softwareimplementiertes fehlermaskierendes System geeignet.

Für eine allgemeine Systemdiagnose besitzt das Verfahren noch einen anschließenden Schritt für eine Netzwerkdiagnose, der im Abgleich der lokalen Diagnosebilder besteht. Hierauf soll jedoch nicht näher eingegangen werden.

Abb. 12 zeigt ein Struktogramm des lokalen Diagnosealgorithmus. Der Nachrichtenaustausch geht in zwei Phasen vor sich. Zunächst tauschen alle Nachbarn ihre Jobresultate miteinander aus und vergleichen sich einander. Wenigstens eine positive Übereinstimmung berechtigt eine Einheit, den Status ihrer Nachbarn zu beurteilen, und den als fehlerhaft (undefiniert) erkannten in der zweiten Phase eine negative Bestätigung (Stop-Nachricht) zuzuschicken. Knoten mit mehr als t Nachbarn werden schon nach der ersten Phase wissen, ob sie intakt oder defekt sind. Die anderen sind jedoch auf die zweite Phase angewiesen.

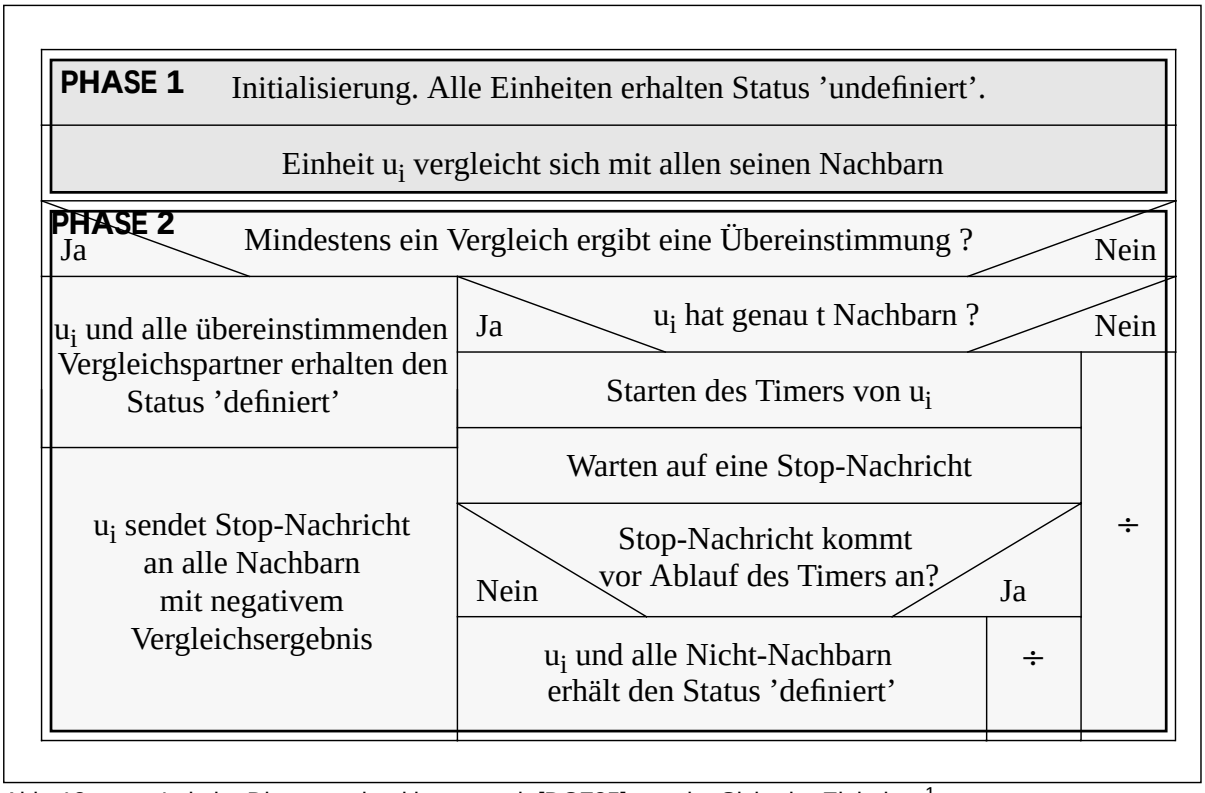


Abb. 12 Lokaler Diagnosealgorithmus nach [DCF85] aus der Sicht der Einheit u_i [1]

Der Beweis der Korrektheit für alle Grade t wird in [DCF85] geführt. Er stützt sich direkt auf das Grundtheorem für t-Diagnostizierbarkeit im Vergleichstestmodell (siehe Kapitel 4.1). Dieses garantiert durch die Testanordnung, daß jeder Knoten immer wenigstens einen korrekten Nachbarn findet, der ihn in seiner Korrektheit durch positiven Testergebnisvergleich bestätigt, oder auf seinen fehlerhaften Zustand durch eine zusätzliche (Stop-)Nachricht hinweisen kann.

1. Struktogramm ähnlich wie in [Alt93].

Allein dann, wenn ein Knoten die nach dem Theorem geringstmögliche Anzahl t von Nachbarn hat, kann der Fall auftreten, daß weder Bestätigung noch Stopnachricht eintrifft. Dann sind aber diese t Nachbarn defekt und es folgt implizit, daß der Knoten selbst, wie übrigens alle weiteren Einheiten auch, intakt ist, da nach Annahme nicht mehr als t Knoten defekt sein können.

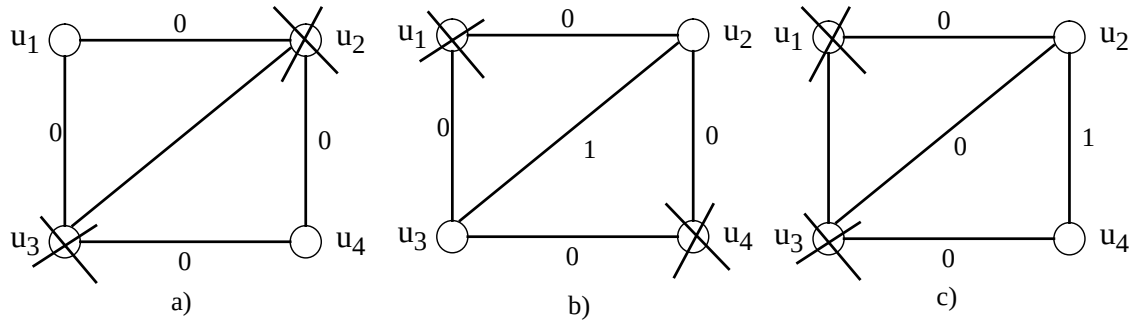


Abb. 13 Nichtäquivalente 2-Fehler-Konstellation im 2-optimalen Diagnose-Graphen

Diese ungünstigtste Fehlerkonstellation entspricht dem Fall a) in Abb. 13, die die Korrektheit des Diagnosealgorithmus für einen 2-diagnostizierbaren Graphen in anschaulicher Weise verdeutlicht. Alle anderen möglichen 2-Fehler-Anordnungen sind unkritisch, da es immer wenigstens einen korrekten Nachbarn gibt. Für 1-Fehler-Konstellationen gilt dies trivialerweise.

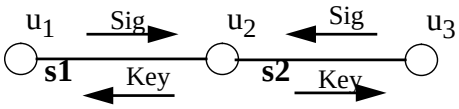
Hinsichtlich des Vorkommens bestimmter Fehlerfälle sind einige Einschränkungen vorausgesetzt: die Timer sind korrekt, und intakte Knoten fallen nicht aus, bevor sie nicht ihre Stopnachricht an defekte Knoten ausgesendet haben. Nachrichten gehen nicht verloren oder werden nicht verfälscht. Die Verbindungen werden ohnehin als ideal vorausgesetzt, da ihr Ausfall nicht diagnostizierbar ist. Ein Problem für die Integrität des Systems stellen auch “taube” Knoten da, deren Fähigkeit zum Nachrichtenempfang verlorengegangen ist. Sie könnten durch das Ausbleiben der Stopnachricht folgerichtig auf eigene Korrektheit schließen. In ATTEMPTO wird dies verhindert, indem auch die Timeoutbenachrichtigung, verpackt in eine Botschaft und an sich selbst versendet, den gleichen physikalischen Empfangsweg nehmen muß wie die Nachrichten von anderen Knoten.

Eine grobe Näherung für den Kommunikationsaufwand ist als Mittelwert im Maximalfehlerfall

$$k < 2v \text{ Signaturen} + v \text{ Stopnachrichten}, \text{ also } k < 3v,$$

und ist wegen des gegenseitigen Vergleichs mit t Nachbarn von der Ordnung O(nt). Obwohl die Voraussetzung für den Algorithmus nur t-Diagnostizierbarkeit ist, ist es von Vorteil, ***optimale*** Testanordnungen zu wählen; das sind t-optimale Anordnungen, die ja geringstes v haben, mit der geringstmöglichen Knotenzahl n=t+2, wie im Beispiel oben verwendet.

Der Algorithmus ist nicht zuletzt auch durch die Verwendung der Timeout-Mechanismen um geringes Botschaftsaufkommen bemüht. Tatsächlich läßt sich der Kommunikationsufwand aber noch stärker reduzieren. Dazu wird eine Variante des Algorithmus anhand eines weiteren Beispiels betrachtet [DC90].



Sig: Signaturnachricht
Key: Schlüssel-Nachricht
s1,s2: Syndrominformation

t=1; n=3

Abb. 14 Geringer Aufwand bei Rollenverteilung im 1-optimalen Graphen

Der Grundalgorithmus verlangt eine Aufteilung in 2 Runden. Verteilt man die Aufgaben des Vergleichens gleichmäßiger auf diese und betraut Knoten, die mehr als t Nachbarn haben, mit besonderen Aufgaben, kann man in symmetrischen t-optimalen Graphen auch im Fehlerfall immer

$$k=2v$$

garantieren.

Abb. 14 soll dies anhand einer 1-optimalen Konfiguration erläutern. u_2 übernimmt die Aufgabe des Vergleichens in der ersten Runde. Dazu erhält diese Einheit die Signatur-Nachrichten ihrer Partner, verschickt aber an diese keine eigenen (Signaturen entsprechen komprimierten Jobergebnissen, um auch die Nachrichtenlänge gering zu halten). Stattdessen antwortet sie in der zweiten Phase mit einem Schlüssel, allerdings nur wenn der Vergleich ihrer eigenen Signatur mit wenigstens einer anderen positiv endet. Dieser stellt für die Partner u_1 und u_3 eine positive oder negative Bestätigung dar und ist auch in diesem Sinne unterscheidbar. Der Schlüssel ist durch eine Zweiwegfunktion F durch Key=F(Sig(u_2)) kodiert, in der Weise, daß der Empfänger u_i eine festgelegte Prüfkonstante A durch Anwendung einer Umkehr- Funktion F* ermitteln kann, A = F*(Key,Sig(u_i)), und zwar nur dann, wenn seine eigene Signatur korrekt ist, d.h. der des Generators des Schlüssels entspricht. A ist eine feste virtuelle Adresse, die den weiteren Programmfluß des Empfängers bestimmt, z.B. die Adresse der Ausgaberoutine, die das Berechnungsergebnis an die Systemumwelt ausgibt. Nur solchen Knoten ist eine Ausgabe erlaubt, die die Bestätigung durch einen korrekten Schlüssel erhalten haben; insbesondere ist dies der Diagnoseeinheit in dieser Dreierkonstellation also nicht gestattet. Bei entsprechend geringer Wahrscheinlichkeit für das unbeabsichtigte Ausführen des Programms an der vorgesehen Stelle kann so wirkungsvoll eine Fehlerunterdrückung vorgenommen werden. Das Dekodieren des Schlüssels mit der eigenen Signatur ersetzt für die Knoten u_1 und u_3 den Vergleichsschritt in der ersten Runde des ursprünglichen Algorithmus, stellt aber ebenfalls einen Vergleichstest dar.

Betrachten wir die möglichen Diagnosesituationen in diesem Beispiel.

s1 s2	Diagnosesituation
1 1	Key an u_1 und u_3
1 0	Key an u_1, NACK an u_3
0 1	Key an u_3, NACK an u_1
0 0	u_2 ist fehlerhaft, Timeout bei u_1 und u_3 liefert dort den Key

Abb. 15 Syndromauswertung bei der Diagnoseeinheit u_2

Ist die Diagnoseeinheit selbst in Ordnung, werden die Partner immer über ihren Zustand informiert und erhalten so die erforderliche Bestätigung. Ist u_2 jedoch fehlerhaft, wird wieder ein

Zeitüberwachungmechanismus benötigt: treffen keine Bestätigungsmeldungen bei u_1 und u_3 ein, oder sind sie ungültig, wird nach Überschreiten der Zeitschranke lokal ein Schlüssel, nun mit der eigenen Signatur kodiert, erzeugt. Es wird vorausgesetzt, daß u_2 nur mit sehr geringer Wahrscheinlichkeit einen gültigen Schlüssel erzeugen kann, wenn sie selbst fehlerhaft ist.

Diese Rollenverteilung läßt sich auch in optimalen Testgraphen höheren Grades anwenden. Bei t=2, n=4 ist dies gut möglich, bei größeren Diagnosemaßen wird dies bei ungeraden t jedoch nicht symmetrisch gelingen. Wenigstens t Diagnoseeinheiten sind notwendig, die sich auch selbst untereinander diagnostizieren.

Erhalten mehrere Knoten eine positive Bestätigung, stellt sich die Frage, wer von ihnen die Ausgabe vornehmen darf. Um dies zu regeln, ist ein zusätzlicher Mechanismus nötig, der jedoch außerhalb der Diagnoseaufgabe liegt.

Die Unterscheidung in negative (NACK) und positive Bestätigung (Key) ist nicht unbedingt erforderlich; in [DC90] wird sie auch nicht getroffen. Ein fehlerhafter Knoten wird eine gültige Key-Nachricht nicht dekodieren können. Auf diese Weise kann aber die Chance wahrgenommen werden, einen defekten Knoten darauf hinzuweisen, sich in einen sicheren Folgezustand (Ruhe oder Selbsttest) zu begeben, falls ihm noch die Möglichkeiten dafür zur Verfügung stehen.

5. ATTEMPTO: Systemkonzept und -implementierung

5.1 Projektverlauf und -umfeld

ATTEMPTO ist ein fehlertolerantes Multi-Mikroprozessor-System, das zur experimentellen Untersuchung von Aspekten der Fehlertoleranz und Parallelverarbeitung sowie von Diagnoseverfahren dient. Das Projekt wurde bzw. wird teilweise von der DFG gefördert.

Die Hardware des Experimentalsystems besteht aus z.Z. 5 Rechnerknoten. Sie sind weitestgehend autonom, d.h. sie verfügen über eigenen Speicher und Hintergrundspeicher, und sind aufgebaut als Single Board Computer (mit 68xxx-Komponenten) auf VMEbus- Steckkarten in einem kommerziellen 19''-Aufbausystem. Das Betriebssystem zur Verwaltung der lokalen Ressourcen ist ein UNIX-SYSTEM-V-Derivat, also ebenfalls eine Standardkomponente. Zusätzlich verfügt das System über einen (einzigen) Netzwerk-Controller (für das Protokoll TCP/IP auf Basis ETHERNET), der als Konzentrator zur Anbindung an das Instituts-Rechnernetz dient.
Zur Erfassung von Meßdaten zur Laufzeit steht eine Hardware-Trace-Einrichtung zur Verfügung, die Buszyklen auf dem VMEbus zusammen mit einer Zeitskala aufzeichnet.

Funktional besitzt das System ATTEMPTO mehrere Ausbaustufen: Die Grundversion ATTEMPTO-1 ist eine fehlertolerante Einbenutzer-Workstation für universelle interaktive Anwendungen. Jobs (Prozeßgruppen) werden behandelt als parallelverarbeitende Einheiten grober Granularität; in Bezug auf Fehlertoleranz kommt Fehlerunterdrückung mit unmittelbarer verteilter Diagnose zum Einsatz. Dieses System wird im weiteren vorgestellt und analysiert. Außerhalb dieses Kapitels wird ATTEMPTO abkürzend als Synonym für ATTEMPTO-1 verwendet. Die zweite Stufe gestattet die Nutzung in einem Netzverbund und eine feinkörnigere Parallelverarbeitung. Hierfür wurde eine Entwicklungsumgebung mit integriertem Expertensystem zur Hardwarediagnose bereitgestellt. Für dieses Teilprojekt ATTEMPTO- 2, das nicht Gegenstand dieser Arbeit ist, sieht das Fehlertoleranzkonzept die Verwendung einer Hybridform von statischer und dynamischer Redundanz in einer oligarchischen Struktur vor, eine Hierarchie mit einem Fehlermaskierungssystem entsprechend der Grundversion auf oberster Ebene.

Das Grundkonzept für die fehlertolerante Workstation, wie auch der Name ATTEMPTO[1] selbst, wurde bereits 1983 durch M.Dal Cin, R.Brause u.a. an der Universität Tübingen definiert und auf der FTCS13 [ABCLR83] präsentiert. Ein besonderer Forschungsschwerpunkt lag in dieser Zeit auf verteilten Diagnosealgorithmen zur Systemselbstdiagnose. Das ATTEMPTO-System war gedacht als Testbett zur Untersuchung dieser Diagnoseverfahren. Die Eignung als Experimentalsystem wird besonders gut durch die modulare Struktur der Fehlertoleranz-Systemsoftware unterstützt. In der Folgezeit wurden verschiedene Teilaspekte simuliert, wie z.B. die Interprozessorkommunikation durch zeitattributierte Petrinetze [Ris87]. Zur Verifizierung von Systemfunktionen sollte eine Systemsimulation auf einer PDP11 unter UNIX Version 7 dienen, die bereits Teile der Originalsoftware verwendete [Brau87]. Eine erste reale Systemimplementation wurde mit Multibus-I-Singleboard-Computern 1986 versucht. Wegen beschränkter Eignung der Hardware - zum Teil bedingt durch den damaligen verhältnismäßig niedrigen Integrationsgrad einerseits und den relativ hohen Speicherbedarf für die

1. “Attempto” war das Motto des Gründers der Universität Tübingen. In Hinblick auf die Funktion steht der Name auch für “A TEsTable Experimental MultiProcessor with FaultTOlerance”.

Systemsoftware andererseits - wurde dieser Versuch jedoch abgebrochen.

Mit der Realisierung des Experimentalsystems in der jetzigen Form wurde 1988 an der J.W.-Goethe-Universität Frankfurt durch den Autor begonnen. Schwerpunkte waren zunächst die Grundinbetriebnahme des Systems (Portierung und Adaption bestehender Softwareteile sowie Programmerweiterungen und -neuerstellungen für die ATTEMPTO-1-Fehlertoleranzfunktionen, Implementation des konzeptspezifischen Kommunikationssystems zur Multicastübertragung von Nachrichten über den parallelen Bus), und die Konzeption und Entwicklung von ATTEMPTO-2-Systemkomponenten (Parallel-Modula [Lu89a], [Lu89b], [Kam89], Parallel-Prolog [Trib89],[Phil89], Werkzeuge zur Konfiguration [Reh90] und zum Debuggen in der verteilten Systemumgebung [Wenz91]), teilweise im Rahmen von Diplomarbeiten. Implementationsunterstützend wurden bereits erste Messungen am Kommunikationssystem vorgenommen sowie einige Konzeptverbesserungen. Als Grundlage für fast alle diese Tätigkeiten war zuerst eine Entwicklungsumgebung selbst zu realisieren, so zum Beispiel zur Systemprogrammierung unter UNIX (dynamisch ladbare Treiber) und für Werkzeuge zur MODULA-2-Programmierung. Im Rahmen einer Diplomarbeit und mit Hilfe eines weiteren wissenschaftlichen Mitarbeiters (befristet) wurde ein verteilter Kommunikationstreiber für den ETHERNET-Konzentrator entwickelt und auf BSD-Sockets basierende Standardkommunikationssoftware portiert [Kn90],[Gü92].

Die Arbeiten am ATTEMPTO- System wurden darauf seit September 1990 an der FAU-Erlangen fortgeführt. Im Hinblick auf ATTEMPTO-1 standen nun folgende Themenkomplexe im Vordergrund:

- Untersuchungen zur Fehlertoleranz-Konzeptvalidierung für ATTEMPTO-1 befassen sich damit, inwieweit die Systemanforderungen: flexible Verwendbarkeit, Portierbarkeit, Erweiterbarkeit, Konfigurierbarkeit, Testbarkeit erfüllt werden konnten und nehmen eine meßtechnische Erfassung und Auswertung von Leistungsdaten vor. Fehlersimulation (und sehr begrenzt Fehlerinjektion) wurden zur Evaluierung bestimmter Fehlertoleranzeigenschaften verwendet, sowie als Grundlage für die Software-Verifizierung und der Bewertung der Testbarkeitseigenschaft benutzt. Die Konzeptvalidierung in Hinblick auf die geforderten Eigenschaften und auch im Vergleich mit existierenden Ferhlertoleranzkonzepten ist wesentlicher Bestandteil der hier vorliegenden Arbeit.
- Im Zusammenhang mit der Erfassung und Beurteilung des Leistungsvermögens spielte die Optimierung des Leistungsverhaltens im System ATTEMPTO-1 eine besondere Rolle. Eine Studienarbeit wurde dazu vergeben [Grill92].
- Die Konzeption und Implementation dezentraler Testkomponenten des Hardware-Diagnose-Systems ist ein weiterer Schwerpunkt, der im Rahmen einer Diplomarbeit bearbeitet wurde [Oppm93. Diese Komponenten sind eigenständige Prozesse auf den lokalen Rechnerknoten, die als “Diagnoseagenten” in Nachrichtenverbindung mit einem zentralen Diagnoserechner stehen und in dessen Auftrag Hardwaretests durchführen. Dies dient der Anbindung an das Diagnose-Expertensystem und schafft die Verbindung zum Teilprojekt ATTEMPTO-2.
- Eine Studienarbeit [Klau92] beschäftigte sich mit der Verbesserung der Bedienbarkeit (Graphik-Benutzeroberfläche). Dies sollte auch zur Erhöhung der Demonstrationsattraktivität beitragen.

Sowohl bzgl. des Diagnosesystems wie auch der Parallelisierungs- und Fehlertoleranzkonzepte von ATTEMPTO-2 wurde eine Verallgemeinerung hin zu einer universellen, nicht auf die ATTEMPTO-1-Hardware begrenzten Systemumgebung angestrebt. Das Diagnoseexpertensy-

stem DIAMONT [Phil91], [Phil89] stellt ein universelles Werkzeug für die wissensbasierte Hardwarediagnose von Multiprozessor- bzw. Mehrrechnerstrukturen im allgemeinen dar. Hardwarestruktur wie auch funktionale Zusammenhänge zwischen den Komponenten sind systemspezifisch modellierbar. Diese Daten sind Bestandteil der jeweiligen Wissensbasis, die auch die speziellen Testregeln enthält. Sowohl logische wie heuristische Inferenzmechanismen werden verwendet, um das Diagnosebild zu fixieren.

Das ATTEMPTO-2- Parallelisierungskonzept wurde ausgehend von einer ursprüglichen Prototyp-Implementation [Kam89] soweit umgestaltet und erweitert, daß die Portierungseigenschaften verbessert sind. Portierungen wurden im einzelnen für ein Netz aus sun-workstations unter sunos4.1.2 und ein Transputersystem unter HELIOS vorgenommen. Dieses Konzept erweitert blockorientierte prozedurale Sprachen um explizite Konstrukte für eine mittelkörnige Parallelität auf der Basis eines RPC-Mechanismus für eine entfernte Dienstbeauftragung. Neben Modula-2 wurde nun eine Implementation mit C als weiterer Basissprache durchgeführt. Hinsichtlich der Fehlertoleranz ging das ursprüngliche Konzept auf in ein sehr flexibles Werkzeug EXFATOS [Trib92] zum expliziten Einsatz von unterschiedlichen Fehlertoleranzverfahren bei punktuellem Redundanzeinsatz innerhalb der durch das Parallelitätskonzept vorgegebenen Client/Server-Hierarchie. Einige Eigenschaften der Parallelität und Fehlertoleranz im ATTEMPTO-2-System und EXFATOS-Konzept sind in Abb. 16 als Überblick aufgelistet.

Parallelität

- Ziel: Verbesserung der Verarbeitungsleistung für ***ein*** Programm
- Laufzeitunterstützung zur Verteilung und Verarbeitung
- Programmierunterstützung zur Expressivität von Parallelität auf hohem Abstraktionsniveau (parallele Bereiche)
- mittelkörnige Granularität (Prozeduren)
- Client/Server - Interaktionen zwischen Envelope- Prozessen (E- Prozeß- Konzept [Lu89a])
- (implizite) asynchrone auftragsorientierte Kommunikationssemantik (“Remote Service Invocation”)
- temporäre hierarchische Client/Server-Strukturen

Fehlertoleranz

- punktuelle Redundanz im Client/Server-Baum
- explizites Einbringen der Fehlertoleranz m.H.v. programmiersprachlichen Konstrukten (z.B. Prozedurattribut “fehlertolerant”)
- wahlweiser Einsatz von statischer oder dynamischer Redundanz
  Serverreplikate zur
  - Fehlererkennung
  - Fehlermaskierung
  - als Ersatzkomponenten für Rekonfiguration nach Rollback zum Punkt der letzten Auftragsvergabe
- automatische Generierung von diversitären Serverversionen zur

Abb. 16 Parallelität und Fehlertoleranz im System ATTEMPTO-2 bzw. in EXFATOS

5.2 Konzeptprämissen

5.2.1 Systemanforderungen und Designmerkmale

Das ATTEMPTO-Konzept zielt auf eine Mehrzweck-Fehlertoleranz im Sinne der in Kapitel 1. aufgeführten Merkmale. Im Vordergrund steht die **variable Systemnutzbarkeit**, die Fehlertoleranz als Alternative zu Systemleistung in einer allgemeinen verteilten Systemumgebung anbietet.

In Verbindung mit der gewünschten Systemvariabilität stehend bzw. darüberhinausgehend fordert das Systemkonzept:

- **Erweiterbarkeit:** die Ausbaufähigkeit des Systems (Skalierbarkeit innerhalb gewisser Grenzen) zum Zwecke der Erhöhung von Zuverlässigkeit und/oder Gesamt- Systemleistung.
- **Transparenz**: Dies klassifiziert Fehlertoleranz als Betriebssystemdienst. Anwenderprogramme müssen sich nicht um die Existenz der Fehlertoleranz kümmern. Die Mechanismen dafür sowie auch deren Verwendung sind im System verborgen.
- **Portabilität**: Die Fehlertoleranz ist software-implementiert. Um ein weites Feld von Anwendungen fehlertolerant ablauffähig zu machen, ist die Kompatibilität mit Standard-Betriebssystemen erforderlich. Anstatt ein existierendes Betriebssystem neu zu schreiben, soll es erweitert werden um Fehlertoleranzfähigkeiten. Damit man flexibel in der Wahl des Basis-Betriebssystems bleibt, ist leichte Portierbarkeit der Fehlertoleranz-Software notwendig.
- **Preisgünstige Hardwarebasis**: auch bei massiver Redundanz von Systemkomponenten soll schon die Grundausstattung für fehlertoleranten Betrieb kostengünstig sein, um die Attraktivität der Fehlertoleranz auch für neue Anwendungsgebiete zu erschließen.
- **Testbarkeit**: die Architektur des Software-Systems für Fehlertoleranz soll eine leichte Verifizierbarkeit seiner Implementation unterstützen. Eine leichte Überschaubarkeit und Nachvollziehbarkeit der Grundmechanismen ist geeignet, das Vertrauen der Anwender in die Funktionsfähigkeit des für Fehlertoleranz erweiterten Systems zu erhärten. Diese Eigenschaft ist bei fehlertoleranten Systemen besonders wichtig.
- **Konfigurierbarkeit**: einzelne Bestandteile des Softwaresystems sollen leicht austauschbar sein, um unterschiedlichen Anforderungen der jeweiligen Systembasis gerecht zu werden. Neben der Installation für den Betrieb ist dies vor allem auch wegen der geplanten Verwendung des ATTEMPTO-Systems als Experimentalsystem interessant.
- **Dezentralisierung** aller Systemfunktionen und Fehlertoleranzmechanismen zur Vermeidung von Fehlertoleranz-Schwachstellen.

Das experimentelle ATTEMPTO-System ist als Arbeitsplatzrechner für einen Einzelnutzer gedacht. Gerade in diesem Einsatzgebiet der interaktiven Allzweck-Anwendung ist nach unserer Meinung die Einfachheit im Gebrauch von herausragender Bedeutung, sogar dann, wenn dies merklich auf Kosten der Performanz geht. Diese Eigenschaft der einfachen Handhabbarkeit der Fehlertoleranz resultiert im wesentlichen aus der oben geforderten Transparenz. Für Erweiterbarkeit, Portabilität, Konfigurierbarkeit und Testbarkeit ist wie in Kapitel 1. beschrieben ursächlich Modularität verantwortlich, sowohl hinsichtlich der Hardwarekomponenten, wie auch im Sinne von funktionaler Modularität der Softwarearchitektur. Die Dezentralisierung ist ein wichtiges Systemprinzip, das es erlaubt, auf teure Maßnahmen zur Zuverlässigkeitsabsiche-

rung von nur einfach im System vorhandenen Funktionseinheiten (*Single Point of Failure*) zu verzichten, und stattdessen Hardware “von der Stange” zu verwenden und trotzdem hohe Systemzuverlässigkeiten zu erzielen. Für die Betriebssoftware bedeutet Dezentralisierung, daß zentralisierte Datenhaltung vermieden werden muß, und Fehlertoleranzmechanismen, wie z.B. die Maskierung, auf verteilten Algorithmen beruhen.

5.2.2 Parallelität und Fehlertoleranz im System; ein Überblick

Die Nutzung eines Multiprozessor- Systems für Parallelität ist auf verschieden Ebenen möglich, die die Körnung der parallel abzuarbeitenden Programmfunktionen bzw. -teile bezeichnen, so z.B. auf Prozeß-, Prozedur- oder Befehlsebene. In einem lose, d.h. über Nachrichtenaustausch gekoppelten System aus mehreren Rechnerknoten, wie wir es verwenden, empfiehlt sich eine grob- oder bestenfalls noch mittelkörnige Parallelität, um die anteiligen Kosten der Kommunikation gering zu halten. Vor allem, weil beim ATTEMPTO-Experimentalsystem hauptsächlich der Fehlertoleranzaspekt im Blickwinkel unseres Interesses liegt, und die replizierten programmtechnischen Grundeinheiten Benutzer-Jobs (Prozeßgruppen) sind, beschränken wir die Parallelität auf eben diese Jobebene. Auf diese Weise wird das Gesamtangebot an Systemleistung erhöht, die Leistungssteigerung für ein einziges Programm allerdings ist nicht möglich. Dieser Aspekt der parallelen Abarbeitung eines einzelnen Programms wird im zweiten Teil des ATTEMPTO-Projekts (siehe Projektüberblick Kapitel 5.1) gesondert berücksichtigt. Eine weiterführende Konzequenz dieser Festlegung auf eine grobkörnige Parallelität ist, daß eine Anwender-Interprozeß-Kommunikation über Prozessorgrenzen hinaus nicht vorgesehen wird: Jobs bleiben lokal begrenzt. Die Absicht ist es, vorhandene (Standard-) Programme einer Monoprozessor-Umgebung auf einer Multiprozessorumgebung fehlertolerant auszuführen, falls dies vom Benutzer gewünscht wird. Auf diese Weise laufen im System entsprechend der Benutzervorgabe an den Fehlertoleranzgrad und dem vorhandenen Reservoir an Rechnereinheiten mehrere fehlertolerante und/oder einfache Programme nebeneinander ab. Die Aufteilung der einzelnen Jobs und gegebenfalls auch deren Replizierung für Fehleroleranz auf unterschiedliche Knoten im System wird automatisch vom Betriebssystem vorgenommen. Der dem Job-Dispatching zugrundeliegende Mechanismus ist ***Selbstattraktion***; darauf wird im Kapitel 5.6.2 näher eingegangen. Er beruht vollständig auf verteilten Algorithmen.

In Bezug auf Fehlertoleranz kommt Fehlerunterdrückung im Sinne einer N-Modular-Redundanz durch unmittelbare verteilte Systemdiagnose zum Einsatz, die auf Vergleichstestmodellen beruht. Die Maskierungsschnittstelle liegt dabei an den Systemgrenzen zu Umwelt, also nicht etwa z.B. im Kommunikationssystem als Schnittstelle zur Prozeßinteraktion, sondern an physikalischen Schnittstellen zum Anschluß externer Geräte. Im Falle des Experimentalsystems ist dies eine serielle asynchrone Datenleitung zum Anschluß eines Datenterminals. Da auch die Synchronisationspunkte zum Vergleich der Aktivitäten der replizierten Programme dort liegen, wo das Programm mit der Ausgabe der Ergebnisdaten die vom Benutzer geforderte Diensterbringung des Rechnens vollendet, also Job-Ergebnisse an die Umwelt gelangen, wird der Maskierungsmechanismus ***“End-to-End Job Result Comparison”*** genannt. Der Term “End-to-End” kennzeichnet in erster Linie die “Über-Alles”-Charakteristik der Fehlermaskierung. Gleichzeitig wird aber auch damit zum Ausdruck gebracht, daß die Fehlermaskierung ein Betriebssystemdienst auf höherer Ebene ist[1].

1. Die Schichtenstruktur des ATTEMPTO-Betriebssystems wird in Kapitel 5.3.2 erläutert.

Parallelität

- Ziel: Verbesserung des Gesamtangebotes an Systemleistung
- grobe Granularität (Jobs = Prozessgruppen)
- benutzertransparentes Job-Dispatching durch “Selbstattraktion”

- **keine** Kommunikation zwischen Jobs
  -> Verteilungscharakter nur auf Betriebssystemebene

Fehlertoleranz

- variabler Fehlertoleranzgrad
- benutzertransparente Fehlertoleranzmechanismen

- Fehlerunterdrückung (“End-to-End “- Strategie)
  - mit wählbarem Redundanzaufwand (N-modular)
  - basierend auf unmittelbarer verteilter Systemdiagnose (“*Job Result Comparison”*)
  - Vorteile:
    - einfaches Fehlermodell
    - hohe Fehlerüberdeckung
    - auch (unabhängige) Mehrfachfehler tolerierbar

aber z.B
- **keine** dynamische Rekonfiguration
- **keine** On-Line- Reparierbarkeit

Abb. 17 Eigenschaften der Parallelität und Fehlertoleranz im ATTEMPTO-System

Bei der statischen Fehlermaskierung ist die Fehlerbehandlung relativ einfach; nach erfolgter Fehlerdiagnose findet z.B. keine dynamische Rekonfiguration statt. Stattdessen werden fehlerhafte Komponenten nur ausgegrenzt, vor allem, um ihre Teilnahme an neuen Jobs zu verhindern. Der maximale Parallelitäts- und Fehlertoleranzgrad degradiert also bei sukzessiven Fehlern zusehends, bis eine Off-Line-Reparatur vorgenommen werden muß. Um die Systemverfügbarkeit zu erhöhen, wird versucht, die Reparaturzeit kurz zu halten, indem Werkzeuge zur Diagnoseunterstützung zur Verfügung gestellt werden. Das Diagnoseexpertensystem DIAMONT [Phil91] führt eigenständig Tests durch und benutzt auch die zuvor on-line bereits gewonnenen Diagnoseresultate, die in sogenannten Fehlerfrequenzlisten während des Betriebs festgehalten werden. Kapitel 5.7 wird noch in besonderer Weise auf das Fehlermodell und die Fehlerbehandlung eingehen. DIAMONT ist jedoch nicht Gegenstand dieser Arbeit.

Die End-to-End-Maskierung gestattet ein einfaches Fehlermodell und führt naturgemäß zu einer hohen Fehlerüberdeckung; solange die Fehler unabhängig und symptomverschieden sind, sind differenzierte Fehlerfallberücksichtigungen nicht notwendig. Der variable Redundanzgrad ermöglicht auf einfache Weise die Tolerierung von Mehrfachfehlern.

5.3 Systemarchitektur

5.3.1 Hardware

Die Systemarchitektur von ATTEMPTO ist die eines homogenen autonomen Systemverbundes [Ne88], bestehend aus lose gekoppelten, eigenständig arbeitenden Rechnereinheiten, die nur zum Zwecke der Fehlertoleranz und der Verwaltung globaler Ressourcen miteinander kooperieren. Die Hardware des Systems ähnelt von daher der eines allgemeinen verteilten Systems. Die Flexibilität des Fehlertoleranzkonzeptes beruht auf einer Softwareimplementation, und dies gestattet die

Verwendung von Standardkomponenten.
Die gewünschte preisgünstige Hardwarebasis wird gebildet von Single-Board-Computern (SBC's) für einen weitverbreiteten Standard-Multiprozessorbus. Der VMEbus ist vor allem im Bereich der Prozeßautomatisierung auf der Ebene der Leit- und Prozeßsteuerung verbreitet und auch bei Entwicklungs- und kommerziellen Allzwecksystemen zu finden, sodaß das Angebot an verhältnismäßig hochintegrierten aber leistungsfähigen Computerkarten recht groß und dadurch auch preislich genügend attraktiv ist, um damit selbst bei einer Verdreifachung entsprechend unserem Konzept der Fehlermaskierung noch zu einer kostengünstigen Grundversion für den Fehlertoleranzbetrieb zu kommen.

Die modulare Erweiterbarkeit des Systems ist begrenzt durch die

Maximale Knotenzahl.
N dieser gleichartigen Systemgrundkomponenten, im weiteren Einheiten oder Knoten genannt, erweitern das System zu einem N-modular redundanten. Die Maximalzahl N ist beschränkt einerseits durch praktische Hardwarevoraussetzungen - hier vor allem durch die Möglichkeit der Ankopplung an das globale Kommunikationsmedium - , andererseits auch durch die zur Verfügung stehende maximale Kommunikationsleistung dieses Mediums. Der angestrebte Verwendungszweck des Experimentalsystems als Einbenutzer-Workstation zur Parallelbearbeitung von Benutzerjobs verlangt auch nicht mehr als nur moderate Parallelität im System, um überhaupt noch nutzbar zu sein. Eine Obergrenze der Erweiterbarkeit ist bei etwa N=16 zu sehen; im realisierten System ist N=7, wobei z.Z. nur n=5 Knoten vorhanden sind[1]. Werden alle Knoten im System für den Fehlertoleranzbetrieb herangezogen, sind sehr hohe Zuverlässigkeiten erzielbar. Der relative Zuverlässigkeitsgewinn nimmt mit steigendem Redundanzeinsatz jedoch ab, wie in Kapitel 6.4 gezeigt wird, sodaß auch aus diesem Grund eine Einschränkung der maximalen Knotenzahl sinnvoll ist.

Entsprechend dem geforderten Dezentralisierungsprinzip kommt der

Knotenautonomie
besondere Bedeutung zu. Die Knoten arbeiten plesiochron, d.h. mit derselben nominalen Taktfrequenz, aber aus lokalen Taktgeneratoren. So gibt es z.B. auch keine globale Zeit im System, es lassen sich jedoch Aussagen über eine maximale Driftrate der fehlerfreien lokalen Timer machen. Die Knotenprozessoren besitzen ausschließlich lokale Arbeitsspeicher und steuern ihre privaten Peripheriekomponenten. Der Aufbau des verwendeten Single-Board-Computers

1. Das Betriebssystem adaptiert sich während der Initialisierungsphase des Fehlertoleranzbetriebs automatisch an die zu diesem Zeitpunkt reale Knotenzahl n.

ist schematisch in Abb. 19 dargestellt. An die Peripheriekomponenten können Geräte im lokalen Zuständigkeitsbereich oder aber solche mit Systembedeutung, sogenannte ***Globale Ressourcen***, angeschlossen sein, siehe dazu Abb. 18. Bei diesen handelt es sich um Komponenten der Systemumgebung, bezüglich deren Ausgabe-Schnittstellen eine Fehlermaskierung durchgeführt werden soll, oder aber deren Eingabedaten zu allen Knoten zugeführt werden müssen. Im lokalen Zuständigkeitsbereich liegen z.B. die Konsolschnittstelle - eine serielle Leitung - und Platten. Um ein fehlertolerantes Dateisystem zu realisieren, können jedoch auch globale Platten vorhanden sein. Im gegenwärtigen Experimentalsystem gibt es eine einzige globale Ressource, eine asynchrone Datenleitung hin zu dem Benutzerterminal. Globale Ressourcen werden über einen Bus angeschlossen, der durch Parallelschaltung der Ein- und Ausgangsleitungen aller sie betreffenden lokalen Schnittstellen entsteht. Zur Wahrung der physikalischen Integrität sind die Ausgangsleitungen in der Regel geeignet zu entkoppeln. Im Falle der asynchronen Terminalleitung geschieht dies durch ein geringkomplexes Netzwerk aus passiven Komponenten (Dioden und Widerstände). Abgesehen von der Parallelschaltung des globalen Ressourcen-Busses sind keine weiteren Änderungen der Standard-Hardware erforderlich, um fehlertoleranten Betrieb zu ermöglichen.

Von ebenfalls globaler Bedeutung ist die Kommunikationsschnittstelle. Das

Kommunikations-Medium
ist der Standard-Parallel-Bus, über den Nachrichten hin zu knotenlokalen *Mailboxen* versendet werden. Diese liegen in einem *Dual-Ported RAM*, auf das der Zugriff sowohl lokal wie vom globalen VMEbus aus möglich ist. Jeder Knoten besitzt die Befähigung zum Busmaster, wobei bei konkurrierendem Buszugriff der VMEbus-Systemarbitrierungsmechanismus zum Tragen kommt. Der Nachrichtentransfer ist interruptgesteuert, dazu ist jedem Knoten eine separate Interuptleitung zugeordnet. Der VMEbus sieht 7 Interruptleitungen vor, die im System ATTEMPTO auch verwendet werden, sodaß hieraus die Beschränkung der Knotenzahl auf N=7 folgt. Für höhere Knotenzahl könnten parallele programmierbare Ein/Ausgangsleitungen verwendet werden, falls die Hardware solche besitzt. Die Grundforderung besteht im Vorhandensein eines alle Eingangsleitungen geschlossen behandelnden Interrupt-Controllers auf den SBCs, was in unserem Fall durch den VMEbus-Interrupt-Handler gegeben ist. Die üblicherweise auf dem VMEbus vorgesehene Betriebsart mit vektorisiertem Interrupt-Bestätigungszyklus wird nicht verwendet. Ein software-implementiertes Kommunikations-Protokoll realisiert einen *Atomaren Rundspruch*, wie in Kapitel 5.4 näher beschrieben wird.

Im Grunde wäre jedes Mehrrechnersystem, das aus örtlich getrennten vernetzten Einzelrechnern aufgebaut ist, und dessen Kommunikationssystem über die Möglichkeit zum atomaren Rundspruch verfügt, als Hardwarebasis für das ATTEMPTO-System einsetzbar. Die örtliche Trennung würde sogar erhöhte Fehlerisolation bewirken, ebenso wie z.B. die Verwendung von seriellen Broadcastmedien, die ohne galvanische Kopplung auskommen (z.B. ETHERNET oder FDDI). Demgegenüber zeichnet sich die gewählte ATTEMPTO-Hardware durch kompakten Aufbau und sehr preisgünstige Realisierung aus. Letzteres resultiert auch aus dem Wegfall aufwendiger Netzkomponenten. Die SBC’s sind Doppeleuropakarten und in einem handelsüblichen 19-Zoll-Gehäuse zusammengefaßt. Die äußeren Abmessungen sind hauptsächlich bestimmt durch den Platzbedarf für die Plattenlaufwerke (je ein 5 1/4 - Zoll-Winchesterlaufwerk pro Knoten). Die VMEbus-Norm verlangt eine zentrale Spannungsversorgung. Um unempfindlich gegenüber Defekten dieser Fehlertoleranzschwachstelle zu sein, kann eine Notspeisung (durch Doppelung der Spannungsversorgung) vorgesehen werden.

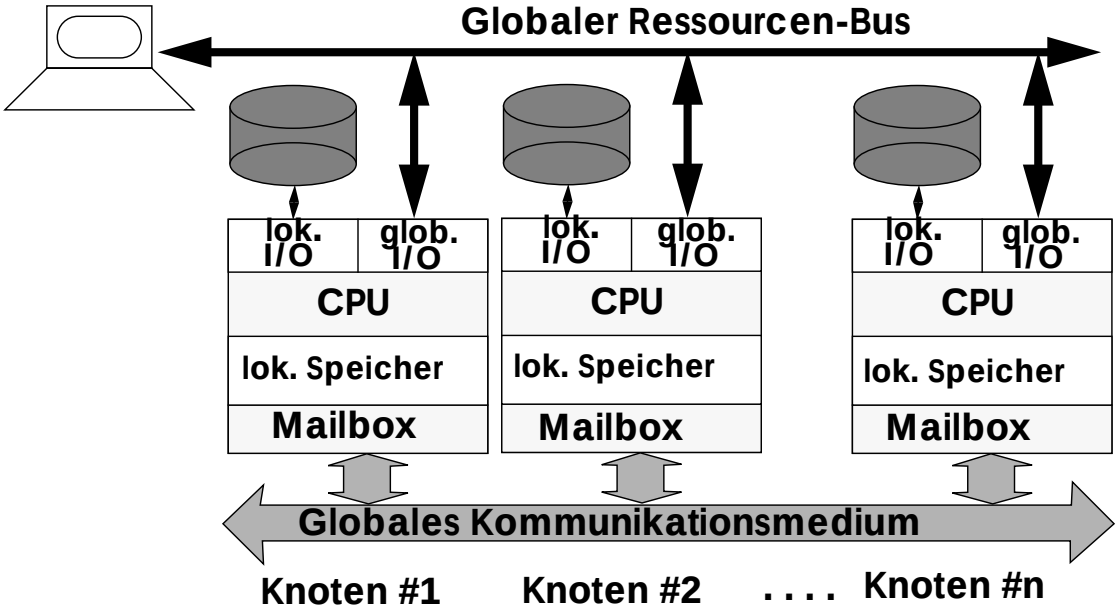


Abb. 18 Hardware-Aufbau des ATTEMPTO-Systems

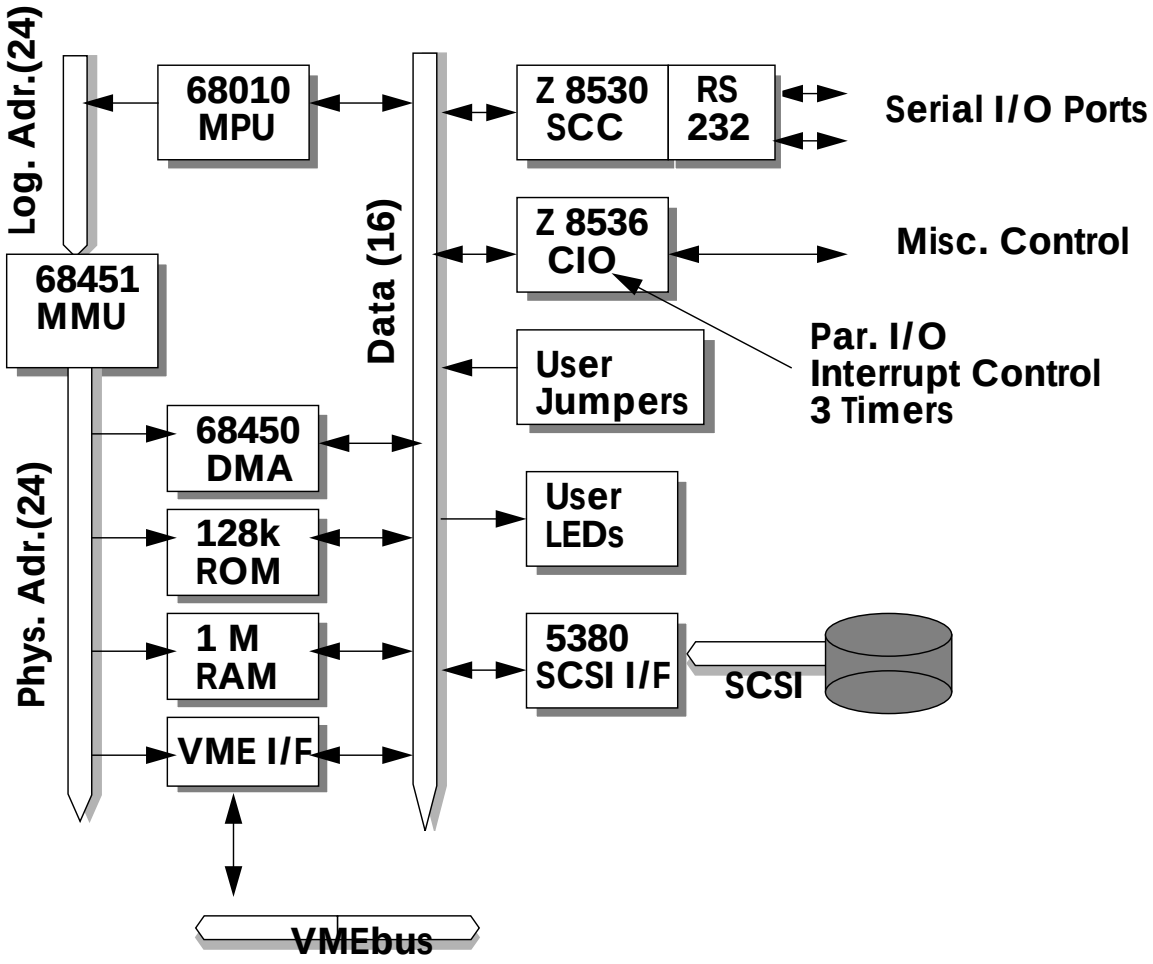


Abb. 19 Single- Bord- Computer Heurikon HK68/V10, Blockdiagramm

5.3.2 Betriebssystemstruktur

Modularität und Erweiterbarkeit wird nicht nur für die Hardware verlangt, sondern auch für die Teile des Betriebssystems, die die Fehlertoleranz realisieren. Entsprechend unserer Vorgabe, als Ausgangsbasis ein austauschbares Standard-Betriebssystem zu wählen, und die Fehlertoleranzfunktionen als ergänzenden Zusatz dazu zu konzipieren, ist es von besonderer Bedeutung, die Mechanismen für Fehlertoleranz in einer abgesetzten eigenständigen Schicht zu isolieren, die mit dem Rest des Betriebssystems sowenig Verbindungen wie möglich hat. Diese Fehlertoleranzschicht (***Fault Tolerance Layer FTL***), sitzt, wie in Abb. 20 dargestellt, auf dem lokalen Wirtsbetriebssystem (***Local Operating System LOS***) auf, und trennt es so von der Anwenderschicht. Die resultierende Betriebssystemarchitektur ähnelt der der bekannten *Newcastle Connection* [BMR82], die aufgebaut wurde, um Funktionen eines verteilten Betriebssystems (ein globales verteiltes Dateisystem) in einem Netzwerk von Rechnern zur Verfügung zu stellen, auf denen - wie in unserem Fall - ein erweitertes Standardbetriebssystem läuft. Ähnlich wie bei ATTEMPTO gibt es dort eine Zwischenschicht, Verbundschicht (*Connection Layer*) genannt, zwischen Betriebssystem-Kern und Anwenderprogram, das die lokalen Knoten zur Kooperation zusammenfaßt. Für Transparenz der neuen Systemfunktionen - Fehlertoleranz in unserem Fall - ist dadurch gesorgt, daß die Anwenderschnittstelle zum Betriebssystem (in Abb. 20 mit **S** bezeichnet) unverändert bleibt, d.h. derselbe Satz von Systemaufrufen bleibt für Anwenderprogramme verfügbar (**S=S'**), ihre Realisierung ist jedoch - verdeckt für den Anwender - möglicherweise verändert.

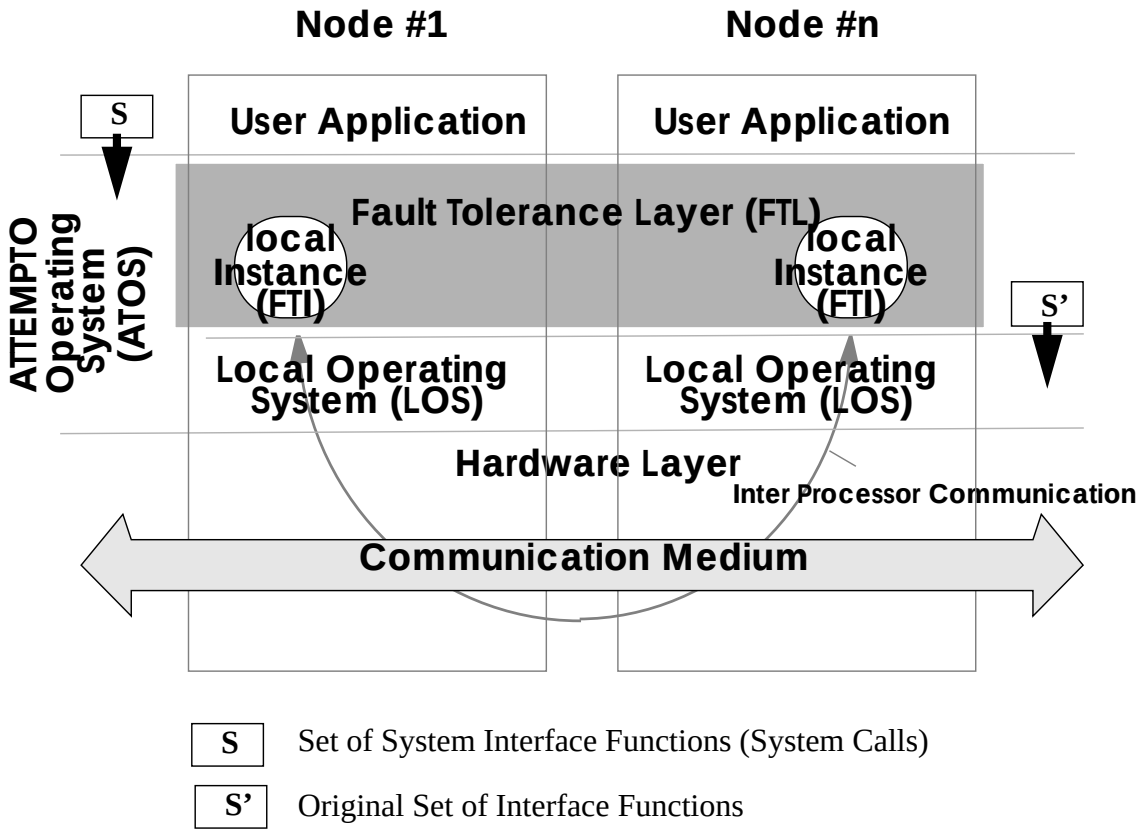


Abb. 20 Schichtenstruktur des Verteilten Betriebssystems für Fehlertoleranz

Anders als bei der Newcastle Connection, wo Transparenz auf die Quellcode-Ebene eines Pro-

gramms beschränkt bleibt, gewährt das ATTEMPTO-System Transparenz auf der Ebene von Binärprogrammen, und bleibt damit unabhängig von Programmiersprachen. Fertige, in der Monoprozessorumgebung ausführbare Programme sind unverändert ohne Rekompilation und auch ohne neuerlichen Linklauf in der Multiprozessorumgebung fehlertolerant ablauffähig. Dies scheint eine einzigartige Eigenschaft des software-implementierten ATTEMPTO-Systems zu sein. Sie wird dadurch möglich, daß das Benuterprogramm kernelseitig an seiner Systemschnittstelle adaptiert wird, anstelle, wie bei der Newcastle Connection, auf der Anwenderseite. Die Anpassungen für den erweiterten Betriebssystemdienst werden also nicht in einer erweiterten Programmlaufzeitbibliothek vorgenommen; stattdessen besteht die Fehlertoleranzschicht aus einer Reihe von Systemprozessen, die den fehlertolerant ablaufenden Benutzerjob auf Systemaktivitäten hin überwachen. Tatsächlich sind diese Systemprozesse hochpriore Prozesse, die außerhalb des Kerns auf der Anwenderebene ablaufen. Der Betriebssystemkern ist nur äußerst begrenzt modifiziert, da, wo es nötig ist, um die Kontrolle über den Benutzerjob zu den Systemprozessen zu transferieren. Die Fehlertoleranzmechanismen außerhalb des Betriebssystemkerns zu realisieren, schafft einen Rahmen für die Software-Modul-Konfiguration mit größtmöglichem Freiheitsgrad und erleichtert die Portierbarkeit des gesamten Softwaresystems für Fehlertoleranz.

Die Fehlertoleranzschicht FTL ist über das System verteilt. Sie besteht aus lokalen ***Fehlertoleranzinstanzen (FTI)*** auf jedem Rechnerknoten, die durch Nachrichtenaustausch miteinander kommunizieren. Der zuverlässige Nachrichtentransport zwischen den Knoten (*Inter Processor Communication*) ist ein Systemdienst, der zum jeweiligen LOS hinzugefügt ist. Beide, LOS und FTL, formieren zusammen das ***ATTEMPTO Operating System ATOS*** [RBDDL84].

Eine lokale FTI selbst ist wiederum in einzelne Prozesse unterteilt. Zunächst besteht sie aus mehreren Schwergewichtsprozessen. Diese sind Verwaltungseinheiten des lokalen Betriebssystems. Abb. 21 zeigt die lokale Prozeßkonstellation zusammen mit den Kommunikationswegen. Es ist zu unterscheiden zwischen einer lokalen Interprozeß-Kommunikationsschicht im LOS, die die Prozesse der lokalen Fehlertoleranz untereinander verbindet, und der globalen Kommunikationschicht. Die letztere verbindet die Prozessoren untereinander und hat mit den Systemschnittstellen zu tun, an denen die globalen Ressourcen angeschlossen sind.

Die globale Kommunikationsschicht besteht aus den Gerätetreibern /dev/ph (port handler device) und /dev/tty (terminal device) im LOS der einzelnen Rechnerknoten. Die von diesen kommenden Eingabedaten werden ausschließlich von jeweils eigens dafür vorgesehenen *Handler-Prozessen* verwaltet. Ein weiterer Handler-Prozeß, der *Kernel Handler*, überwacht die Interaktionen zwischen Benutzerjob und lokalem Betiebssystem. Der *System Call Handler* im LOS-Kern wurde modifiziert, um Systemaufrufe von Anwenderprozessen, die zum Benutzerjob gehören, zu einem Pseudogerätetreiber im LOS umzuleiten (communication instance kernel /dev/cik). Dieser steht unter Kontrolle des Kernel-Handler-Prozesses. Device und Handler sind durch ein eigenes Nachrichten-Kommunikationssystem miteinander verbunden, das zur Übermittlung von Status- und Kommandoinformation und zur Übertragung von Zusatzdaten benutzt wird, die den Systemaufruf begleiten. Auf diese Weise ist es möglich, Systemaufrufe von Anwenderprozessen auszufiltern und außerhalb des Betriebssystems einer Sonderbehandlung zuzuführen. Auf diesen Mechanismus geht Kapitel 5.5 ausführlich ein.

Die Handler-Prozesse kommunizieren mit der eigentlichen lokalen Fehlertoleranz-Zentrale, dem lokalen FTI-Prozeß, über eine einzige gemeinsame Pipe (*Handler* Pipe), die das LOS dafür bereitstellt. Die Semantik der Pipe-Kommunikation garantiert atomare Schreiboperationen,

wodurch die Integrität einzelner Nachrichten als zusammenhängende strukturierte Datenmenge sichergestellt ist. Für alle Handler-Prozesse ist unidirektionale Kommunikation typisch. Deshalb ist es sehr leicht, ihre Funktion zu verifizieren, indem ihr Ein/Ausgabeverhalten beobachtet wird.

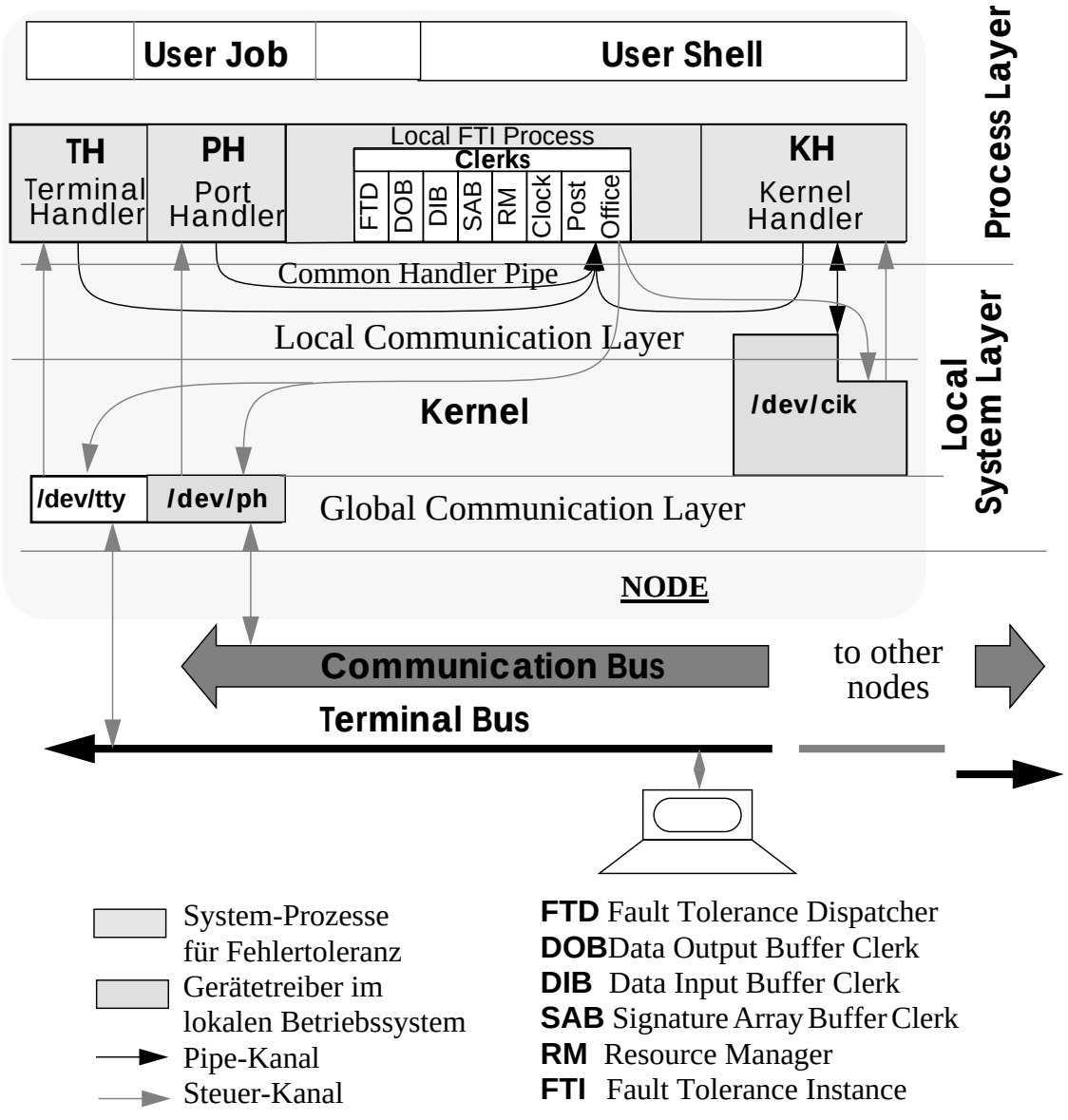


Abb. 21 Lokale Prozeßkonstellation und Kommunikationsbeziehungen

Es gibt allerdings noch einen weiteren Grund, der die Kapselung der Handler-Funktionen in eigenständigen Schwergewichtsprozessen erzwingen kann: möglicherweise stellt das Wirtsbetriebssystem keine Möglichkeit für Prozesse zur Verfügung, im suspendierten Zustand (Schlafen) durch eines von mehreren äußeren Ereignissen auf Ein/Ausgabe-Kanälen aufgeweckt zu werden (*passives Polling*). Dann müssen die Handler-Prozesse als Stellvertreter für den lokalen Fehlertoleranz-Hauptprozeß fungieren, die nun suspendiert werden können zum Warten auf ein einziges Ereignis, ohne die übrigen Funktionen zu behindern. Ansonsten müßte der Hauptprozeß die einzelnen Kanäle zyklisch abfragen (*aktives Polling*), was jedoch erhebliche Rechenleistung verbraucht. Tatsächlich gestattet das Basisbetriebssystem des ATTEMPTO-

Experimentalsystems, ein UNIX-SYSTEM V-Derivat, passives Polling nicht[1], sodaß wir auf die Handlerprozesse als Stellvertreter angewiesen sind. Da wir jedoch geringstmögliche Anforderungen an den Leistungsumfang eines Wirtsbetriebssystems stellen wollen, um die Portierbarkeit nicht zu gefährden, ist eine solche Strukturierung der Schwergewichtsprozeß-Landschaft ohnehin ratsam. Damit sind allerdings Einbußen im Leistungsverhalten verbunden, wie später noch gezeigt werden wird.

Der FTI-Prozeß ist ein zentraler Koordinator der lokalen Aktivitäten auf einem lokalen Rechnerknoten. Entsprechend des Prinzips der funktionellen Modularität besteht dieser Prozeß wiederum aus Teilprozessen, die *Clerks* genannt werden. Diese sind Leichtgewichtsprozesse, Aktivitätsträger für Programmmodule, die in einem gemeinsamen Adreßraum ablaufen. Sie werden in einem eigenständigen, d.h. vom LOS unabhängigen Prozeßmanagement (*dispatch*ing) verwaltet, dessen Funktionen für die Prozessorvergabe auf expliziten Kontrolltransfers beruhen. Die Prozeßverwaltungsfunktionen sind für den (System-)Programmierer im lokalen Kommunikationssystem verborgen. Ein besonderer Systemdienst, das *Post Office*, enthält in sich den Dispatcher. Es besteht aus zwei einzelnen Clerks, die sich getrennt um das Versenden und Empfangen von Nachrichten kümmern. Allgemein befaßt das Post Office sich mit dem Botschaftsverkehr sowohl zwischen den Leichtgewichtsprozessen als auch empfangsseitig zwischen Handlerprozessen und den Fehlertoleranz- Clerks (Abb. 22). Ein einheitliches Botschaftsformat wird verwendet, unabhängig davon, ob Botschaften zu Clerk-Mailboxen oder auf der Handler-Pipe verschickt werden. Das Postoffice stellt sehr vorteilhaft eine zentrale Schnittstelle dar, an der für Debug- und Überwachungszwecke der gesamte lokale Botschaftsverkehr abgelauscht werden kann (*message logger*).

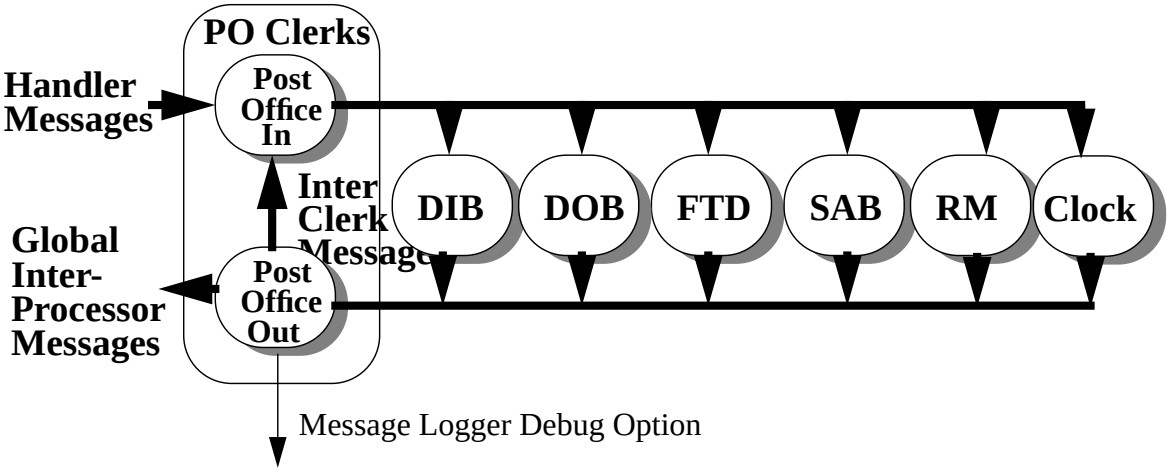


Abb. 22 Kommunizierende Clerks

Die Clerks sind Spezialisten für Teilaufgaben. Sie sind befaßt mit

- der Administration von Eingaben und Ausgaben von/zu globalen Ressourcen (DIB: *Data Input Buffer* Clerk, bzw. DOB: *Data Output Buffer* Clerk),
- der Berechnung und dem Vergleich von Signaturen (SAB: *Signature Array Buffer* Clerk) für eine verteilte Fehlerdiagnose, die auf einem Vergleichstestverfahren beruht, und die Grundlage für die Fehlermaskierung ist,

1. Etwa durch einen *select()*-Systemaufruf, wie er von BSD-UNIX her bekannt ist.

- die Aufteilung von Jobs auf Rechnerknoten (FTD: *Fault Tolerance Dispatcher* Clerk),
- der Zuteilung globaler Ressourcen (RM: *Resource Manager* Clerk), um Ausgabe-Konflikte aufzulösen, und mit
- Zeitüberwachungsaufgaben (*Clock* Clerk).

5.3.3 Modularisierung im Softwaresystem für Fehlertoleranz

Die strukturelle Modularität geht konform mit der funktionalen Modularität wegen der Kapselung von Systemprozessen in Programme, die als separate LOS-Prozesse oder als Leichtgewichtsprozesse ablaufen. Im Hinblick auf die letzteren wird dies durch die Wahl einer Hochsprache (MODULA-2 [Wi88]) begünstigt, die eine modulare Programmstruktur erlaubt und gleichzeitig programmiersprachliche Möglichkeiten zum Ausdruck von Nebenläufigkeit aufweist (Coroutinen-Kontroll-Transfer), mit deren Hilfe Sub-Prozesse kreiert werden können.

Die Fehlertoleranzschicht von ATOS ist hierarchisch aufgebaut. Abb. 23 zeigt die Strukturierung in vier Schichten. Während sich in den obersten beiden Schichten die Prozeßstruktur widerspiegelt, sind die unteren prozedurorientiert. Sie stellen Dienstleistungen zur Verfügung, auf die die oberen Schichten aufbauen. Die Schicht 4 enthält die Schwergewichtsprozesse. Neben den Programm-Modulen der Handler-Prozesse und des lokalen FTI-Prozesses gibt es hier noch einen Initial-Prozeß (FTLinit), der die Installation des Programmsystems vornimmt, indem er Kindprozesse abspaltet, die durch die Fehlertoleranz-Systemprozesse überlagert werden. Der lokale Fehlertoleranzinstanz-Prozeß wird gebildet von den Leichtgewichtsprozessen der Schicht 3. Das sind die Module, die neben privaten Datenstrukturen noch die Clerks als Aktivitätsträger enthalten. Alle Prozeßmodule zeichnen sich dadurch aus, daß ihre Datenstrukturen nach außen verborgen sind, sodaß Fremdzugriffe auf sie über Modulgrenzen hinweg nicht möglich sind. Obwohl die Clerks der Schicht 3 in einem gemeinsamen Adreßraum zusammengefaßt sind, wird auf Querreferenzierung von Daten verzichtet. Modulinteraktionen beruhen ausschließlich auf dem Austausch von Botschaften. Die Kommunikation wird unterstützt durch die Prozesse des Post-Office und dem weiteren Dienstprozeß *Clock*, der Funktionen der Zeitverwaltung zur Verfügung stellt. Das Prinzip der ***objektorientierten Datenkapselung*** erleichtert die Modul-Rekonfigurierbarkeit und hilft zusammen mit der hierarchischen Strukturierung von Programmobjekten, den Nachweis der Korrektheit des Programmsystems zu vereinfachen. Die Objekthierarchie legt die Importbeziehungen zwischen Objekten fest: höhere Schichten können auf Objekte niedrigerer zugreifen, umgekehrt ist dies nicht möglich.

Die unteren beiden Schichten fassen Dienstleistungensprozeduren in Bibliotheken zusammen. Die Schicht 2 enthält Dienstleistungen zur Unterstützung des Fehlertoleranzbetriebs. Soweit möglich, wurde auch hier das Prinzip der Datenkapselung angewendet. Eine Dreiteilung ist in Abb. 23 erkennbar. *CImain, CIKernel* sind Module, die Kommunikationsprimitiven für das Kernel-Device und verschiedene andere Prozeduren zur Interaktion mit dem LOS zusammenfassen. *JCB* (ein abstrakter Datentyp) enthält Operationen auf Job-Kontrollblöcken, *Diagnose* kapselt den Diagnosealgorithmus und ist so für Experimente leicht austauschbar. Neben diesen spezifischen Funktionen gibt es noch eine Modulklasse, die die Steuerung des Leichtgewicht-Prozeß-Systems und den Nachrichtentransport unterstützt. Das Modul *Messages* definiert Operationen zur Erzeugung von Botschaften und Mailboxen und dem Verschicken bzw. Empfang von Nachrichten. Mit der Botschaftsübermittlung ist ein impliziter Kontroll-Transfer zwischen Clerks verbunden. Solche Prozeß-Steuerungsfunktionen sind im Modul *iProcesses* implementiert. Die dritte Klasse innerhalb der Schicht2 enthält Dateimodule, die häufig verwendete Datentypen, Konstanten oder globale Variablen der Allgemeinheit zur Verfügung stellen.

	PROZESSORIENTIERT		
4	Schwergewichts-Prozesse	Programm-Module	~/uprocs: FTI, KH, TH, PH, FTLinit
3	Leichtgewichts-Prozesse	FT-Clerks Kommunikations-Unterstützung	~/mprocs: DIB, DOB, SAB, FTD,RM POin, POout, Clock
	PROZEDURORIENTIERT		
2	FT-Dienstleistungen	Spezifische Funktionen Systemsteuerung Allgemeine Programmobjekte	~/attlib: CImain, CIKernel, JCB, Diagnose, ... Messages, iProcesses FTLbase, TypVar, MsgTypes
1	Systemschicht	Allgemeine System-Dienstleistungen System-Anpassungen	~/lib: Lists, Strings, Terminal, ... SystemType, SysCalls, ...

Abb. 23 Schichtenhierarchie des Fehlertoleranz-Software-Systems

Die unterste Schicht schließlich nimmt eine Anpassung an das lokale Betriebssystem vor und bietet allgemeine Dienstleistungen der MODULA-Programmlaufzeitumgebung.

Schon während der System-Implementation wurde deutlich, das diese Modularitätsstruktur tatsächlich die Testbarkeit und Portierbarkeit unterstützen kann. Nur bei Modulen, die direkt mit dem Standardkern interagieren, wurden Schwierigkeiten beobachtet. Hier gehen die Besonderheiten des jeweiligen Betriebssystems ein. Für den Bereich der Systemprogrammierung, der sich mit den Kernmodifikationen und -erweiterungen befaßt, ist eine tiefergreifende Kenntnis der Betriebssystemmechanismen vonnöten, und die Vorgaben des jeweiligen Betriebssystems an die Programmierumgebung sind zu beachten. Dieser Teil der Fehlertoleranz-Erweiterungen ist deshalb in einem eigenen Programmsystem isoliert, das in der Programmiersprache C und zu einem geringen Teil in Assembler verfaßt ist. Um die Portabilität der Handler-Module im soeben beschriebenen MODULA-Softwaresystem zu erleichtern, sind die Schnittstellen zum Betriebssystem in leicht austauschbaren Low-Level-Modulen separiert.

5.3.4 Allgemeine Programmstruktur eines Clerks

```
PROCEDURE DoAproc(); (* DoBProc(), DoXProc() is similar *)
BEGIN
  .......
  createMsg(OutMsg,receiver,MsgTypeX,InfoX);
  sendMsg(OutMsg, POMbx);

END DoAProc;

PROCEDURE FTClerk;
BEGIN
 LOOP
  receiveMsg(InMsg,ClerkMbx); (* none available: dispatch *)
  WITH InMsg^ DO
    CASE (MsgType) OF
      AMsg :DoAProc |
      BMsg :DoBProc |
      XMsg :DoXProc
      ELSE DoError;
    END; (* CASE *)
  END; (* WITH *)
  disposeMsg(InMsg);
 END; (* LOOP *)

END FTClerk;
```

Abb. 24 Programmtorso eines Fehlertoleranz-Clerks

FT-Clerks - wie übrigens die Handler-Prozesse auch - haben eine sehr einfache Programmstruktur, die aus ihrem Charakter als ***botschaftsgetriggerter Automat*** resultiert. Darunter soll sehr allgemein ein Funktionsmodul mit Ein- und Ausgängen verstanden werden, dessen Ausgabeverhalten und dessen interner Folgezustand deterministisch aus den Eingaben - diese sind Nachrichten von anderen Funktionsmodulen - und dem aktuellen Modulzustand resultiert. Ein (Implementations-) Modul eines solchen Leichtgewichtprozesses, als Torso in MODULA-2-Syntax in Abb. 24 dargestellt, besteht im wesentlichen aus Bearbeitungsprozeduren, die sich um eingetroffene Botschaften eines bestimmten Typs kümmern, und einem zentralen Botschaftsverteiler in Form einer Programmschleife, der eingetroffene Nachrichten aus der Mailbox entnimmt und an die zuständigen Prozeduren weiterleitet. Die Eingaben werden in den Clerk-zugehörigen Mailboxen entgegengenommen (*receiveMsg()*), und die Ausgaben werden dem Post Office übergeben (*sendMsg(..,POMbx)*). Die Bearbeitung endet immer mit dem Verschicken einer Folgenachricht an einen anderen lokalen Clerk, es sei denn, der aktuelle FTI-lokale Arbeitszyklus muß ausgesetzt werden oder ist zuende gegangen. Ein solcher Arbeitszyklus bezieht sich immer auf die Behandlung eines asynchronen Ereignisses von außerhalb des lokalen Fehlertoleranzinstanz-Prozesses und besteht aus synchronen Clerk-Interaktionen. Bei äußeren Ereignissen handelt es sich um Botschaften von Nachbarknoten oder Nachrichten von den lokalen Handlerprozessen, die Aktivitäten der Bedienperson (user) oder des Anwenderprozesses (user process) melden; im Fehlerfall ist noch eine Timeout-Botschaft vom lokalen Clock-Clerk möglich. Das Post Office priorisiert FTI-lokale Nachrichten, sodaß ein Überfluten der Mailboxen durch äußere Ereignisse nicht zu befürchten ist. Ist die eingetroffene Botschaft verarbeitet, geht der Clerk wieder in den Empfangszustand über. In der Implementation der *receiveMsg()*-Primitive ist das Dispatching der Leichtgewichtsprozesse verborgen. Der FT-Clerk wird suspendiert, bis er eine neue Botschaft in seiner Mailbox entnehmen kann.

5.3.5 Systemsteuerfunktionen des Post-Office

Das Versenden von Nachrichten (per *sendMsg()*) an andere Clerks geschieht immer unter Vermittlung des POout-Clerks des Post Office (siehe auch Abb. 22); nur hier ist die eigentliche Empfängeraddresse (Mailbox oder globaler Kommunikationskanal) bekannt. Für den FT-Clerk ist im besonderen auch nicht ersichtlich, ob der Empfänger sich lokal oder entfernt befindet. Der Indirektionsschritt in der Adressierung erleichtert die Modul-Rekonfigurierbarkeit und unterstützt so z.B. wirkungsvoll einen Integrationstest der FTL-Programmmodule auf einer Monoprozessorumgebung. Diese Eigenschaft hat auch die Vorab-Simulation [Brau87] erleichtert. Die lokale Vermittlungstätigkeit des POout-Clerks ist beendet, wenn er die Botschaft in die private Mailbox des jeweiligen Empfänger-Clerks kopiert hat, und dies dem Dispatcher in einer Warteschlange *SignalQueue* signalisiert hat.

Die zweite Hälfte des Post Office, der POin-Clerk, ist für den Empfang von Nachrichten zuständig. Immer wenn sich ein anderer Clerk in seiner *receiveMsg()*-Primitive suspendiert, gewinnt er die Kontrolle über den Prozessor. Mit dem Wechseln eines Leichtgewichtsprozesses auf Empfangsbereitschaft ist ein Bereit-Eintrag in eine weitere Dispatcherwarteschlange *ReadyQueue* verbunden. Diese wird zunächst zusammen mit der *SignalQueue* vom Dispatcher im POin-Clerk ausgewertet, um einem bereiten Prozeß den Vortritt zu lassen, bevor die zentrale Handler-Pipe gelesen wird. Im Falle, daß dort keine Botschaft vorrätig ist, geht das gesamte Leichtgewichtsprozeß-System in den Ruhezustand über.

5.4 Fehlertolerante Interprozessor-Kommunikation

5.4.1 Das Problem der konsistenten Systemsicht

Die Interprozessorkommunikation in der verteilten Systemumgebung von ATTEMPTO dient der globalen Koordination von Systemaktionen für Ressourcenverwaltung und Fehlertoleranzfunktionen, nicht jedoch zur Übermittlung von Information zwischen Anwenderprozessen. Der Verteilungscharakter bleibt somit beschränkt auf das Betriebssystem. Wie in allen verteilten Systemen ist die konsistente Datenhaltung ein grundsätzliches Problem, wenn auf zentralisierte Lagerung verzichtet wird, und stattdessen Daten von globaler Bedeutung in lokalen Abbildern gehalten werden. In unserem Fall beinhalten diese lokalen Abbilder Systemdaten, z.B. Belegungszustände von globalen Ressourcen, Jobwarteschlangen usw. Die Konsistenzforderung verlangt, daß die lokale Systemsicht immer dann einen gültigen globalen Systemzustand reflektiert, wenn die jeweiligen Daten referenziert werden. Eine Lösung dafür ist, bei jeder Zustandsänderung alle Replikate zu synchronisieren (*Update*). Für den Datenabgleich sind Protokolle erforderlich, die gewisse Fehlertoleranzeigenschaften aufweisen sollten, um Inkonsistenzen durch den Abgleichvorgang selbst zu vermeiden.

Verschiedene Ansätze für die Lösung des Konsistenzproblems sind bekannt, die auch auf unterschiedlichen Fehlermodellen beruhen. Zwei Hauptansätze sind zu sehen: die Bereitstellung von Kommunikationsprimitiven wie z.B. Atomarer Rundspruch (atomic broadcast [CAS85]), die eine zuverlässige Botschaftsübermittlung auch trotz Fehlern im System zu garantieren suchen, und - auf einer höheren Ebene - Transaktionsmechanismen für durch Prozedur-Fernaufruf kommunizierende Prozesse, z.B. [BJ85]. Transaktionsmechanismen lösen das Commit-Problem (Lampson-Sturgis 1976: 2-Phase-Commit) [La81], das sich - obwohl verwandt - vom Übereinstimmungsproblem (Problem der byzantinischen Generäle [Lam82]) unterscheidet, das Grundlage für den Rundspruchansatz ist. Der Unterschied liegt hauptsächlich darin, daß das Commit-Problem Übereinstimmung zwischen allen Beteiligten verlangt, und also nur im fehlerfreien Fall überhaupt einen Fortschritt der Berechnungen zuläßt, dafür aber den Schutz des existierenden Datenbestandes garantiert. Transaktionsmechanismen haben daher ihre besondere Bedeutung in Datenbankanwendungen, wo es vor allem auf Integrität der Datenhaltung ankommt. Das Übereinstimmungsproblem hingegen verlangt Übereinstimmung nur zwischen den intakten Beteiligten, d.h. es sucht Fehler im System zu tolerieren, indem es sie unwirksam macht. Verfahren, die dem oben zuerst genannten Hauptansatz folgen, befassen sich also mit zuverlässigem und atomarem Rundspruch (reliable broadcast, atomic broadcast [CAS85], [SDS84]) bzw. byzantinischen Übereinstimmungsprotokollen [DolStro85][1].

Allgemein können die Mechanismen zur Konsistenzsicherung entweder direkt zur expliziten Benutzung bereitgestellt werden, oder aber verdeckt durchgeführt werden, d.h. sie sind entweder im System oder einer Programm-Laufzeitumgebung verborgen. Den Synchronisationsvorgang transparent für den Programmierer zu halten, schafft die Abstraktion eines *synchronen replizierten Datenspeichers* (*synchronous replicated storage* [CDSA90]), die es erlaubt, bekannte nebenläufige Programmierparadigmen für Systeme mit gemeinsamem Speicher auf solche mit verteiltem Speicher zu übertragen.

Im ATTEMPTO-System kommunizieren Betriebssystemprozesse und -prozessgruppen. Letztere bestehen aus den Fehlertoleranzinstanzen einzelner Rechnerknoten, die sich zur Bearbei-

1. Siehe dazu auch Kapitel 3.3

tung eines Benutzerjobs zusammengefunden haben. Ihre Zahl richtet sich nach dem jeweiligen Fehlertoleranzgrad. In einem solchen nachrichtengekoppelten System sind explizite Kommunikationsprimitive von Vorteil, die einen zuverlässigen atomaren Multicast realisieren, d.h. Prozeßgruppen addressierbar machen und die Nachrichtenbermittlung auf niedriger Systemebene implementieren. Dieser Systemdienst bildet die Grundlage für ein auf höhere Schichten übergreifendes Konzept zu Konsistenzwahrung.

Broadcast (“Einer an alle”) oder allgemeiner *Multicast* (“Einer an viele”) bezieht sich zunächst auf die Kommunikationsstruktur 1:n, d.h. daß einem Sender einer Botschaft n, also mehrere Empfänger gegenüberstehen. Ist das Kommunikationsmedium linienförmig (z.B. ein Bus), können alle adressierten Empfänger gleichzeitig ein- und dieselbe Nachricht empfangen, ansonsten muß die Nachricht indirekt über Relaisstellen weitergeleitet werden. Beides - Gleichzeitigkeit und Einmaligkeit -ist äußerst erwünscht in verteilten Systemen zur Wahrung einer konsistenten Systemsicht, wenn globale Systemzustände in lokalen Abbildern gehalten werden, die mithilfe von Nachrichten aktualisiert werden müssen. Genau genommen kommt es bei lose gekoppelten Rechnern nicht so sehr auf das “gleichzeitig”, d.h. zum selben absoluten Zeitpunkt, an, als auf die garantiert gleiche zeitliche Reihenfolge des Eintreffens von Nachrichten bei den Empfängern (eine Eigenschaft, die “gleichzeitig” sozusagen gleichzeitig miterfüllt). Eine exakte Gleichzeitigkeit der Entgegennahme bzw. auch Bearbeitung von Botschaften kann bei asynchronen, d.h. autonom getakteten Rechnereinheiten in einem möglicherweise sogar inhomogenen System ohnehin nicht gewährleistet werden. Ähnlich verhält es sich mit der “Einmaligkeit”; dahinter verbirgt sich die Eigenschaft der *Atomarität* (Unteilbarkeit, s.u.): eine Nachricht gelangt entweder zu allen vorgesehenen (fehlerfreien) Empfängern oder zu keinem, und der *Identität*: natürlich ist es auch wünschenswert, daß alle die gleichen Daten empfangen. Wegen unvermeidlicher physikalischer Ungleichheiten (zB. unterschiedliche Schwellwerte in Leitungsempfängern, ortsabhängig verteilter Störbeeinflussung) ist Identität auch nicht naturgegeben und muß durch Maßnahmen der Fehlererkennung und -korrektur erst erlangt werden. Dadurch wird der Broadcast zum *zuverlässigen Broadcast.* Zuverlässigkeit ist zunächst ein allgemein die Qualität kennzeichnendes Merkmal und bedarf deshalb einer genaueren quantitativen Bestimmung. Dazu muß ein *Fehlermodell* definiert werden, das die Sichtweise angibt, welche Fehlertypen bzw. -klassen berücksichtigt werden und wie sich Fehler im System auswirken.

Atomarer Rundspruch (atomic multicast) ist ein zuverlässiger Rundspruch mit folgenden Eigenschaften [CAS85]:

- **Atomarität:** entweder alle intakten Kommunikationspartner des Senders erhalten die Botschaft oder aber keiner von ihnen.
- **Globale Reihenfolge**: alle Botschaften im System (in der Prozeßgruppe) treffen bei allen intakten Empfängern (der Gruppe) in derselben Reihenfolge ein.
- **Terminierung:** das Protokoll hat eine endliche maximale Laufzeit Δ.

Die letzte Eigenschaft gilt nur in sogenannten *synchronen Systemen* [DolStro85], in denen eine maximale Nachrichtentransferdauer und eine Begrenzung für den lokalen Rechenzeitbedarf zur Kommunikationsbearbeitung angebbar ist. Dies ermöglicht, die Rundspruchprotokolle in mehreren synchronisierten Phasen durchzuführen, und führt zu einem zeitphasensynchronem Abgleich der dezentral gelagerten Daten. In diesem Sinne ist das ATTEMPTO-System ein synchrones System, wenn auch die einzelnen Rechnereinheiten autonom und nicht takt- oder rahmensynchron arbeiten.

Die Gewährleistung der o.a. Bedingungen kann abhängig vom betrachteten Fehlermodell und der jeweiligen Netztopologie erheblichen Aufwand erfordern. Besonders das Einhalten der globalen Empfangsreihenfolge kann zeitaufwendig werden, sodaß die Benutzung des atomaren Rundspruchs auf ein Minimum beschränkt werden muß. Im ISIS-System [BJ858] werden daher unterschiedliche Klassen von Rundspruchprimitiven angeboten, deren Kosten aber auch Leistungsumfang differieren. So gibt es z.B. einen billigen Rundspruch, der nur *lokale Reihenfolgekonsistenz* sichert , d.h. bei y treffen die Nachrichten von x ind der Reihenfolge ein, wie x sie ausgesendet hat, Nachrichten von z sind aber beliebig, d.h. bei unterschiedlichen Knoten möglicherweise verschieden, eingereiht. Dies genügt für viele Anwendungsfälle; ein besonderes Beispiel ist die Übermittlung von *Commit*- bzw. *Abort*-Nachrichten in einem verteilten Transaktionssystem mithilfe dieser Primitiven.

Die gesicherte globale Empfangsreihenfolge im Kommunikationssytem ermöglicht eine *Serialisierung* von Aktionen in höheren Systemschichten bei konkurrierendem Zugriff auf Systemdaten durch unterschiedliche Instanzen. In ATTEMPTO werden dazu alle Aktionen, die sich auf bestehende globale Systemdaten beziehen, durch Botschaftsaustausch synchronisiert; dies gilt sowohl für die Auswertung wie auch die Modifikation dieser Daten, damit synchrone Entscheidungen aufgrund des globalen Systemzustandes möglich werden. Die Konsistenzwahrung beruht also im wesentlichen auf einem globalisierten Aktionsstrom, an dem alle Rechnerknoten beteiligt sind. Der eigentliche Mechanismus heißt ***Selbstattraktion***: zunächst wird durch eine Bewerbung versucht, die Verfügungsgewalt über eine globale Date (oder allgemeiner ein globales Betriebsmittel) zu erlangen, bevor sie verändert werden darf. Bei mehreren konkurrierenden Zugriffswünschen entsteht ein Wettlauf. Dieses Grundprinzip zum synchronen Abgleich der dezentralen Systemtabellen ist in Kapitel 5.4.3 näher beschrieben.

Globalisierte Aktionen der Fehlertoleranzschicht dienen dazu, eine Übereinkunft über die Behandlung der lokal überwachten Anwenderprozeßreplikate zu erzielen, z.B. zur dezentralen Fehlermaskierung, zur Festlegung, welches Replikat die Ausgabe durchführen darf, etc. Solange diese Entscheidung aussteht, wird der Anwenderprozeß angehalten. Auf diese Weise werden die Anwender-Prozeßexemplare mit in den Aktionsstrom eingefügt und dadurch synchronisert. Synchrone Entscheidungen der Fehlertoleranzinstanzen sind möglich, weil Aktionsanreize in Form von Rundspruchnachrichten die Fehlertoleranzinstanzen im gleichen Zustand[1] erreichen. Diese Betrachtungsperspektive erlaubt es, sich die Fehlertoleranzinstanzen zusammen mit ihren Prozeßreplikaten als “botschaftsgetriggerte Automaten” vorzustellen. Tatsächlich spiegelt sich dies wider in der Programmstruktur der Fehlertoleranz-Systemprozesse (Kapitel 5.3.4). In dieser Hinsicht wird eine Verwandtschaft des ATTEMPTO-Ansatzes mit dem Konstrukt der *“fehlertoleranten Zustandsmaschine” (state machine approach* [Schn90]) deutlich. Eine Zustandsmaschine besteht aus Variablen, die ihren Zustand repräsentieren, und Kommandos, die eine Zustandstransformation und möglicherweise eine Ausgabe bewirken. Kommandos sind deterministische Programme und werden als atomare Aktionen abgearbeitet. Äußere Anreize erhält die *state machine* durch *requests*, die die Kommandos kennzeichnen und alle Informationen für die Durchführung des Kommandos mit sich tragen. Die begriffliche Identität mit einem *endlichen Automaten* ist nur oberflächlich: die *state machine* besitzt auch lokale Zustandsvariablen und ist nur insofern endlich, als reale Speicherplatzbeschränkungen dies erzwingen. Letztendlich stellt dieser Ansatz eine Programmstrukturierungsvorschrift dar und bietet eine Unterstützung für Replikation zum Aufbau fehlertoleranter Systeme. Die Ähnlichkeit mit dem ATTEMPTO-Strukturmodell für die Fehlertoleranz-Systemprozesse ist zufäl-

1. Hiermit ist ein globaler Zustand gemeint. Daneben existiert ein lokaler Zustand, der auf der lokalen Aktionshistorie beruht.

lig. Der ATTEMPTO-Ansatz ist zudem weit weniger formal, dafür aber gut an das flexible Fehlertoleranz-Konzept angepaßt.

Die Fähigkeit zum atomaren Rundspruch schafft die Grundlage für Konsistenz im verteilten ATTEMPTO-System. Diese Eigenschaft des Kommunikationssystems auf der Basis einfacher Hardware zu realisieren, ist eine besondere Aufgabe im Rahmen des ATTEMPTO-Projekts. Das Message-Passing-Kommunikationssystem für physikalisch speichergekoppelte Multiprozessoren mit Broadcast- und Fehlertoleranzeigenschaften beruht auf programmgesteuerten Mechanismen zur Datenübermittlung und ist weitestgehend unabhängig von der zugrundeliegenden Standard-Parallelbus-Hardware (VMEbus). Die Wirkungsweise wird im weiteren erläutert, und einige spezielle Besonderheiten der Implementation als Treiber unter UNIX werden dargestellt.

5.4.2 Kommunikationskanäle

Mit dem VMEbus liegt ein linienförmiger Bus vor. Broadcast-Datentransferzyklen im Sinne einer gleichzeitigen Mehrfachadressierung sind bei asynchronen Multiprozessor-Parallelbussen wie dem VMEbus in der Regel aber nicht möglich; dies würde nämlich zB. eine geschlossene Behandlung der einzelnen Datentransferbestätigungssignale sowohl beim schreibenden Busmaster wie auch den -slaves erfordern. Deshalb muß der Sender die gleiche Botschaft mehrfach (n-mal) verschicken, wobei die Nachrichtenreihenfolge bei allen Empfängern auch bei konkurrierenden Sendewünschen gewahrt bleiben muß. Dazu benötigt man einen Arbitrationsmechanismus für unteilbare Nachrichtenpakete. Der Systemarbiter könnte dazu verwendet werden, wobei der VMEbus (durch Aktivhalten des BBSY*-Signals) für die Gesamtübertragungsdauer eines Paketes bestehend aus n gleichen Botschaften von dem einen Master (Sender) blockiert wäre. Genau dies ist jedoch nicht wünschenswert einerseits aus Fehlertoleranzgründen (defekte Einheit blockiert das ganze System), andererseits, weil die maximal zulässige Blockierungsdauer des Busses mit der festgelegten Zeitkonstanten von häufig vorhandenen Hardware-Watchdog-Timern für die Busfehlerdetektion abgestimmt werden müßte. Deshalb wählen wir einen eigenen Arbitrationsmechanismus, der im folgenden noch vorgestellt wird. Die eigentliche Arbitrationsaufgabe beruht im wesentlichen auf einer global koordinierten Synchronisation von Interruptsignalen, die auf einem gesonderten Bus geführt werden. Auf untersten Protokoll-Ebene, der VMEbus-Hardwareebene, kann der Bus-Requester in der einzeloperationsbezogenen RWD - (Release When Done-) Betriebsart verwendet werden. Das bedeuted, daß beim nebenläufigen Transport von Datenpaketen unterschiedlicher Sender verzahnte Buszugriffe zugelassen sind.

Als getrennte logische Kommunikationskanäle werden separate Adressbereiche in lokalen *Dual-Ported-Rams* definiert (Abb. 25). Dies ermöglicht das quasi-parallele Versenden von Botschaften, ohne die Integrität eines einzelnen Datenpaketes zu zerstören. Gleichzeitig sollen damit noch weitere Designanforderungen erfüllt werden:

- **Dezentralisierung von gemeinsam genutzten Speicherbereichen:**
  Jede Rechnereinheit (Knoten) öffnet einen begrenzten lokalen Speicherbereich zum Zugriff durch andere Systemeinheiten über den VMEbus. Hier befindet sich für jeden Knoten ein Briefkasten (Mailbox) für dessen Botschaften. Interessanterweise gibt es also auch einen Briefkasten für Botschaften an sich selbst. Die Aufteilung des gemeinsamen Speichers in einzelne knotenzugehörige Bereiche vermeidet einen zuverlässigkeitskritschen zentralen Speicher.

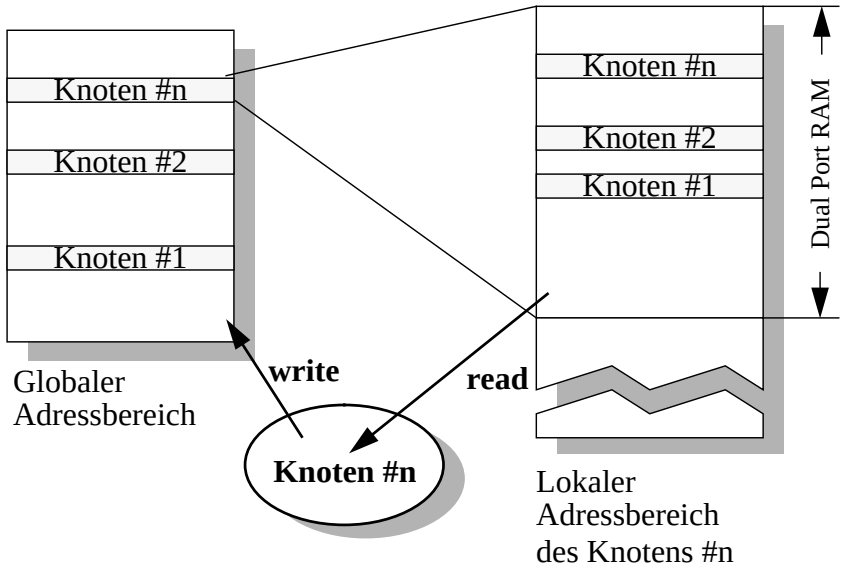


Abb. 25 Mailbox- Adressbereiche (Memory Map)

- **Entkopplung der Kommunikationsbeziehungen:**
  Wegen der separaten Briefkästen, die jeder Knoten für jeden bereithält, sind unidirektionale Kommunikationskanäle gegeben. Allgemein gilt: das Schreiben in fremde Briefkästen führt über den globalen Bus, das Lesen hingegen über lokale Adressen. Diese Separierung ermöglicht den Einsatz von hardwareunterstützten Speicherschutzmechanismen (MMU) und vermeidet die Zerstörung von bereits eingetroffenen Nachrichten durch den Empfängerknoten selbst. Die Festlegung der Lese-Einwegrichtung auf den lokalen Bereich hält das globale Kommunikationsmediums frei von Abfrageaktionen auf eingetroffene Botschaften in den jeweiligen knotenzugehörigen Mailboxen und trägt so mit zu einer Verkehrsentlastung bzw. hohem Busdurchsatz bei. Das Übermittlungsprotokoll ist fehlertolerant, d.h. es enthält Mechanismen zur Fehlererkennung und zur Korrektur durch Wiederholung von Botschaften. Es reagiert jedoch sensibel auf Verfälschung der Telegramme in den den einzelnen Knoten zugeordneten Mailboxen durch Fremdeinfluß, also durch andere Rechnerknoten. Solche Verfälschungen sind möglich durch transiente Störungen auf den Adressleitungen während der Übertragung (einem Schreibzyklus) auf dem VMEbus. Da hier keine Möglichkeit zur Fehlerkorrektur besteht, wird versucht, die Wahrscheinlichkeit für solche Störauswirkungen durch eine *robuste Kodierung der Mailboxbasisadressen* zu minimieren. Dabei ist zu unterscheiden zwischen einer fehlerhaften Knotendekodierung (im globalen Adreßbereich) und einer Fehldekodierung der Mailbox dort. Es muß also sowohl im globalen wie im lokalen Memory Map ausreichender Sicherheitsabstand vorhanden sein. Die robuste Kodierung wird in der Art vorgenommen, daß der Sicherheitsabstand es erforderlich macht, daß gleichzeitig mehrere Adressleitungen ihre Polarität wechseln müssen, damit es zu einer Fehladressierung in einen gültigen Bereich kommt.

Die Vorgehensweise zur robusten Addreßkodierung entspricht der Erstellung eines systematischen Codes [Pet67] unter der Annahme eines *symmetrischen Binärkanals*: ein Fehler bedeutet das “Umkippen” eines Binärelements in seinen dualen Wert. Die Größe des gesamten global für Mailboxen zur Verfügung gestellten Speichersegmentes bestimmt zusammen mit der Ausdehnung einer globalen Mailbox die Dimension des Coderaumes, wobei bei 2^k maximal vorgesehenen Knoten und n-stelligen Codevektoren n-k Prüfstellen (Adreßleitungsbits) für die

Redundanzkodierung vorhanden sind. Unter diesen gegebenen Umständen ist ein *Linearcode mit maximaler Hammingdistanz* zu suchen In der vorliegenden Realisierung erreicht das globale Adressmapping einen Hammingabstand von 4, das lokale einen Abstand von 3. Die Größe einer privaten Mailbox und damit die maximale Paketgröße einer Nachricht beträgt 1kbyte [Gü88]. Die resultierenden Mailboxbasisaddressen und ihre Berechnung können dem Anhang entnommen werden.

5.4.3 Übermittlungsprinzip

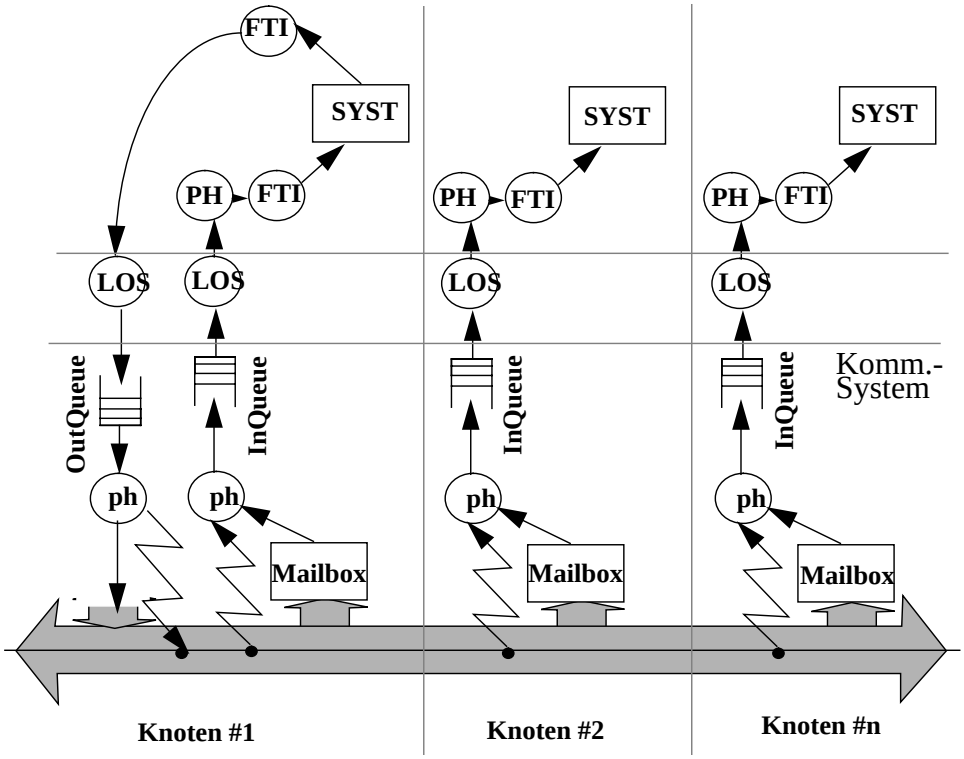


Abb. 26 Rundspruch-Übermittlung auf Prozeß- und Kommunikationssystemebene

Für einen Multicast werden zunächst alle (dem Sender gehörenden) Mailboxen auf den Empfängerknoten beschrieben, danach wird bei allen Knoten ein Interrupt ausgelöst und somit das Ende des gesendeten Nachrichtenpakets angezeigt. Auf diese Weise werden mehrere Bustransferzyklen zu einem logischen Paket zusammengefaßt. Jedem Knoten ist eine Interruptleitung fest zugeordnet. Für die Interrupts besteht eine Prioritätsfestlegung. Alle Leitungen sind in einem Bus angeordnet, der parallel zu allen Knoten führt. Das Broadcastproblem wird dadurch von der VMEbus-Datentransferbuskomponente auf diesen Interruptbus verlagert. Das Reihenfolgeproblem wird damit zu einem Problem der globalen Interruptsynchronisierung, eine Aufgabe, die das Übermittlungsprotokoll programmgesteuert übernehmen muß. Das Übermittlungsprinzip ist in Abb. 26 dargestellt. Knoten #1 fungiert hier als Sender des Rundspruchs. Der Rundspruch wird notwendig, da die lokale Fehlertoleranzinstanz (FTI) dieses Knotens einen schreibenden Zugriff auf Systemdaten vornehmen will. Um diesen mit allen replizierten lokalen Systemtabellen (SYST) zu synchronisieren, wird zunächst eine Anforderung (*Request*), dies zu tun, als Broadcast verschickt. Der Sender befindet sich selbst in der Empfängermenge. Mit dem Erhalt der Anforderung dürfen nun alle Knoten ihre Systemdaten modifizieren. Die gesicherte globale Empfangsreihenfolge von Rundspruch-Botschaften bewirkt eine

Serialisierung des Zugriffs auf diese kritischen Daten und verhindert so eine nichtdeterministische Durchmischung von Zugriffen auf den lokalen Systemdatenrepräsentationen. Das Verschicken von Botschaften ist ein Dienst des lokalen Betriebssystems LOS. Hier nimmt ein *Porthandler-Devic*e (/dev/ph) diese Aufgabe wahr. Beim Empfang einer Botschaft - angezeigt durch den Interrupt - wird die Mailbox entleert. Sie wird für den Empfängerprozeß in einer Warteschlange *Inqueue* bereitgestellt und von dort über den Porthandler-Prozeß (PH) der lokalen FTI zugeführt.

5.4.4 Paketstruktur einer Nachricht

Im Zusammenhang mit diesem Kapitel wird unter einer Nachricht eine strukturierte Informationseinheit zur Datenübermittlung verstanden. Im OSI-Kommunikationsmodell ist diese Funktion der Schicht 2 zugeordnet. Im Sinne dieses Modells ist die Schicht 3 leer (kein Netzwerkrouting). Der Anwenderzugriff zum Kommunikationssystem liegt auf der Schicht 4, d.h. auf der Transportschicht. Das lokale Betriebssystem erbringt als Dienstleistung einen gesicherten Transport von Nachrichten. Eine virtuell geschaltete Verbindung ist wegen der geforderten Rundspruchscharakteristik nicht realisiert und auch nicht nötig, d.h. auf der Ebene des Anwenderzugriffs beinhalten die Nachrichtenpakete als eigenständige Einheiten alle die komplette Adressierungsinformation. Um einen Gruppenrundspruch addressierbar zu machen, enthält das Schicht-4-Adreßfeld ein Bitfeld, das die Empfängerknoten durch ein gesetztes Bit ausweist. Weitere Kontrollelemente dieser Schicht sind Länge der Nutzinformation (InfoLen) und ein Nachrichtentyp (MsgTyp), der letztendlich den zuständigen Clerk der lokalen Fehlertoleranzinstanz des (bzw. der) adressierten Knoten(s) kennzeichnet. Für die Datenübermittlungsschicht ist die Struktur des Transportschicht-Datenpaketes transparent.

Jedes Nachrichtenpaket besteht aus einer Folge von Datenwörtern, deren Ausdehnung der Busbreite entspricht. Diese Vereinheitlichung gestattet automatische Blocktransfers bei maximaler Transfergeschwindigkeit mit optimal großen Atomaroperationen[1]. Die Nachricht besitzt einen Kopf (Header), der verschiedene Steuerelemente enthält, wie z.B. Absenderkennung, Sendefolgenummer, Längenausweisung des Pakets, Checkinformation zur Datensicherung, und einen Rumpf, der die eigentliche Nachrichteninformation trägt, siehe Abb. 27. Die maximale Paketlänge ist ein Systemparameter, der von der gewählten Mailboxgröße abhängt. Das Übermittlungsprotokoll sieht keine Maßnahmen zur Stückelung großer Informationsblöcke vor; dies wäre Aufgabe eines Transportprotokolls auf Anwenderebene. Für den Fehlertoleranzbetrieb reichen kurze Meldungen (<64 Byte), sodaß Vorkehrungen für große Datenblöcke z.Z. ohne Anwendung blieben. Im Nachrichtenkopf befindet sich zusätzlich ein Kontrollfeld zum Versenden von Empfangsbestätigungen (ACKs). Auf diese Weise ist sozusagen jedem Nachrichtenkanal ein Steuerkanal (ACK channel) beigeordnet, der auch unabhängig von einer Nachrichtenübermittlung genutzt werden kann: es gibt also Datenpakete mit Nutzinformation (eigentliche Nachrichten), solche, die gleichzeitig (“huckepack”) Steuerinformation mit sich führen, und reine Steuerpakete (Nachrichtenlänge = 0).

Das Kontrollfeld weist die Bestätigung selbst aus (ACK) oder kann Statusinformation transportieren (z.B. Receiver Not Ready). ACKs werden im beschränkten Rahmen zur Flußsteuerung herangezogen, hauptsächlich jedoch dienen sie als Sicherungselement. Eine Botschaft gilt dann als zuverlässig übermittelt, wenn von allen Empfängern eine Bestätigung zurückgekommen ist, bis dahin ist weiteres Senden von der gleichen Einheit über den Bus blockiert. Eine

1. Byte-Transfers würden genausolang dauern, aber weniger Information transportieren, Langwort-Transfers erfordern 2 Buszyklen/Grundinformationseinheit bei einer Busbreite von 16 Bit.

Zeitüberwachung wird bei Ausbleiben einer erwarteten Bestätigung eine festgelegte Maximalzahl von Sendewiederholungen anreizen. Dieses Protokoll ist als “PAR-Protokoll” (Positive Acknowledgement Reply) bekannt.

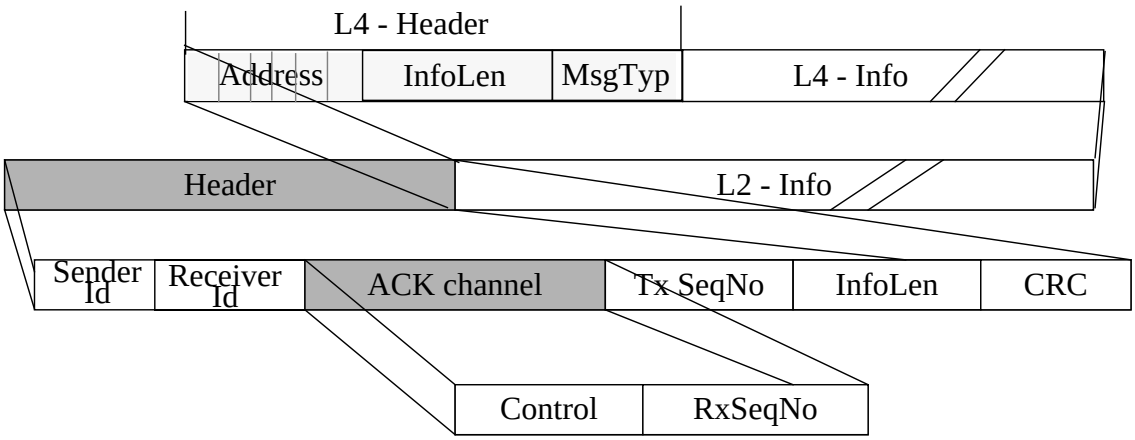


Abb. 27 Aufbau eines Nachrichtenpaketes der Datenübermittlungs- und Transportschicht

5.4.5 Fehlermodell des Übertragungskanals

Die bereitgestellten Strukturelemente und Mechanismen gestatten die Tolerierung transienter und intermittierender Fehler bei der Datenübermittlung sowie das Erkennen permanenter Fehler. Fehler können sich auswirken durch Pegelverfälschung auf einzelnen oder auch mehreren Daten- und Adressleitungen, sowie durch das Ausbleiben von Bestätigungsmeldungen. Da die Fehlerbehebungsstrategie ausschließlich auf Zeitredundanz beruht (Wiederholungen im Fehlerfall) können permanente Hardwarefehler des Übertragungskanals nicht toleriert werden. In einem solchen Fall ist die Übertragung von Interprozessornachrichten generell unmöglich, solange keine alternativen Kommunikationsverbindungen existieren. Aus der Sicht von intakten sendenden Einheiten ist es nicht sofort möglich, permanente Knotenfehler von permanenten Fehlern des übertragungskanals zu unterscheiden. Dazu sind weitere Tests nötig, die auch mehrere Knoten in Interaktion miteinbeziehen. Der Gewinn ist jedoch nur gering, weil entsprechende Fehlerbehebungsmaßnahmen für das Kommunikationsmedium fehlen. Deshalb unterscheidet unser Fehlermodell nicht zwischen permanenten Knotenfehlern und Verbindungsfehlern. Der Fehlerbereich der einzelnen Knoten wird dazu auf die Verbindungen ausgedehnt. Eine solche Vereinfachung wird häufig getroffen; sie ist jedoch dann kritisch, wenn die einzelnen Fehlerbereiche gemeinsame Schnittmengen bilden. Bei einem zentralen Kommunikationsmedium ist dies nur dann zu rechtfertigen, wenn dieses ausreichend zuverlässig ist, um nicht die Gesamtzuverlässigkeit des Systems zu stark zu begrenzen. Dies trifft im System ATTEMPTO zu, worauf im Kapitel 6.4 näher eingegangen wird. Permanente Fehler werden während des Betriebs als solche (pauschal) erkennbar, wenn die Wiederholungsmechanismen ohne Erfolg bleiben. Sie werden in diesem Fall willkürlich dem bzw. den Empfängerknoten zugeordnet. Für den weiteren Betrieb werden diese Einheiten vom Broadcast ausgenommen.

Gewisse Fehlereinwirkungen werden im Modell nicht berücksichtigt; dazu gehören: Meldungsüberflutung, Verlust von Interrupt-Signalen, Zerstörung fremder Mailbox-Inhalte, Störungen auf bestimmten Steuerleitungen und Kontrollelementen.Diese Fehler gehören zu den *byzantinischen Fehlern*, die hier nicht als geschlossene Klasse zur Tolerierung in Betracht gezogen werden. Diese Klasse kennzeichnet Fehler, die sich äußern, als sei böswilliger Hinterhalt im Spiel, und die sich deshalb nur sehr schwer erkennen lassen. Einige weitere Beispiele: ein

Knoten gibt sich als fremder aus, Nachrichteninhalte sind trotz korrekter Checkinformation falsch, Broadcast-Adressaten werden mit unterschiedlichen Daten versorgt. Insbesondere der letzte Fehlerfall, der sogenannte *Replikationsfehler*, ist zum Inbegriff byzantinischen Fehlverhaltens geworden. Allgemeiner kann formuliert werden: byzantinisches Fehlverhalten bezeichnet nichtspezifiziertes Systemverhalten beliebiger Art einer Systemkomponente, für das aus isolierter lokaler Sicht in der Regel auch keine Möglichkeit zur Fehlererkennung besteht. Im Abgleich mit anderen replizierten Komponenten jedoch können solche Fehler unter bestimmten Voraussetzungen erkannt und unwirksam gemacht werden. Dazu dienen z.B. die byzantinischen Übereinstimmungsprotokolle (Kapitel 3.3). Neben den byzantinischen Fehlern gibt es weitere übliche Fehlerklassen, die das Fehlverhalten beschreiben. Sie lassen sich in einer Hierarchie anordnen [DolSto85].

Unterlassungsfehler $\subseteq$ **Zeitfehler** $\subseteq$ **Byzantinische Fehler**

Die Klasse der byzantinischen Fehler ist als Obermenge anzusehen, die neben allgemein unspezifiziertem Verhalten auch nichtzeitgerechtes Verhalten und Unterlassungsfehler beinhaltet. Unterlassungsfehler sind wieder eine Untermenge der Zeitfehler. Diese Klassen kennzeichnen auch Symptome des Ausfalls von Komponenten. Unterlassungsfehler sind z.B. einem *Fail-S*top-Fehlermodell (Anhalteausfall [Echt90]) zuzuordnen, sofern sie permanent sind. Es geht davon aus, daß alle Nachrichten einer Komponente vertrauenswürdig sind. Solange sie ausgesendet werden, ist die Komponente intakt, ansonsten werden aber keinerlei Meldungen ausgesendet. Jede Komponente stellt somit einen ideal isolierten Fehlerbereich dar, was Zuverlässigkeitsbetrachtungen und die Festlegung geeigneter Fehlertoleranzmechanismen erheblich vereinfacht. Im Fehlerfall geht die defekte Komponente in einen Zustand über, der es den mit ihr kommunizierenden Komponenten gestattet, den Ausfall zu erkennen, im Idealfall unmittelbar nach dem Ausfall. Sehr häufig wird der Ausfall aber erst beim Ausbleiben von erwarteten Reaktionen, z.B. von Nachrichten, sichtbar. In diesem Fall unterscheidet sich der Fail-Stop-Ausfall nicht vom sogenannten *Crash-Ausfall* (Komponentenzusammenbruch), der ein abruptes Dienstversagen kennzeichnet ohne Mitteilung an benachbarte Komponenten [DolSto85]. Das System erleidet einen *persistenten Unterlassungsfehler* [Lap90] und damit Unterlassungsausfall. *Byzantinisches Ausfallverhalten* ist gekennzeichnet durch Fehlerpropagierung über den lokalen Fehlerbereich der Defektkomponente hinaus, das Fehlverhalten ist willkürlich. Byzantinische Übereinstimmungsprotokolle (siehe auch Kapitel 3.3), die solche Fehler tolerierbar machen, verlangen in der Regel sehr zeitaufwendige Interaktionen zwischen den beteiligten Einheiten (Absprachen und Rückabsprachen), die sehr viele Meldungen produzieren. Deshalb sind diese Protokolle in der Praxis kaum anzutreffen.

Das ATTEMPTO-Fehlermodell geht in erster Linie von Unterlassungsfehlern aus. Die unterste Stufe der Fehlererkennung liegt im Kommunikationssystem: hier zeigt sich der Fehler unmittelbar beim Senden durch Ausbleiben von Bestätigungsmeldungen des Empfängers. Aufgrund von Akzeptanzkriterien beim Empfänger als fehlerhaft erkannte Botschaften werden dort nicht verworfen, stattdessen wird die Bestätigung unterdrückt und auf die Wiederholungsmeldung gewartet, um die bereits eingetroffene Botschaft möglicherweise zu korrigieren. Die verstümmelte Nachricht behält solange ihrem Platz in der Empfangswarteschlange. Bleiben wiederholte Sendevorgänge ergebnislos, wird der Fehler auch in höheren Schichten des ATTEMPTO-Betriebssystems (ATOS) sichtbar, z.T. durch synchrone Ergebnismitteilung (PHYSERROR-Anzeige in der UNIX-*errno*-Variablen) oder durch Ablaufen von Zeitschranken dieser höheren Schicht.

Im ATTEMPYTO-System wird versucht, die Interaktionen zwischen den beteiligten Knoten

so zu minimieren, daß keinerlei Rücksprache erforderlich ist[1]. Die Folge davon ist, daß allgemeines byzantinisches Fehlverhalten vom Kommunikationssystem nicht toleriert werden kann. Die Zuverlässigkeits- und Atomaritätseigenschaften des atomaren Rundspruchs verlangen aber, daß wenigstens Replikationsfehler ausreichend unwahrscheinlich sind, um die Konsistenz von Systemabbildern auf intakten Knoten nicht zu gefährden. Dabei hilft die Signatur (Checksumme) in den Meldungspaketen. Damit wird eine Unterklasse der byzantinischen Fehler erkennbar und tolerierbar, die *durch Authentifikation erkennbaren byzantinischen Fehler* [CDSA90]. Der Authentifikationsmechanismus ist ursprünglich dazu gedacht, Botschaften bei ihrer Weiterleitung durch Dritt-Übermittler fälschungssicher zu machen. Da im ATTEMPTO-System eine vollständige Vernetzung existiert, die Netztopologie also Vermittlungsstellen unnötig macht, ist hier ein unterschiedliches Szenario gegeben. Als übermittelnde Instanz kann jedoch die Kommunikationsschicht des lokalen Betriebssystems gesehen werden, die Broadcast-Botschaften der ATOS-Betriebssystemprozesse zu den einzelnen Empfängern sukzessive versendet. Hier macht die Signaturbildung vor allem folgende bestimmte Fehlerfälle unwahrscheinlich:

- ***Verfälschung fremder Nachrichten nach erfolgter Ablagerung:*** robuste Addreßdekodierung, Checksumme im Datenpaket, doppelte Addreßinformation durch globale Bus-Addresse der Mailbox und Knotennummer im Absenderfeld. Möglichkeit des hardwareüberwachten Schreibschutzes.
- ***Replikationsfehler bei der Botschaftsvervielfachung durch den Sender:*** Transport in die Mailboxen aus einem gemeinsamen senderseitigen Lagerort (OutQueue) nach vorheriger Konfektionierung: wenigstens Übertragungsfehler lassen sich durch Inkonsistenz mit der Checksumme erkennen.
- ***Verfälschung von Kontrolleinheiten des Protokolls:*** kann erkannt werden, da die Checksumme auch den Botschaftsheader mit einschließt.

Weitere byzantinische Fehlerfälle sind durch Interrupt- und Bestätigungsmechanismus abgedeckt:

- ***Verlust von Interruptsignalen:*** wenn, dann höherwahrscheinlich bei allen wegen Nichtanlegung des Interrupts (entspricht Unterlassungsfehler), da ein Broadcast-Sendeinterrupt auf einer einzigen physikalischen Leitung geführt ist. Für ACK-Interrupts greifen die Wiederholungsmechanismen.

Die Robustheit gegenüber byzantinischen Fehlern beruht also hauptsächlich auf Kodierungs- bzw. Informationsredundanz. Diese unterstützt einerseits die Fehlererkennung und gestattet so in Verbindung mit Zeitredundanz Fehlertolerierung. Andererseits wird das Auftreten bestimmter Fehler unwahrscheinlich gemacht. Solche Maßnahmen sind immer dann wirkungsvoll, wenn das Fehlermodell *Angriffs-Ausfälle* [Echt90] auschließen kann, d.h. bewußte böswillige Manipulationen zur absichtlichen Erzeugung von Fehlern nicht angenommen werden müssen.

5.4.6 Sicherung der Reihenfolgekonsistenz

Da die Hardware nicht fähig ist, einen atomaren Broadcastdatentransfer zu unterstützen, wird das Problem auf die Ebene der Datenübermittlung (OSI-Schicht 2) verlagert. Hierfür ist zunächst eine Strukturierung des physikalischen Kommunikationsdatenstromes zu definieren, wie bereits in Kapitel 5.4.4 geschehen, und es ist ein Protokoll zu definieren, das die konflikt-

1. Unter Rücksprache soll hier eine Indirektion verstanden werden in der Form: A teilt B mit, C hätte ihm gesagt, daß .. .

freie Nutzung der Übertragungskapazität durch die beteiligten Rechnereinheiten festlegt. Ziel ist, den Nachrichtenaustausch der einzelnen Knoten untereinander so zu koordinieren, daß ausgesendete Nachrichtenpakete unzerteilt bei allen Rundspruchaddressaten in derselben Reihenfolge empfangen werden. Die synchronisierte Empfangsreihenfolge muß dabei nicht unbedingt der tatsächlichen Sendereihenfolge entsprechen. Kapitel 5.4.3 hat bereits das Übermittlungsprinzip vorgestellt. Dort wurde auch bereits verdeutlicht, daß die Gewährleistung der globalen Reihenfolge eine Aufgabe der globalen Synchronisation von Empfangsinterrupts ist. Bei der zeitlich losen Kopplung der autonomen Rechnerknoten besteht hier ursächlich die folgende Problematik:

- ***Die Unbestimmtheit in der lokalen Erfassung.***

  Dies ist das allgemeine Problem der Quell-Kongruenz, das in asynchronen Systemen immer bei replizierten Eingabe-Sensoren auftritt; es wurde bereits in Kapitel 3.2 beschrieben. Eine Streuung von physikalischen Parametern der replizierten Sensoren, z.B. unterschiedliche Schwellwerte der Interrupt-Latches, sowie unterschiedliche Interrupt-Service-Latenzen bedingt durch die lokale Taktung der Prozessoren, sind unvermeidlich. Bei konkurrierendem Zugriff (multiplen Interrupts) gibt es keine deterministische lokale Reihenfolge des Eintreffens der Unterbrechungen, und deshalb bei replizierten Interrupthandlern auch keine globale. Anstelle der realen muß also eine künstliche Reihenfolge treten, über die global Übereinstimmung besteht. Die Basis dafür schafft eine Prioritätsfestlegung für die Unterbrechungen. Die Interrupts werden bis zum Zeitpunkt der Auswertung in einem Interrupt-Register gespeichert.

- ***Rundensynchronisierung.***

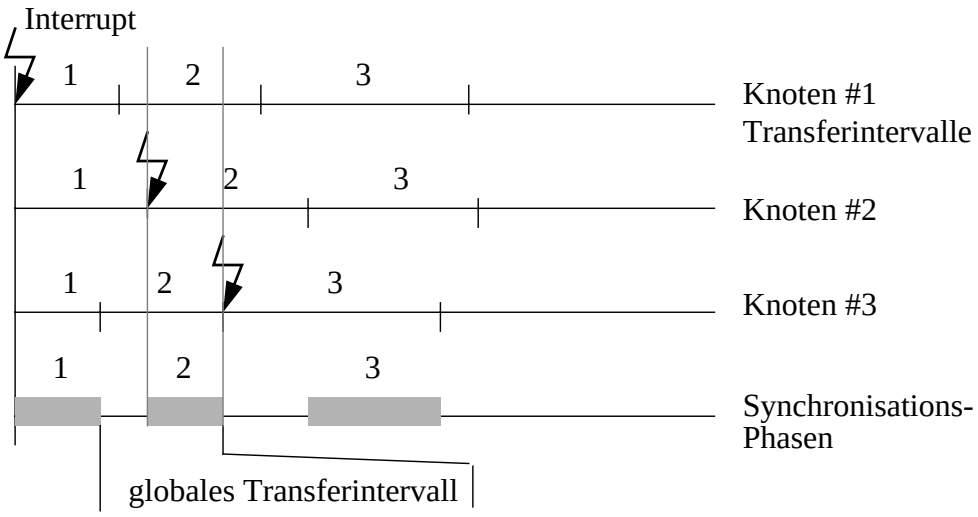


Abb. 28 Synchronisationsintervalle

Aufeinanderfolgende Nachrichtenübertragungen (Interruptfolgen) müssen unterscheidbar sein. Allgemein muß es Betriebsphasen der Übertragung geben, während der eine synchrone Auswertung der Interrupt-Register möglich ist. Gleichzeitig müssen während dieser Synchronintervalle die Mailboxen entleert werden können, um ein Überschreiben zu verhindern. Während dieser Zeit darf kein Knoten im System einen Kommunikationsinterrupt auslösen. Diese Verhältnisse sind in Abb. 28 dargestellt: die einzelnen durch die lokalen Bedingungen in ihrer Länge bestimmten Transferintervalle müssen gemeinsame nicht verschwindende Deckungsbereiche besitzen. Diese kennzeichnen den globalen

Synchronisationszustand. Mit dessen Beendigung ist das aktuelle Transferintervall global beendet. Erst dann darf das nächste beginnen. Eine Übertragung gilt also dann als abgeschlossen, wenn sicher ist, daß alle Adressaten die Botschaft erhalten haben.Der Konsens über diesen Zeitpunkt ist bei allen intakten Knoten im System erforderlich, auch wenn sie nicht durch den Rundspruch addressiert wurden. Er kann auf verschiedene Weise erlangt werden.

1. Aufgrund einer expliziten Absprache (geregelte Synchronisation, siehe auch Kapitel 3.1), z.B. durch Kenntnisgabe des jeweiligen lokalen Zustandes an die Allgemeinheit. Im fehlerfreien Fall gelangen alle Knoten in den synchronisierten Zustand und das aktuelle Nachrichtenübertragungsintervall gilt als beendet. Im Fehlerfall sind jedoch zusätzliche Maßnahmen erforderlich, um die Synchronisation der intakten Komponenten und die unbedingte Terminierung von globalen Transferintervallen zu garantieren. Je nach angenommenem Fehlermodell sind sie entsprechend aufwendig, siehe dazu auch Kapitel 3. Für Fehlertoleranz des Synchronisationsmechanismus selbst kann dann z.B. ein Maskierungsmechanismus eingesetzt werden. Dieser Weg wird in [FriWe82], [DaWa78] und auch in [YST85] gegangen. Die Votierung kann aufgrund von binärwertigen Zustandsvariablen durchgeführt werden (sychron/nicht synchron) und kann durch einen Mehrheitsentscheid bei 2t+1 Knoten t nicht synchronisationsfähige Knoten tolerieren (Synchronization Voting [DaWa78]). Benutzt man mehrwertige Maskierungsvariablen oder auch Mengen von Zustandsvariablen, die die lokal erkannten Interrupts kennzeichnen, kann mit der Votierung gleichzeitig die Reihenfolgeabsprache selbst vorgenommen werden, dies sichert z.B. die globale Interruptpriorität in SIFT [FriWe82][1]. Die dezentralen Votierungseinheiten müssen allerdings selbst wieder synchronisiert werden. Dazu werden in allen oben erwähnten Anwendungsbeispielen implizite Zeitschranken für angenommene Maximalabweichungen der Synchronität im fehlerfreien Fall festgelegt. Grundsätzlich müssen natürlich Wege zu Weiterleitung der lokalen Zustandsdaten existieren. Im speichergekoppelten SAFE-System [YST85] werden dazu lokale Speicherzellen verwendet, auf die die Allgemeinheit Zugriff hat. Die Rundensynchronisierung ist hier jedoch recht komplex.

2.) Aufgrund von ausschließlich impliziten Zeitvorgaben (implizite Synchronisation). Wie in Kapitel 3.1 bereits ausgeführt, setzt sie voraus, daß Annahmen über eine Maximalabweichung in der Synchronität getroffen werden können. Wenn man auch aus lokaler Sicht in der Regel nichts über die absolute Dauer der einzelnen Transferintervalle weiß, so läßt sich doch häufig eine maximal mögliche Abweichung korrespondierender Intervalle angeben, d.h. mit welcher maximalen Verzögerung das langsamste Knotenexemplar gemessen am schnellsten auf den Interrupt reagiert. Weiß man auch etwas über die maximale Bearbeitungsdauer für den Interrupt - dies ist in der Regel immer gegeben, da es durch die Implementation des Kommunikationsprotokolls selbst bestimmt ist -, kann man auch eine maximale Transferintervallzeit angeben. Wenn Zeitfehler nicht toleriert werden müssen, ist dann keine Absprache mehr nötig, um die Synchronität intakter Einheiten zu gewährleisten, vorausgesetzt das Gesamtsystem startet aus einem definierten Anfangszustand. Für das Protokoll heißt dies: alle Knoten im System müssen nach erkanntem Empfangs-Interrupt eine ***Sperrzeit*** abwarten, bevor sie selbst einen

1. Der in [FriWe82] nur oberflächlich erklärte Mechanismus zur Reihenfolgesynchronisierung kann in der vorliegenden Form nicht überzeugen. Bei mehr als 2 quasi gleichzeitig auftretenden Interrupts scheint bei einer TMR-Konfiguration sogar bei Fehlerfreiheit eine Maskierungsentscheidung nicht immer möglich.

Sendeinterrupt generieren. Dies verhindert einerseits, daß Interrupts zu Zeitpunkten ausgelöst werden, zu denen ihre Auswertung nicht mehr allgemeingültig gesichert ist, und andererseits, daß sich aufeinanderfolgende Nachrichten in einer Mailbox überschreiben.

Beide Synchronisationsvarianten können Unterlassungsfehler von Komponenten tolerieren. Im Falle der impliziten Synchronisation werden diese jedoch nicht direkt erkannt, da das korrekte Funktionieren nicht sofort überprüft wird. Zusätzliche Bestätigungsmechanismen, die zur Erlangung der geforderten Zuverlässigkeitseigenschaften des Rundspruchs ohnehin erforderlich sind, können hier jedoch zumindest dem Sender in unmittelbarer Folge diesen Fehler aufdecken. Im Falle der expliziten Synchronisation erhalten allerdings sogleich alle intakten Rechnereinheiten Kenntnis davon. Ein weiterer Vorteil des ersten Verfahrens ist, daß auch die eigentliche Existenz des Interrupts global verifiziert wird. So können z.B. transiente Fehler tolerierbar werden, die den Signalpegel auf der allen gemeinsam zugeführten Interruptleitung in einen verbotenen Pegelzwischenbereich verfälschen, und so bei intakten Knoten unterschiedlich erfaßt werden können. Dieser Fehler gehört in die Klasse der Replikationsfehler (die Hauptklasse der byzantinischen Fehler). Im allgemeinen können aber weder das erste noch das zweite Verfahren die gesamte Fehlerklasse tolerieren. Können z.B. im ersten Fall die Zustandsmeldungen Replikationsfehlern ausgesetzt sein, kann dies in einer weiteren Votierungsstufe durch nichtdeterministischen Maskierungsentscheid berücksichtigt werden, wie bereits in Kapitel 3.3 verdeutlicht wurde. Sowohl [FriWe82] wie auch [DaWa78] gehen auf diese Möglichkeit ein. Auch Zeitfehler, wie zu spät kommende Zustandsmeldungen, Interrupts in der Sperrzeit, usw., können allgemein von beiden Verfahren nicht toleriert werden. Interruptleitungen und lokale Zeitgeber sind also dem Perfektionskern zuzuordnen.

Den relativ geringen Vorteilen in der Fehlerüberdeckung des ersten Verfahrens zur Rundensynchronisierung steht allerdings stark gestiegener Aufwand in der Protokollabwicklung gegenüber. Deshalb wurde für das ATTEMPTO-System das implizite Synchronisationsverfahren angewendet. Vor dem Senden muß also jeder Knoten prüfen, ob die Sperrzeit bereits verstrichen ist.

- ***Entscheidungsinvalidation (Interrupts bei der Zeitabfrage).***

  Für das Protokoll wichtige Entscheidungsabfrage-Ergebnisse dürfen nicht durch eingeschobene Interrupts invalidiert werden.
  Bei der zuvor beschriebenen impliziten Synchronisationsvariante ist es möglich, daß direkt nach einer Abfrage des lokalen Sperrzeit-Timers ein fremder Interrupt eintrifft, der zunächst bearbeitet wird. Die Bearbeitung schließt ein, daß ein neuer Transferzyklus mit einer ebenfalls neuen Sperrphase beginnt. Hat die vorhergehende Sperrzeitabfrage das Ergebnis “Sperrzeit ist abgelaufen” erbracht, wird diese Information nach Beendigung der Interrupt-Service-Routine veraltet sein und dennoch fälschlicherweise benutzt. Der Knoten wird dann unter Mißachtung der Sperrzeit senden. Aus diesem Grund muß die Aktionsfolge *Abfrage des Sperrzeit-Timers* und *Senden im erlaubtem Fall* atomar gestaltet werden, wodurch der Sendeinterrupt dem richtigen Transferintervall zugeordnet wird. Zu diesem Zweck werden Empfangsinterrupts lokal für die Dauer dieses kritischen Abschnitts gesperrt.

- ***Arbitrierung.***

  Interrupts können lokal unterschiedlich gesperrt sein. In diesem Fall wird die Empfangsinterrupt-Bearbeitung unterschiedlich verzögert einsetzen. Insbesondere kann die Verzögerung auch dadurch entstehen, daß ein Knoten sich gerade in dem o.a. kritischen

Abschnitt zum Senden aufhält. Alle Knoten, die vor Auslösen des Interrupts durch den vorgenannten Knoten bereits ihre Interrupt-Service-Routine begonnen haben, werden den zu spät gekommenen (nachzügelnden) Interrupt nicht erkennen. Damit die globale Reihenfolge nicht inkonsistent wird, muß aber die Arbitration, d.h. die Auswertung der lokalen Interrupt-Latches unter Berücksichtigung der Prioritäten, und damit die Festlegung der Reihenfolge, dezentral in einer Zeitspanne erfolgen, in der Änderungen ausgeschlossen sind. Um dies zu erreichen, müssen alle Knoten die Möglichkeit in Betracht ziehen, daß Interrupts "nachzügeln" können. Dazu warten sie nach Eintritt in die Interrupt-Bearbeitung eine ***Karenzzeit*** ab, die ausreicht, um stabile globale Verhältnisse zu schaffen, bevor sie ihre Interrupt-Register auswerten.

Mit dieser Schilderung der grundsätzlichen Problematik war bereits der Lösungsansatz verbunden. Es stellt sich jedoch die Frage, ob bzw. unter welchen Bedingungen die Synchronisation über die Betriebsdauer hinweg aufrecht erhalten werden kann. Insbesondere muß dazu geklärt werden, wie die wichtigen protokollspezifischen Zeitparameter Sperrzeit und Karenzzeit zu dimensionieren sind. In der Regel wird dafür ein formaler Beweis verlangt, dem eine anerkannte Verifikationstechnik zugrundeliegt. Insbesondere für Kommunikationsprotokolle besteht eine besondere Notwendigkeit dazu, da hier vielfältige, für den Programmimplementierer nicht sofort einsichtige Möglichkeiten für Verklemmungen, Hyperaktivitäten, Aushungern und dergleichen vorliegen. Deshalb wurde bereits in [Ris87] eine Simulation des Protokolls anhand von zeitattributierten Petrinetzen vorgenommen, die geeignet ist, eine Grundvoraussetzung für Vertrauenswürdigkeit zu schaffen. Sie gibt allerdings keine Auskunft über konkrete Dimensionierungsvorschriften für die Zeitparameter. Im Rahmen dieser Arbeit soll auf eine formale Verifikation verzichtet werden. Kapitel 6.2.2 gibt Aufschluß darüber, welche Maßnahmen zur Protokollverifikation durch Testen vorgenommen wurden. Um im Kontext dieses Kapitels gleichzeitig mit der Vorstellung des Protokoll-Funktionsprinzips die generelle Funktionsfähigkeit zu zeigen, eignet sich aber gut ein Vergleich mit einem existierenden Kommunikationsprotokoll. In der Bewältigung der Asynchronität zwischen Sender und Empfanger besteht hier nämlich eine Ähnlichkeit zum asynchronen Start/Stop-Betrieb, z.B. entsprechend der CCITT - V-Normen (gemeinhin als V.24 bzw. RS232C bekannt). Hier besteht das Problem darin, taktautonome Sender/Empfängerpaare auf asynchron übermittelte Datenpakete zu synchronisieren. Die Datenpakete werden dazu in eine Envelopstruktur (Startbit-Datenbits-Stopbit) festgelegter Nominallänge verpackt. Das Startbit ist das Synchronisationselement für den Paketbeginn, das Stopbit dient der Abtrennung aufeinanderfolgender Pakete und garantiert so die Erkennbarkeit des nächsten Paketanfangs. Die Abastung des Signals beim Empfänger kann die gesendete Information fehlerfrei regenerieren trotz der abweichenden Phasenlage und real unterschiedlichen Taktfrequenz, da zusätzliche (Zeit-)Bedingungen eingehalten werden: die Phasenabweichung ist begrenzt wegen der Abtastung mit einem Vielfachen der Übertragungsrate, und die maximal zulässige Frequenzdifferenz der lokalen Taktgeneratoren ist definiert.

Bei uns besteht das Problem natürlich darin, mehrere Empfänger zeitlich zu synchronisieren; ähnlich wie beim Start/Stop-Betrieb geschieht dies aber dadurch, daß das globale Übertragungsmedium nur zeitlich gerastert genutzt werden kann. Die Rasterintervalle starten jeweils mit der Erkennung eines Paketbeginns, danach folgt eine Sperrung des Mediums für weitere Pakete zur Entgegennahme der Daten und eine Erholungszeit zum sicheren Erkennen eines neuerlich folgenden Pakets.

"Globales Übertragungsmedium" meint hier jedoch nicht die Datentransferbuskomponente des VMEbus, sondern einen abstrakten Interruptkanal, der nur geringe Anforderungen an die realen physikalische Gegebenheiten stellt: die Hardware der Rechnerknoten muß Interrupts eines

jeden Knoten durch jeden unterstützen[1]. Im einfachsten Fall - und von diesem wird hier ausgehen - sind damit die regulären VMEbus-Interrupts gemeint, allerdings in einer vorteilhaften *point-to-multipoint*-Betriebsart ohne Interrupt-Bestätigungszyklus. Dazu müssen Interrupt-Requester und -Handler geeignet steuerbar sein, was jedoch häufig der Fall sein dürfte. Jeder Rechnerknoten besitzt also einen nur ihm zugehörigen Interrupt auf einer separaten Prioritätsstufe, was allerdings so die maximale Zahl von Rechnereinheiten auf 7 beschränkt. Ein solcher globaler Interrupt kennzeichnet den Start einer (zu empfangenden) Botschaft (entspricht Startbit); er wird von allen Rechnerknoten - also auch dem sendenden - bearbeitet. Er wird jedoch bei den Empfängern mit unterschiedlichen Verzögerungen erkannt (entspricht unterschiedlicher Phasenlage der Abtastfrequenz) je nachdem, ob einzelne Prozessoren gerade interruptgeschützt arbeiten oder nicht. Die maximale “Phasenabweichung” wird in unserem System durch die Karenzzeit berücksichtigt, die zunächst abgewartet werden muß, bis sich alle Prozessoren im Zustand “unterbrochen” befinden. Während dieser Zeit werden alle Interrupts aufgesammelt; verbunden mit der Prioritätsvergabe der Interrupts sorgt dieser Mechanismus für die notwendige Arbitration, d.h. die konfliktauflösende Zuteilung des “Übertragungsmediums” Interruptkanal. Als nächstes erfolgt die Entgegennahme der Daten beim Empfänger , d.h. die Entleerung der Mailbox. Ein weiteres Datenpaket darf erst dann folgen, wenn der (bzw. die) Empfänger die Daten auch störungsfrei empfangen hat (haben) und bereit für das nächste ist (sind). Der globale Empfang einer Botschaft ist dann beendet, wenn mit Sicherheit jeder Prozessor seine Empfangs-Interrupt-Service-Routine beendet hat. Erst nach Ablauf dieser “Cleartime” bzw. Sperrzeit dürfen weitere Sendeinterrupts ausgelöst werden. Diese Bedingung garantiert gleiche zeitliche Reihenfolge der Botschaften dadurch, daß alle Prozessoren in einen Ruhezustand gebracht werden, aus dem heraus sie gleichermaßen in der Lage sind, neue Empfangsinterrupts aufzusammeln. Wie beim Start/Stop-Betrieb muß die Informationsmenge normiert werden, damit eine solche Zeit überhaupt definiert werden kann: die Zeit muß immer ausreichen, auch die längste Botschaft aus der Empfangsmailbox zu entnehmen und darauf die Interrupt-Service-Routine zu beenden. Zur qualitativen Bestimmung von für das Funktionieren des Protokolls notwendigen Zeitbedingungen soll das Zeitdiagramm Abb. 29 dienen. Es demonstriert gleichzeitig die Analogie zum Start/Stop-Betrieb: wie dort kann man die maximale Zeitdauer für die Start/Daten/Stop-Phasen aus vorgegebenen Randbedingungen (Toleranzen) angeben.

Die Einzeldiagramme demonstrieren das zeitliche Verhalten in unterschiedlichen Knoten während der Bearbeitung eines externen Portinterrupts (INTx), dabei gilt für alle eine gemeinsame Zeitachse. Knoten #1 weist eine minimale Interruptlatenz tk1=0 auf. tki ist die Zeit, die der Knoten #i benötigt, um mit der Bearbeitung eines Empfangsinterrupts zu beginnen (Start der Interrupt Service Routine ISR). Alle beteiligten Knoten müssen innerhalb der Karenzzeit tkar ebenfalls soweit sein.Der Knoten #n bearbeitet den Interrupt mit der maximal zulässigen Verzögerung tkn=tkar. Zu Beginn der Bearbeitung warten alle Prozessoren die Zeit tkar ab, bevor sie mit der eigentlichen Auswertung ihres Interrupt-Registers beginnen und die Empfangsdaten entgegennehmen, d.h. die Mailbox entleeren. Dafür brauchen sie unterschiedliche Zeiten tisrx, die jedoch die Sperrzeit (Cleartime) tclear nicht überschreiten dürfen. Während dieser Zeit dürfen keine Sendeinterrupts ausgelöst werden. Dies verhindert, daß ein Prozessor einen Interrupt zu niedriger Buspriorität zu früh bearbeitet, ohne etwaige im zulässigen Zeitraum später folgende höher priorisierte zu berücksichtigen. Die früheste Zeit für einen Folgeinterrupt auf dem

1. Die Single Board Computer können z.B. einen Interrupt-Controller für parallele Eingangsleitungen (je eine pro Knoten) und eine frei programmierbare Ausgangsleitung besitzen. Die Ausgangsleitungen werden zu einem Interruptbus zusammengefaßt, der über den P2-Stecker zu allen Interrupt-Controllern führt Ein Interrupt wird dann durch einen Puls auf einer Ausgangsleitung ausgelöst.

Interruptbus ist bestimmt durch das kürzest mögliche Intervall Tmin=tkar+tclear, wie für den Knoten #1 dargestellt. Erfolgt ein solcher (INT1 in der Abb. 29), dürfen weitere höchstens bis zum Zeitpunkt Tmin + tkar = 2 tkar + tclear ausgelöst werden (oder gar nicht in diesem Intervall)..

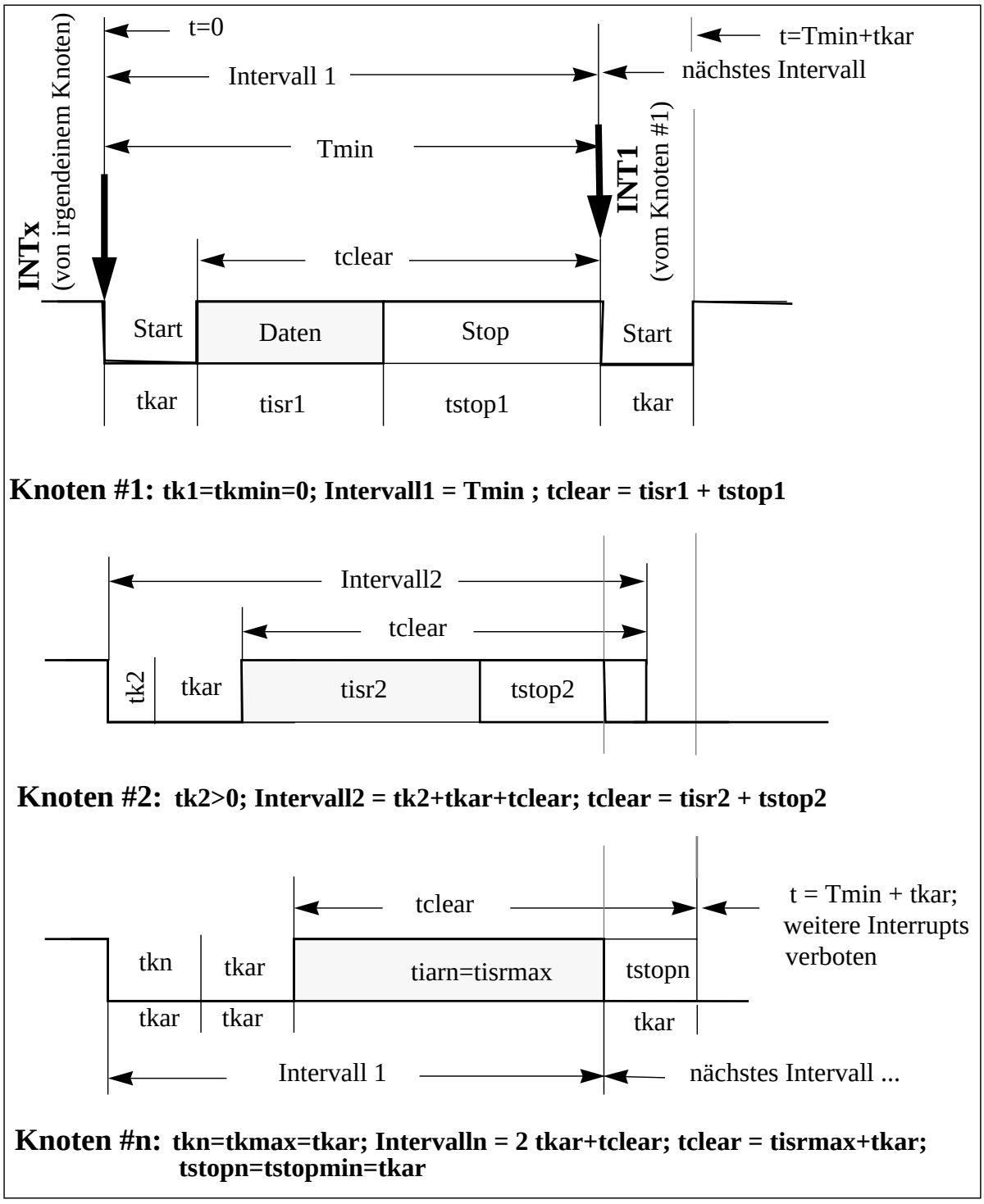


Abb. 29 Interruptsynchronisierung in Transferintervallen

Diesen pessimistischen Fall zeigt die Abb. 29 für den Knoten #n; falls er einen Sendewunsch hat, muß er diesen um ein weiteres Intervall verzögern. Um dies gewährleisten zu können, muß unbedingt der (ja bei #n schon anstehende) INT1 zunächst bearbeitet werden, um eine Retrig-

gerung des Cleartimers vorzunehmen, bevor dessen Ablaufen einen eigenen Sendeinterrupt (von #n) auslösen kann. Realisiert man z.B. die Sperrzeitüberwachung mithilfe eines retriggerbaren Hardwaretimers, der nach Ablauf einen internen (Sende-) Interrupt generiert, so muß dieser Interrupt auf einer niedrigeren Prioritätsstufe liegen als der Portinterrupt.

Es lassen sich folgende (notwendige) Bedingungen für die Dimensionierung der kritischen Zeitwerte (Konstanten) tkar und tclear aus der Darstellung Abb. 29 ableiten:

$$(\mathbf{I})\ tki < tkar\ ;\ tkar > t_{interrupt\ disable\ max}\ ;$$

d.h. die Karenzzeit muß größer sein als die längste Zeit, in der der Portinterrupt gesperrt sein kann.;

Dies umfaßt die Summe aller Interruptbearbeitungszeiten für Interrupte höherer Prozessorpriorität, und muß auch die längste Serviceroutine auf gleicher Prioritätsstufe und die Maximaldauer für Interruptsperren zur Absicherung kritischer Abschnitte im Betriebssystem berücksichtigen. Die meßtechnische Erfassung der einzelnen Zeitanteile ist selbst bei detaillierter Betriebssystemkenntnis nicht einfach. Fehlt die Kenntnis - etwa weil das Betriebssystem nur in der Binärversion vorliegt -, kann die Karenzzeit nur geschätzt werden.

$$(\mathbf{II})\ tclear > tkar + tisrmax;$$

die Sperrzeit startet nach Beginn eines Intervalls aus einem für alle Knoten geltenden Ruhezustand nach Ablauf der Interruptreaktionszeit tki und der Karenzzeit tkar. tisrmax ist die garantiert längste Bearbeitungszeit bis zur Beendigung der Portinterrupt-Service-Routine. Hierin ist nicht nur die eigentliche Bearbeitung des Portinterrupts selbst enthalten, sondern auch eine mögliche Unterbrechung dieser durch einen höherpriorisierten Interrupt. Bis zum Zeitpunkt des frühest möglichen Folgeinterrupts durch irgendeinen Knoten müssen alle Portinterruptbehandlungen abgeschlossen sein.

Grundsätzlich müssen natürlich wegen des Fehlens einer Absolutzeit alle Zeiten als toleranzbehaftet angesehen werden. Dies läßt sich aber einfach durch einen Sicherheitszuschlag auf die Karenz- und Sperrzeit berücksichtigen. In Bezug auf die Sperrzeit wirkt er sich so aus, als ob zu ihr noch eine minimale Stoppzeit gehört, also

$$tclear = tkar + tisrmax + tstopmin;\ tstopmin = fest > 0,$$

was das Minimalintervall Tmin verlängert und dadurch jedem Knoten mehr Zeit läßt, im Ruhezustand zu verbleiben. In jedem Fall wirkt sich der Sicherheitszuschlag auf die Übertragungsgeschwindigkeit aus. Ein Sicherheitszuschlag delta auf die Karenzzeit

$$tkar = tinterrupt\ disable\ max + delta$$

wird sich im Falle von passivem Warten während dieser Zeit zusätzlich auf die Rechenleistung auswirken.

Durchsatzbezogene Eigenschaften dieses Kommunikationsprotokolls werden im Rahmen der Leistungsbewertung (Kapitel 6.5) untersucht.

5.4.7 Der Kommunikationstreiber unter UNIX

Die Kommunikationsinstanz, Porthandler /dev/ph genannt, ist als *Device* ein Treiber im UNIX-Kern. UNIX gibt gewisse Randbedingungen vor zur Konstruktion von Gerätetreibern [Egan88]. Ein Prozeß kommuniziert mit seinem Treiber über einen Satz von fünf Primitiven. Dies sind die Systemaufrufe *open(), close(), read(), write(), ioctl()*. Die Implementation dafür in Prozeduren stellen für zeichenorientierte Gerätetreiber (Character Devices) gleichzeitig den Grundrahmen zur Strukturierung des Treiberprogramms dar. Die Systemaufrufe *read()* und *write()* werden von aktiven Prozessen benutzt, um Sende- und Empfangsvorgänge über einen Kommunikationstreiber einzuleiten. Ein Treiber bearbeitet diese zunächst im jeweiligen Prozeßkontext, d.h. synchron. Die vom Kern für Treiber zur Verfügung gestellten Systemaufrufe (*driver system calls*) *sleep()* und *wakeup()* werden benutzt, um Wartebedingungen gerecht zu werden, etwa weil zur Befriedigung des *read*-Systemaufrufes noch keine Daten vorrätig sind, oder weil Ressourcen fehlen (z.B. bei Pufferspeichermangel oder weil der Kommunikationskanal belegt ist). Mit *sleep()* ist eine explizite Prozessorabgabe durch den Prozeß selbst verbunden, eine andere Möglichkeit als die eigene Suspendierung gibt es im Treiber nicht: Prozesse, die im Kern agieren, sind nicht verdrängbar. Neben der Bearbeitung im Prozeßkontext findet bei Asynchron-Treibern noch außerhalb jedes Prozeßkontextes die Bearbeitung von Hardware-Interrupts statt. Sende- und Empfangsinterrupt koordinieren die Interprozeßkommunikation auch über Prozessorgrenzen hinweg. Sie übermitteln entweder Botschaften (*messages*) oder bestätigen solche (ACKs).

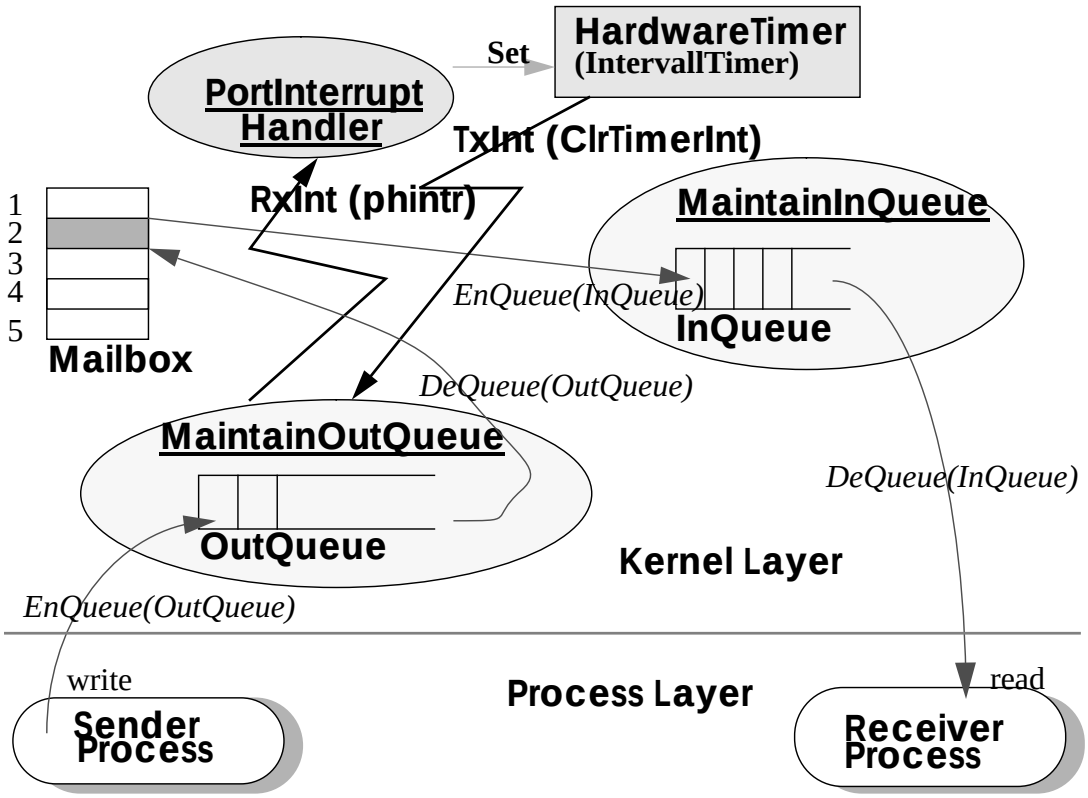


Abb. 30 Asynchroner Port-Treiber, Interrupt- Bootstrap-Mechanismus

Aus der Sicht der Kommunikationstreiber besteht das Transportmedium aus den Mailboxen, die nichts weiter sind als nichtlokale Speicherbereiche. Zugriffe auf diese dauern genausolang wie lokale Speicherzugriffe, falls konkurrierende Busbelegungen fehlen.

Zur Charakterisierung des Sendevorgangs sind zunächst noch gewisse Festlegungen nötig:

Asynchrones Senden heißt: der *write*-Systemaufruf eines Prozesses terminiert sofort, nachdem das (lokale oder entfernte) Betriebssystem die Daten entgegengenommen hat. Der Sendeprozeß kann dann sofort in seiner Programmbearbeitung fortfahren. Wenigstens auf einer Seite muß das System die Daten zwischenpuffern. Sendeseitiges Zwischenpuffern wird i.a. nötig sein, um mehreren Sendeprozessen die Möglichkeit zu geben, den gleichen Kanal benutzen zu können, ohne auf komplette Erledigung vorheriger Sendewünsche warten zu müssen. Empfangsseitige Ablagerung in einer Warteschlange schafft schnell Platz in den Mailboxen und unterstützt die Reihenfolgeverwaltung. Ein ACK wird erzeugt, wenn die Botschaft dort eingereiht worden ist. Dadurch kann mit maximaler Frequenz gesendet werden (hohe Übertragungsrate auf der Leitung). Wegen eines drohenden Überlaufs der Empfangswarteschlange muß bei Erreichen eines gewissen Füllstandes das Senden von Bestätigungen sonderbehandelt werden.

Synchrones Senden bedeutet: der *write*-Systemaufruf wird solange an der Terminierung gehindert, wie Bestätigungen noch ausstehen, d.h. der Adressat die Botschaft als empfangen quittiert hat. In der Regel qualifiziert man den Synchroncharakter auf Prozeß- bzw. Anwendungsebene. Der Adressat ist damit der Empfangsprozeß; er gibt explizit die Bestätigung, oder das entfernte System erteilt diese nach dem Kopieren der übermittelten Daten in dessen Speicher, kurz bevor der *read*-Systemaufruf des Empfangsprozesses terminiert. Der Vorteil des synchronen Sendens ist, daß diese Methode eine natürliche Flußkontrolle durch ACKs bewirkt. Eigentlich ist weder sende- noch empfangsseitig eine Zwischenpufferung notwendig, dann ist jedoch der gesamte Kanal zwischen zwei Knoten für alle Prozesse blockiert für die Dauer der Übertragung. Außerdem ist der Mutiprozeßbetrieb auf der Senderseite solange unterbrochen, bis die Sendedaten direkt vom Benutzeraddreßraum in die Mailbox kopiert sind. Zu berücksichtigen ist jedoch, daß die Empfangsmailbox sobald wie möglich nach Erhalt einer Botschaft freizuräumen ist, da möglicherweise ACKs von der Gegenstelle (das ist der Sender der Nachricht) als Reaktion auf eine zuvor selbst gesendete Botschaft Platz finden müssen. Das Senden von Bestätigungen darf nicht von zuvor von der Gegenseite erhaltenen ACKs abhängig gemacht werden. Um Verklemmungen zu vermeiden, muß entweder eine Zeitbedingung eingehalten werden - wie durch das Freiräumen -, oder es muß ein separater ACK-Kanal existieren.

Während bei Synchronbetrieb sich hauptsächlich ein Gewinn in der Gesamt-Transportzeit ergibt durch Wegfall von Zwischenpufferinstanzen, liegt bei Asynchronbetrieb der Hauptvorteil in der schnellen Fortführung des schreibenden Prozesses, der sozusagen vom Kommunikationskanal entkoppelt ist. Hier ist bedingt durch das erforderliche Kopieren im Speicher die Netto-Prozeß-zu-Prozeß-Transferrate geringer als bei synchronem Senden. Ein anderer Aspekt ist, daß der sendende Prozeß die Kenntnis darüber verliert, welche Meldungen überhaupt auf der Gegenseite eingetroffen sind: sicherere Kanäle sind nötig, oder aber das Kommunikationssystem überläßt das Erbringen der geforderten Dienstgüte der Anwenderschicht und stellt z.B. nur einen ungesicherten Datagrammdienst zur Verfügung.

In unserem Fall soll das Kommunikationssystem zuverlässige und reihenfolgekonsistente Übertragung garantieren. Wir bevorzugen asynchrones Senden, um den Multiprozeßbetrieb lokal möglichst wenig zu behindern, da die lokalen Fehlertoleranzinstanzen in mehrere Schwergewichtsprozesse aufgespalten sind.

Die folgende Betrachtung der Grundabläufe in der unteren ATTEMPTO-Kommunikationsschicht wird durch die Abb. 30 - Abb. 32 unterstützt.

```
 ph_write() /* Treiber - Einsprung für write-Systemaufruf */
{
   new = createQueueMember();
   copyMsgfromUserSpace();
   s_old = spl(NEW);
   /* lokaler Interruptschutz gewährleistet Schlafen vor Interrupt */
      EnQueue(OutQueue,new);
      MaintainOutQueue();
      sleep(&new,FTLPrio);
   splx(s_old);
   deleteMember(new);
} /* ph_write */

ph_read() /* Treiber - Einsprung für read-Systemaufruf */
{
   s_old = spl(NEW);
     while (InQueueEmpty) sleep(&InQueue);
   splx(s_old);
   /* MaintainInQueue(): Start */
   copyMsgtoUserSpace();
   s_old = spl(NEW);
     DeQueue(InQueue);
   splx(s_old);
   DeleteQueueMember();
   /* MaintainInQueue(): Ende */
} /* ph_read */

MaintainOutQueue() /* Sendebearbeitungsprozedur */
{
   if (OutQueueNotEmpty) {
      if (AckComplete()) {
        DeQueue(OutQueue);
        wakeup(&old);
        first = TRUE; /* erstmaliger Sendeversuch */
        MaintainOutQueue(); /* nimmt sich nächste Nachricht vor */
      };
      if (first) {
        xmit();
        return;
      }
   }
   if (NextTim) {    /* Sperrzeit abgelaufen? */
     prepareAcks(); /* ACKs ggf. schreiben und dies markieren */
      if (IntToFire) /* falls markiert in prepareACKs */
        pulseInt();
   }

} /* MaintainOutQueue */
```

Abb. 31 Stark vereinfachte Treiber-Prozeduren in C-Notation (Teil A)

Im Treiber werden zwei Warteschlangen verwaltet: eine sogenannte *InQueue* für eingehende Botschaften und eine *OutQueue* für zu sendende Nachrichten. Der Treiber besitzt Aktionsprozeduren *MaintainInQueue()* und *MaintainOutQueue()* zur Verwaltung dieser Datenstrukturen. In diesem Zusammenhang erledigt er aktiv das Senden bzw. verteilt Empfangsdaten auf wartende Prozesse. *EnQueue(Queue)* und *DeQueue(Queue)* sind Warteschlangenoperationen feinerer Körnung, die hier noch von Bedeutung sind. Sie bewirken das Ein- bzw. Ausketten von

Warteschlangenelementen. Die damit verbundene Datenübergabe erfolgt immer für komplette Nachrichtenpakete. Die Empfangsinterruptroutine (PortInterrupt Handler) übernimmt eingetroffene Nachrichten aus der Mailbox in die Empfangswarteschlange und versucht dann, die Botschaft am Kopf dieser InQueue dem entsprechenden Empfängerprozess zuzustellen. Ein *read*-Systemaufruf, der bereits zuvor gesendete Daten im Empfangspuffer vorfindet, wird sofort befriedigt. Wenn der Empfängerprozeß die Kontrolle über den Prozessor erlangt hat, kopiert der Treiber die Daten aus dem Kern- in den Anwenderspeicher. In umgekehrter Richtung puffert der Treiber die Daten des Senderprozesses in der OutQueue. Von dort aus gelangen sie in die Mailboxen der entsprechenden Empfängerknoten. Nach dem Ablegen der zu sendenden Nachricht in der OutQueue und einem erstmaligen Sendeversuch wird der sendende Prozeß schlafen gelegt, bis die Übermittlung in die Empfangswarteschlange(n) des (der) Rundspruchaddressaten als gelungen bestätigt ist. Das Senden erfolgt nur im Kontext des Senderprozesses, wenn der Übertragungskanal frei ist, d.h. die Sperrzeit abgelaufen und also die Empfangsmailboxen frei sind. Ansonsten wird die Sendewarteschlange außerhalb eines definierten Prozeßkontextes geleert aufgrund eines Sendeinterrupts.

```
xmit(msg) /* eigentliche Senderoutine */
{ /* Senden, falls Cleartimer abgelaufen, ansonsten nichts! */
  s_old = spl(NEW);
  /* Interruptsperre bei der Zeitabfrage im Prozeßkontext */
    if (NextTim) {
      NextTim = FALSE;
      prepareACKs(;
      writembx(msg); /* Empfänger-Mailboxen beschreiben */
      setTimeOut(AckTimeOut); /* ACK-Zeitüberwachung starten */
      first = FALSE;
      pulseInt();
    }
  splx(s_old);
} /* xmit */

ClrTimerInt() /* TxInt, Sperrzeit abgelaufen */
{
       NextTim = TRUE;
       MaintainOutQueue();
} /* ClrTimerInt */

phintr() /* RxInt, PortInterrupt-Bearbeitung */
{
  setClrTimer(INTERVALPERIOD); /* Intervall-Timer starten */
  busywait(CARENCECOUNTER);
  for (allIntsFound) { /* Auswertung in Prioritätsreihenfolge */
    if (AckReceived()) ACKhandle(); /* vermerken in OutQueue */{
    if (MsgReceived()) { /* Msghandle() */
      AdmissibilityCheck();
      /* bei inakzeptabler Nachricht Sonderbehandlung*/
      createMemb(InQueue);
      EmptyMbx(); /* Mailbox entleeren */
      EnQueue(InQueue);
      wakeup(&InQueue); /* Empfängerprozeß benachrichtigen */
      MarkAckToBeSent();
  }
  ClearInterruptService();
} /*phintr*/
```

Abb. 32 Stark vereinfachte Treiber-Prozeduren in C-Notation (Teil B)

In diesem Fall treten Probleme im Zusammenhang mit der Realisierung des ATTEMPTO-Protokolls auf: Wollte man aus der Portinterrupt-Routine heraus Botschaften senden, würde eine Prämisse des Protokolls, nämlich daß die Zeit zur Entleerung der Mailboxen bei den Empfängern definiert werden kann, verletzt. Diese Zeit, die Sperrzeit, hängt von der maximalen Interrupt-Laufzeit ab, und damit also vom Portinterrupt selbst, wenn der Sendeinterrupt lokal zugleich Empfangsinterrupt ist. Die entstehende Rekursion macht eine zeitbegrenzte Sperrzeit unmöglich. Sende- und Empfanginterrupt müssen also getrennt werden.

Dazu bleibt die Möglichkeit, das Senden auf einen anderen Interrupt niedrigerer Priorität zu verlagern. Die kann z.B. ein Timeout-Interrupt sein, der aus der Empfangsinterrupt-Service-Routine heraus angereizt wird. Diese Methode der Verlagerung auf einen Ersatzinterrupt wird in ATTEMPTO verwendet. Sie wird hier als "*Interrupt Bootstrapping*" bezeichnet. Als Reaktion auf einen Portinterrupt (RxInt) wird als erste Aktion innerhalb der Interrupt-Service-Routine (*phintr*()) ein Hardwaretimer gesetzt (*Intervall-Timer*: 16 Bit, Maximallaufzeit ca. 25 ms). Dessen Laufzeit umfaßt Karenz- und Sperrzeit. Liegt -wie beim Experimentalsystem - die Interruptpriorität des Timerbausteins auf derselben Ebene wie der Portinterrupt, ist aber innerhalb dieser Ebene niedriger priorisiert (in der Daisy-Chain des Interrupt-Controllers), tritt auch das Problem "Interrupt bei der Zeitabfrage" nicht auf, da der Portinterrupt das Senden aufgrund abgelaufener Sperrzeit nicht unterbinden kann. In der Behandlung des Sperrzeitinterrupts (*ClrTimerInt()*, Abb. 32) wird - genau wie auch beim erstmaligen Versuch im Kontext des Sendeprozesses - die Sendebearbeitungs-Prozedur *MaintainOutQueue()* durchgeführt. Ihre Laufzeit trägt zur Karenzzeit bei. Ein Senden wird immer versucht, wenn die OutQueue nicht leer vorgefunden wird. Ausstehende ACKs verhindern das Nachrücken von weiteren Nachrichten (anderer Sendeprozesse) an den Kopf der Sendewarteschlange. Eintreffende ACKs werden vom PortInterruptHandler vermerkt (*ACKhandle()*). Ausgesendet werden eigene Bestätigungen frühestens im nächsten Sendeintervall nach Eintreffen der Nachricht als Reaktion auf eine vorher geprüfte (*AdmissibilityCheck()*) und als akzeptabel erkannte Nachricht. Daß dies erfolgen soll, wird vorab vermerkt (*MarkAckToBeSent()*). Die erforderliche Sonderbehandlung bei negativem Akzeptanztest besteht in der vorläufigen Einkettung eines Platzhalterelements in die Empfangswarteschlange und Verweigerung der Bestätigung. Eine Wiederholungsnachricht kann später dann diesen Platzhalter ersetzen. Solange dies nicht erfolgt ist, oder eine Zeitüberwachung für Wiederholungsmeldungen abgelaufen ist, kann die verstümmelte Nachricht nicht aus der InQueue entfernt werden, was zu einer zeitweisen Blockierung der Empfangsprozesse führen kann. Erwartete ACKs werden ebenfalls zeitüberwacht. Die Zeitüberwachung wird direkt nach der Botschaftsübermittlung durch die eigentliche Senderoutine (*xmit()*) gestartet (*setTimeOut(AckTimOut)*). Vor der Botschaftsübermittlung wird zunächst geprüft, ob ACKs zum Senden anstehen (*prepareACKs()*), und diese werden übersendet, gegebenenfalls im Huckepack zusätzlich zu Botschaften.

Die Karenzzeit wird abgewartet durch aktives Pollen (*busywait(CARENCECOUNTER)*) des Zählregisters im Intervall-Timer zu Beginn der Interruptbearbeitungsroutine. Da dies Prozessorleistung bindet, empfiehlt sich die sorgfältige Dimensionierung der Karenzzeit auf das absolut notwendige Minimum. Dazu stellen wir folgende Neubetrachtung an:

Wir hatten ursprünglich die Karenzzeit definiert als die maximale Verzögerung bis zum Erkennen eines Portinterrupts (Gesamt-Interruptlatenz); danach befinden sich alle Prozessoren im Zustand unterbrochen und können in gleicher Weise ihr Interruptregister auswerten. Die Karenzzeit als absolute Gesamt-Interrupt-Latenz ist zusammen mit der Sperrzeit wichtig zur Bestimmung der Intervallzeit. Diese ist die feste "Zeitschlitzbreite" für diese Art von

“asynchronem Zeitmultiplex”; sie beeinflußt damit direkt die maximale Übertragungsrate. Die Gleichbehandluung bei der Auswertung der Interrupts ist das Wesentliche zur Garantie des atomaren Rundspruchs. Eine konsistente Systemsicht über ausgelöste Sendeinterrupts wird nur dann unmöglich, wenn solche Interrupts nachzügeln, wie bereits schon erwähnt wurde. Das werden sie nur dann können, wenn Sendeaktionen schon begonnen wurden, bevor der erste Portinterrupt ausgelöst wurde. Andere (nicht Sende-) Interupts im System sowie auch andere Gründe zur verzögerten Auswertung der Interruptregister können die Empfangsreihenfolge nicht beeinflussen.

Demzufolge definieren wir die Karenzzeit neu: diese Zeit beinhaltet die Laufzeit der eigenen Portsendeinterrupt-Routine (*MaintainOutQueue()*) und die aller höherpriorisierten Interrupt-Routinen zusammen, nicht jedoch die gleichpriorer und keine kritischen Abschnitte des Betriebssystems. Diese beiden letztgenannten sind jedoch nun bei der Sperrzeit zu berücksichtigen; die Nutzbitrate auf dem Kanal bleibt natürlich unverändert.

Die so definierte Karenzzeit läßt sich meßtechnisch ermitteln. Die Laufzeit der Port-Senderoutine ist ein Parameter im eigenen Zuständigkeitsbereich und läßt sich daher optimieren. Dabei hilft folgender Ansatz: der weitaus größte Anteil an dieser Laufzeit besteht im eigentlichen Kopiervorgang. Die maximale Nutzbitrate ist dominant begrenzt durch den Datentransfer in die Mailbox/en, denn er ist abhängig von der maximalen Paketgröße und wächst zudem noch proportional zur Knotenzahl. Ansätze zur Optimierung, etwa durch verwendungsangepaßte Begrenzung (Normierung) der Nachrichtenlänge, bringen hier den höchsten Gewinn. Eine effektive Maßnahme zur Verkürzung der Karenzzeit bringt z.B. auch die Verlagerung des eigentlichen Kopierens heraus aus der Sendeinterrupt-Routine. Das Kopieren kann als DMA-Blocktransfer im Hintergrund ablaufen; seine Beendigung wird durch einen weiteren, in der Priorität wieder notwendigerweise niedrigeren Interrupt angezeigt. Innerhalb dessen Bearbeitung wird dann die nichtunterbrechbare Zeitabfrage nach Ablauf der Sperrzeit durchgeführt. Im einzelnen muß genau geprüft werden, ob sich durch die Einsparung an Rechenleistung durch Reduktion der *busy-wait*-Abfragezyklen überhaupt noch ein Gewinn ergibt. Für den Gesamtdurchsatz kann sich dieser “Dreifach-Interrupt-Bootstrap” wegen der höheren Anzahl an Interruptlatenzen auch negativ auswirken[1].

ACKs können, wie bereits erwähnt, gleichzeitig Elemente zur Flußkontrolle sein. Dies ist praktisch immer möglich bei Synchronbetrieb. Probleme gibt es hier allenthalben bei der Dimensionierung von Zeitschranken zur gleichzeitigen Fehlerüberwachung. Bei Asynchronität kommt noch das Problem des empfangsseitgen Puffermangels hinzu, und es werden Maßnahmen zur Verklemmungsvermeidung bzw. -auflösung erforderlich. Denkbar ist in diesem Zusammenhang z.B. eine dynamische Verzögerung des ACK-Sendens je nach Füllstand der Warteschlange auf der Empfängerseite (*weiche Flußsteuerung*) bei Verwendung von Warteschlangenelementen als Reservepuffer zur Deadlockvermeidung. Grundsätzlich müssen jedoch bestimmte Randbedingungen in Bezug auf maximale Botschaftsfrequenz, Lastverhalten der Rechnerknoten, maximale Paketlänge usw. beachtet werden, die die maximale Transferverzögerung begrenzen. Für den Fehlertoleranzbetrieb lassen sich solche einhaltbaren Grundzeitrahmen definieren. Die Botschaften sind typischerweise sehr kurz, dadurch kann eine hohe Zahl an Warteschlangenplätzen bei noch geringem Gesamtspeicherbedarf bereitgestellt werden. Da außerdem die die Sendefolgefrequenz relativ gering ist - Senden und Empfangen stehen lokal in kausaler Wechselbeziehung, und im Vergleich zu den lokalen Aktivitäten ist das

1. In der aktuellen Implementation des Porthandlers wurde wegen der relativ kleinen Paketgröße und der moderaten Systemgröße (geringe Knotenzahl) darauf verzichtet.

globale Verschicken von Botschaften seltener -, und da durch den dedizierten Unix-Prozeß PH (Port-Handler-Prozeß) eine weitere Zwischenpufferung von Empfangsnachrichten in der Handler-Pipe vorgenommen wird, erreicht der Füllstand in den Empfangswarteschlangen der lokalen Kommunikationstreiber keine besonders hohen Werte. Die Fehlertoleranz-Systemprozesse sind zudem in ihrer Priorität gegenüber den Anwenderprozessen bevorteilt[1], sodaß auch bei hoher Systembelastung durch Anwenderprozesse noch ausreichend Rechenleistung zur Entgegennahme und Bearbeitung von FTL-Nachrichten bleibt.

1. Im Rahmen der Möglichkeiten von UNIX.

5.5 Adaption an das lokale Basisbetriebssystem

Die Betriebssystemerweiterung für die Fehlertoleranz ist auf der Ebene der Anwenderprozesse angesiedelt, um die Eigenschaften der Modularität, Konfigurierbarkeit und Portabilität zu gewährleisten. Sie in einem eigenen Prozeßsystem außerhalb des Kerns zu kapseln, schafft eine weitgehende Unabhängigkeit vom Basisbetriebssystem sowohl hinsichtlich dessen Funktionalität wie auch dessen Struktur. UNIX ist bekannt als wenig änderungsfreundlich wegen seiner monolithischen Struktur, die über einen längeren Zeitraum “natürlich” gewachsen ist. Gerade bei solchen Eigenschaften ist es wichtig, die Interaktion zwischen lokalem Betriebssystem und Erweiterung auf wenige klar definierte Schnittstellen zu begrenzen. Anwenderprozesse stehen mit dem Betriebssystem über eine Systemaufruf-Schnittstelle in Verbindung; diese ist für die Anpassung an die Erweiterung von besonderer Bedeutung, da sie die am weitesten außen liegende Schnittstelle zur Erbringung der Dienstleistung des Betriebssystems für den zu überwachenden fehlertolerant laufenden Anwenderprozeß darstellt. Hier kann auf den Dienst selbst noch mit höchstem Freiheitsgrad Einfluß genommen werden. Die eigentliche Diensterbringung bleibt aber nach wie vor Aufgabe des lokalen Betriebssytems, das aber im Auftrag der höherrangigen globalen Fehlertoleranzschicht agiert. Um die entsprechende Modifikation des Dienstes durchführen zu können, kommt das Konzept der ***Umlenkung von Systemaufrufen*** zur Anwendung, siehe Abb. 33. Der Anwenderprozeß wird an seiner Systemaufrufschnittstelle überwacht. Relevante Systemaufrufe werden vom Kern der lokalen Fehlertoleranzinstanz zugeführt und dort einer Sonderbehandlung unterzogen. Die Systemaufrufschnittstelle ist ein synchrones Interface: der Anwenderprozeß wird in einen wartenden Zustand versetzt, bis der geforderte Dienst vom System erbracht worden ist. Die Verwaltung der Prozeßzustände ist die Aufgabe des lokalen Betriebssystem: ein globales Prozeß-Scheduling im Sinne von synchronisierten Zeitscheiben ist nicht vorgesehen. Der Synchroncharakter an der Systemaufrufschnittstelle bietet jedoch einen Ansatzpunkt, um den Lauf der Anwenderprozesse durch Verzögerung der Diensterbringung zu beeinflussen. Verständigen sich die beteiligten Knoten, gelangt man auf diesem Wege zu einer globalen Ereignissynchronisierung, zum Beispiel zur Synchronisation von Maskierungsvorgängen (geregelte Synchronisation, siehe Abschnitt 3.1).

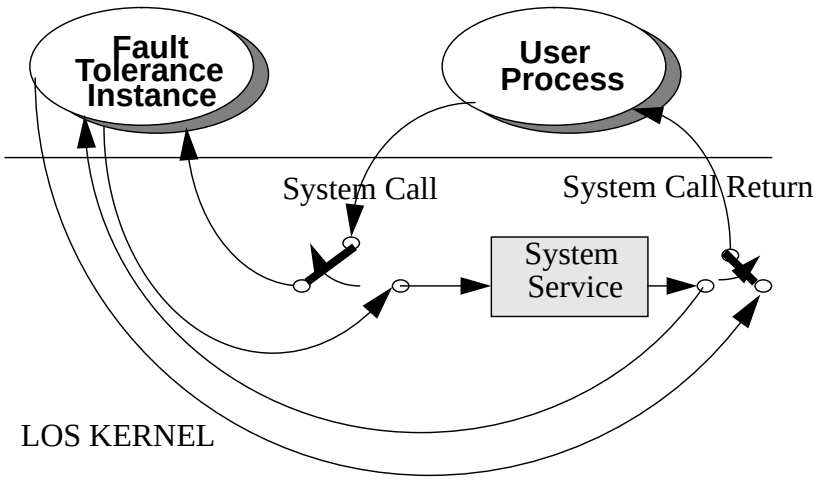


Abb. 33 Prinzip der Umlenkung von Systemaufrufen

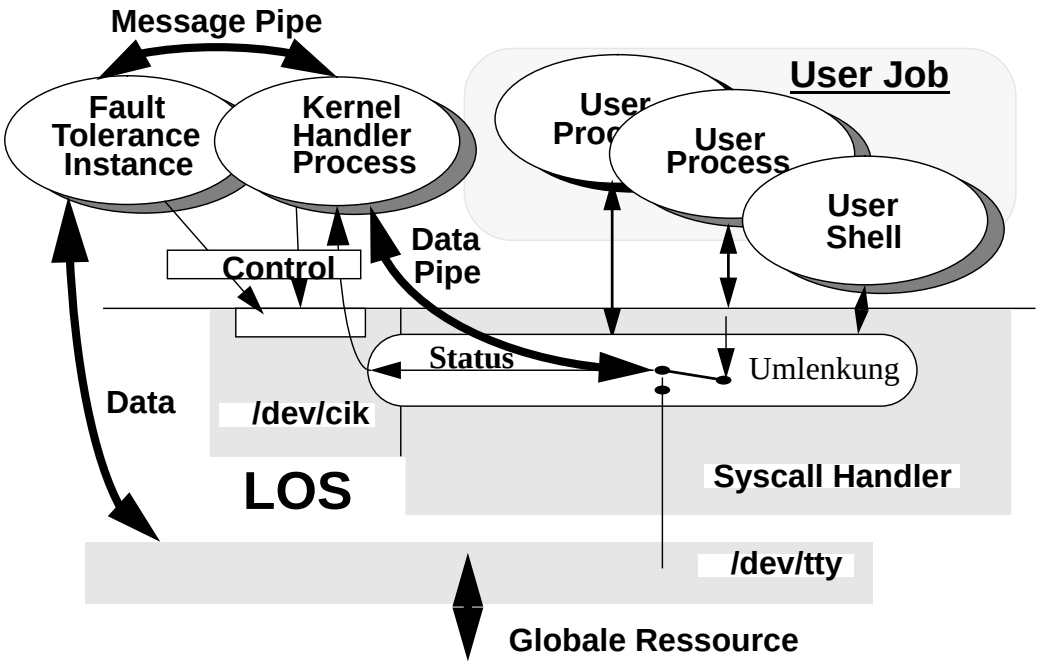


Abb. 34 Systemcall - Umlenkung und Datentransport durch die Fehlertoleranzschicht

In Abb. 33 erkennt man die Beeinflussungsmöglichkeiten auf den Prozeßablauf durch die lokale Fehlertoleranzinstanz an den Umlenkstellen; sowohl vor der Bearbeitung durch das lokale System wie auch danach ist Beeinflussung möglich.

UNIX - Systemaufrufe, deren Überwachung für den Fehlertoleranzbetrieb relevant sind, verlangen Dienste, die mit folgendem zu tun haben:

- ***Datentransport über globale Schnittstellen:*** Standard-Systemaufrufe: read, write[1]. Die Hauptaufgabe der FTL besteht hier in der Sicherung gleicher Eingabe für alle Prozeßreplikate eines fehlertolerant laufenden Jobs und der Votierung zur Fehlermaskierung im Falle von Ausgaben.
- ***Zuordnung von Kommunikationskanälen*** (dup, pipe, open, close). Alle Kommunikationskanäle eines Prozesses werden in UNIX durch einen Portadressierungsmechanismus (*file-Deskriptoren)* referenziert. Dies ermöglicht die begriffliche Vereinheitlichung von Dateien und Geräten als Grundlage für die Ein/Ausgabe-Umleitung (*I/O redirection* [Bach86]). Die Fehlertoleranzschicht muß bei allen Kanal-Zuordnungsfunktionen überprüfen, inwieweit globale Ressourcen (als Geräte) betroffen sind. Der Bezug der Dateideskriptoren zu Objektnamen muß verwaltet werden. Dazu braucht jede lokale Fehlertoleranzinstanz ein Abbild der prozeßbezogenen Dateiindextabelle (*process file table* [Bach86]).
- ***Veränderung von Betriebsparametern auf globalen Schnittstellen*** (ioctl,fcntl). Die Verwaltung muß auf Fehlertoleranzsystemebene analog zu der in den lokalen Gerätetreibern nachvollzogen werden. Dies stellt ein erhebliches Problem dar, weil Geräteeigen-

1. Andere sind entsprechend vorhandener (Interprozeß-)Kommunikationsmöglichkeiten denkbar, z.B. *send, recv* bei BSD-Sockets. Sie sind dann von Bedeutung, wenn die Netzschnittstelle eine globale Ressource ist. Generell sind im weiteren Verlauf dieser Klassifizierung nur die typischen Vertreter für die jeweilige Klasse von UNIX-Systemdiensten aufgelistet.

schaften keiner Normung unterliegen, und somit möglicherweise Hardwareabhängigkeiten in der Fehlertoleranz-Software Berücksichtigung finden müssen. Die Modulkonfigurierbarkeit begünstigt jedoch die Handhabung unterschiedlicher Softwareversionen.

- ***Veränderung der Prozeßlandschaft*** (fork, exec, exit). Alle Prozeßverwaltungsfunktionen des lokalen Betriebssystems müssen auf FTL-Ebene nachvollzogen werden, um den Kontakt zu der Prozeßmenge des überwachten Anwenderjobs nicht zu verlieren. Die FTD-Clerks der FTL verwalten Jobs in Job-Kontrollblöcken (JCB). Lokale Prozeßbezeichner (PIDs: Prozeßnummern) sind globalen Jobnummern zugeordnet, um sie systemweit einheitlich referenzieren zu können. Ein JCB wird angelegt bei Aufruf des Jobnamens durch den Benutzer und existiert solange in der System-Jobliste (JCB list), bis alle Jobkollegen (Replikate) mit ihrer gesamten Prozeßfamilie gestorben sind.
- ***Lokale Informationsdienste*** (getpid,time,..) ***und Signalbearbeitung*** (signal, kill). Hier sind Absprachen zwischen den beteiligten Fehlertoleranzinstanz-Kollegen erforderlich, um den deterministischen Programmlauf des replizierten Anwenderjobs nicht zu gefährden, wie im Kapitel 3.2. bereits näher beschrieben wurde.

Systemaufrufe in UNIX kennzeichnen den verlangten Dienst mit einer Nummer und transportieren zusätzliche aufrufspezifische Parameter. In der vorliegenden UNIX-Implementation sind dafür bestimmte CPU-Register reserviert. Sie tragen i.d.R. auch die Rückgabewerte. Daneben gibt es noch zusätzlich Variablen in der Prozeßlaufzeitumgebung, wie z.B. *errno*, die den Fehlerstatus von Dienstaufrufen für den Anwender bereithält. Sollen größere Datenmengen zwischen Betriebssystem und Prozeß ausgetauscht werden, z.B. Lese- oder Schreibdaten von/zu offenen Kanälen oder Objektbezeichner wie Dateinamen, bezeichnen solche Parameter Adressen von Datenpuffern im Anwender-Adreßbereich. Da es sich um für die Fehlertoleranzschicht relevante Daten handeln kann (zu votierende Ausgabedaten, Dateinamen, usw.), benötigt die FTI einen Kanal zum Absaugen dieser. Dieser ist eine Pipe, in die das lokale Betriebssystem die Userdaten umlenkt. Sie ist in Abb. 34 dargestellt (Data Pipe).

Für die Steuerung der Umlenkung, sowohl des Systemaufrufes wie der Anwenderdaten, ist ein separates Interface (Control) zu einem Pseudo-Gerät /dev/cik (communication instance kernel) vorgegeben. Dieser Gerätetreiber ist eigentlich eine Erweiterung des vorhandenen *System Call Handlers*, eine Betriebssystem-Routine von zentraler Bedeutung im UNIX-Kern. Hier treten Anwenderprozesse in den Kernmodus über (durch einen *Software-Trap*), bzw. wieder zurück in den Anwendermodus, hier befindet sich die zentrale Schaltstelle für Prozeßwechsel und münden Signale in den Programmfluß. Wenngleich der Code nur von geringem Umfang ist (ca. 2 Bildschirmseiten C-Quellcode), erfordert eine Modifikation an dieser Stelle eine weitreichende Einsicht in die Mechanismen der UNIX-Prozeßverwaltung. Wie man aus Abb. 35 erkennt, bleibt die Änderung aber auf den einzigen Teil beschränkt, der sich mit der Abarbeitung des eigentlichen Systemdienstes befaßt. Die Struktur entspricht also dem in Abb. 33 dargestellten Prinzip.

Zur gesteuerten Abarbeitung des umgelenkten Systemaufrufes durch den Kernel-Handler Prozeß KH ist ein leistungsfähiger Mechanismus vorgesehen, der auf dem Austausch von Botschaften - Kommandos an das Kernel-Device und dessen Rückmeldungen an den KH-Prozeß - beruht[1].

1. Ein ähnlicher Mechanismus wurde bereits in [Schm85] vorgestellt.

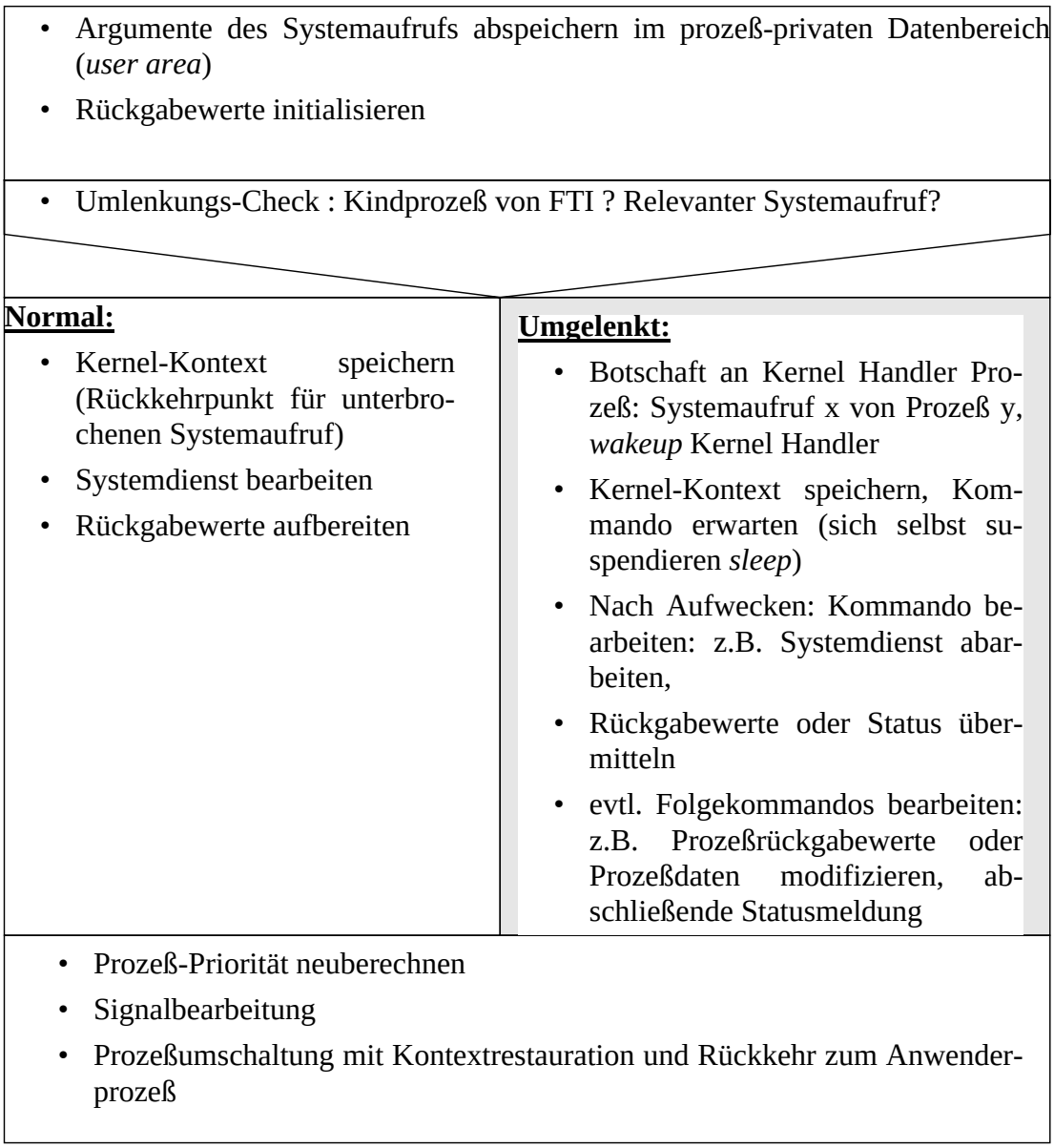


Abb. 35 Programmstruktur des modifizierten System Call Handlers

Die eigentliche Interaktion zwischen Fehlertoleranzinstanz (repräsentiert durch den KH-Prozeß) und überwachtem Anwenderprozeß besteht in der Bereitstellung der Botschaften in einem dem Pseudogerät /dev/cik zugeordneten Pufferreservoir und dem expliziten Aufwecken des Empfängerprozesses. Eine besondere Eigenschaft des Mehrprozeßbetriebs unter UNIX ist, daß Kerndienstleistungen im Prozeßkontext erbracht werden [Bach86]. Device-Treiber sind daher wiedereintrittsfähig (*re-entrant*) zu konstruieren. Der Kernel-Handler schläft in der *read*-Systemroutine des Kernel-Devices (*sleep*), darauf wartend, vom Anwenderprozeß selbst nach Bereitstellung der entsprechenden Informationen durch den modifizierten *System Call Handler* aufgeweckt zu werden (*wakeup*, s.u.). In umgekehrter Richtung führt das Auslösen eines Kommandos durch den KH-Prozeß (durch einen Schreibvorgang auf das Kernel-Device) zu einem Aufwecken des jeweiligen Anwenderprozesses. Anwenderprozesse werden durch ihren Prozeßkanal addressiert. Dieser kennzeichnet den Tabelleneintrag für den Prozeß in der System-Prozeßtabelle.

Der Trap-Mechanismus verbunden mit Speicherschutzvorkehrungen durch die MMU schottet das Betriebssystem von den Benutzerprozessen ab. Weil dies nach wie vor der einzige Zugang bleibt, erstreckt sich dieser Schutz auch auf die Betriebssystemerweiterung. Der neugeschaffene Zugang über das Kernel-Device wird nämlich außer für die beiden Systemprozesse KH und FTI durch Selbstsperrung im Device unterbunden, um hier keine zusätzliche Beeinflussungsmöglichkeit zu bieten.

Während einer Initialisierungsphase werden die entsprechenden Verhältnisse eingerichtet. Dazu stehen der Fehlertoleranzinstanz folgende *ioctl*-Kommandos zur Verfügung:

```
/* ioctl commands used by /dev/CIK: */

#define INIT 1        /* init FTL */

#define EXIT 3        /* exit fault tolerance mode */

#define ENABLE 4      /* enable diversion of syscalls */

#define DISABLE 5     /* disable diversion of syscalls */

#define SYSDIVERT 7   /* init diversion of syscall (syscall no is arg) */
```

Die Befehle ENABLE bzw. DISABLE aktivieren bzw. desaktivieren die Fähigkeit zum Umlenken von Systemaufrufen überhaupt. Die Aktivierung geschieht durch Einrichten eines Sprungs zum modifizierten Systemcall-Handler (durch *"Patchen"* des Kernel-Codes) am Einsprungspunkt des Original-System-Call-Handlers.

Mit der Initialisierung (Befehl INIT) wird das Device für zwei Prozesse (dem aufrufenden und dem im System als nächstes geborenen) geöffnet, und für alle weiteren gesperrt. Gleichzeitig wird der Kernelhandler (sein Prozeßkanal) als Adressat für Botschaften vom Kernel-Device vermerkt, und er wird als Urahne aller zu überwachenden Anwenderprozesse fixiert, indem in seinem *User Record* eine entsprechende Notiz vorgenommen wird. Diese Notiz wird allen Kindern vererbt. Sie dient dazu, Prozesse kenntlich zu machen, für die eine Umlenkung in Frage kommt: die Systemaufrufe der Systemprozesse, deren Entstehungszeitpunkt früher liegt, sollen ja nicht selbst überwacht werden.

Im Zuge der Systeminitialisierung kann der Kernelhandler dem Kernel-Device mit dem Befehl SYSDIVERT angeben, welche Systemaufrufe überhaupt zu ihm umgelenkt werden sollen. Dies schafft vor allem eine Entkopplung von der Kern-Software während der Test- und Entwicklungsphase und verhindert das zeitaufwendige Neugenerieren des UNIX-Kerns[1].

Die Grundfunktionen des KH sowie seine Programmstruktur sind einfach wie bei allen übrigen ATTEMPTO-Device-Handlern auch: Ereignisregistrierung und Datenweitertransport in Form von Botschaften über die *Message Pipe* (siehe Abb. 34) zum lokalen FTI-Prozeß. Je nach umgelenktem Systemaufruf sind jedoch bestimmte Dienste vom KH eigenständig abzuwickeln. Müssen Aktionen des Anwenderprozesses global synchronisiert werden, wird der überwachte Prozeß bis zur Erledigung angehalten, und die Rückkehr in den Anwendermodus wird dann vom FTI-Prozeß selbst gesteuert.

1. Device-Treiber sind i.a. statische Bestandteile des UNIX-Kerns. Änderungen sind mit einem kompletten Linklauf verbunden und werden durch einen Reboot installiert.

Die Botschaften zum Kernel-Device sind kurz und haben eine einheitliche Struktur:

```
struct CIK_messages {
   int m_Epid;          /* PID (Prozeßkanal) des Empfaengers.0=frei */
   int m_Spid;          /* PID (Prozeßkanal) des Senders */
   int m_buf[MLENGTH];  /* Puffer fuer die Nachricht */
   int m_len;           /* Laenge der Nachricht in m_buf*/
};
```

Das Nachrichtenfeld trägt die eigentlichen Kommandos an das Kernel-Device. Hier wurde vor allem auf hohen Freiheitsgrad Wert gelegt, um die erforderlichen Sonderbehandlungen für alle möglichen Systemaufrufe geeignet durchführen zu können. Abb. 35 listet die Kommandos auf.

.

NOTIFY /* fuehre Systemaufruf aus + sende Ergebnis an FTL */
Parameter: Nummer des Original-Systemaufrufes
Kommentar: Führe Original-Systemaufruf im Kontext des Sohnprozesses aus, melde die Rückgabewerte davon der FTL, und lasse den Sohn ohne weitere Interaktion mit der FTL fortfahren.

SELF /* fuehre Systemaufruf selber aus */
Parameter: Nummer des Original-Systemaufrufes
Kommentar: Führe Original-Systemaufruf im Kontext des Sohnprozesses aus und lasse den Sohn ohne weitere Interaktion mit der FTL fortfahren,

DIVWRITE /* lenke Schreibdaten zur FTL um */
Parameter: Nummer des *write*-Systemaufrufes (4),
Kommentar: Führe write-Systemaufruf im Kontext des Sohnprozesses aus aber mit Ausgabe zur FTL-Pipe. Der Original-Filezeiger wird gespeichert und nach dem Schreiben wieder in der Prozeß-File-Tabelle restauriert. Gebe Rückgabewerte zur FTL und erwarte weitere Kommandos.

DIVREAD /* lese nicht aus *user file* sondern aus FTL-Pipe */
Parameter: Nummer des *read*-Systemaufrufes (3),
Kommentar: Führe Original-read-Systemaufruf durch aber mit Eingabe von der FTL-Pipe. Fahre fort ohne Rückgabewerte zu übermitteln oder weitere Kommandos abzuwarten.

VRETURN /* gebe Rückgabewerte eines durch die FTL initiierten Systemaufrufs */
Parameter: keine
Kommentar: fahre dann fort.

RETURN /* beende Systemaufruf */
Parameter: keine
Kommentar: fahre fort, d.h. kehre zum Sohn-Prozeß zurück.

(Fortsetzung auf der nächsten Seite)

EXECUTE /* uebernehme Argumente aus Nachricht und führe Systemaufruf aus */
Parameter: Systemaufrufargumente
Kommentar: Dieses Kommando benutzt die FTL, um Systemaufrufe zu diktieren. Liefert Rückgabewerte (R0 und R1) zurück, und läßt den Sohn direkt fortfahren.

SENDBUF /* sende Puffer per *write* in die FTL-Pipe */
Parameter: Nummer des *write*-Systemaufrufes (4),
File-Deskriptor der Daten-Pipe zur FTL,
Datenpufferaddresse des Sohns (vorher extrahiert aus SystemCall-Meldung),
Anzahl der Pufferzeichen (falls bekannt,
sonst 0 = Puffer ist Null-terminiert)
Kommentar: Dieses Kommando benutzt die FTL, um Datenpuffer (z.B. Pfadnamen) des Sohnprozesses zu examinieren. Das Schreiben in die FTL-Pipe wird veranlaßt durch Simulation eines write-Systemaufrufs des Sohns durch Änderung von dessen Systemaufruf-Argumenten. Der Original-Systemaufruf muß dann durch die FTL per Befehl EXECUTE durchgeführt werden, um die alten Argumente wieder zu restaurieren.

SELFREAD /* fuehre READ aus. Nächster Befehl von FTL: SENDEBUF */
Parameter: Nummer des *read*-Systemaufrufes (3),
File-Deskriptor der Daten-Pipe zur FTL,
Datenpufferaddresse des Sohns,
Anzahl der Pufferzeichen
Kommentar: Führe einen read-Systemaufruf durch mit Argumenten wie o.a. (sie entsprechen i.d.R. denen des Original-Systemaufrufes). Erwarte nächstes FTL-Kommando. Die Lesedaten gelangen zum Sohn-Prozeß. Die FTL kann sie mit Befehl SENDBUF ebenfalls lesen.

SENDPID /* fuehre Systemaufruf aus und sende als 2. Wert PID */
Parameter: Nummer des Original-Systemaufrufes
Kommentar: Führe Original-Systemaufruf im Kontext des Sohnprozesses aus, gebe Rückgabewerte zusammen mit Sohn-PID und erwarte weitere Kommandos von der FTL.

Abb. 35 Kommandos an der Kontroll-Schnittstelle zum Kernel-Device

Damit ist ein Kommunikationsmechanismus gegeben, der es gestattet, alle Systemaktivitäten des Anwenderpozesses zu überwachen, das Ergebnisverhalten von Systemdiensten zu verändern und darüberhinaus sogar dem Anwenderprozeß fremde Aktivitäten aufzuzwingen. Welche Systemdienste wie für den Fehlertoleranzbetrieb zu beeinflussen sind, ist jedoch im hohen Grade systemabhängig.

5.6 Fehlertoleranzmechanismen

Unter Fehlertoleranzmechanismen sollen für den weiteren Verlauf alle Systemverwaltungsprozeduren verstanden werden, die den Fehlertoleranzbetrieb direkt oder indirekt unterstützen. Hauptaufgabe ist natürlich das Verfahren, das zur Fehlerunterdrückung angewendet wird, da es den Schlüssel zur Erhöhung der Systemzuverlässigkeit darstellt. Daneben, und auch als Voraussetzung für die Fehlermaskierung, wie z.T. schon im Kapitel 3. aufgezeigt wurde, bedarf es jedoch noch der Erledigung anderer Aufgaben. Eine Redundanzverwaltung ist nötig, die die Aufteilung von Benutzerjobs und deren ggfs. notwendige Replizierung auf die Rechnerknoten des Systems vornimmt. Weiterhin müssen Benutzereingaben zu den jeweiligen Jobexemplaren geleitet werden - dabei ist insbesondere darauf zu achten, daß der deterministische Gleichlauf der Jobreplikate nicht gefährdet wird -, und in umgekehrter Richtung gilt es, den von möglichen Fehlern unbeeinflußten Ergebnisdatenstrom an die Systemumwelt auszugeben, und diese Ausgabe mit der von anderen parallellaufenden Jobs zu koordinieren. Wichtigstes Realisierungsprinzip für alle diese Mechanismen ist die Dezentralisierung: es werden ausschließlich verteilte Algorithmen verwendet, und es gibt keine zentral gelagerten Systemdaten. Die Befähigung des Kommunikationssystems für einen reihenfolgekonsistenten Rundspruch hilft dabei, die Algorithmen verhältnismäßig einfach zu gestalten.

5.6.1 Jobverteilung und -replizierung

Die Zuordnung von Benutzerjobs zu Rechnereinheiten ist gemeinsame Aufgabe aller Fehlertoleranz-Dispatcher FTD der Fehlertoleranzschicht. Anforderungen des Benutzers für einen Programmlauf und jobzugehörige systemglobale Statusinformationen werden in sogenannten Job-Kontrollblock-Listen (JCBlist) gehalten, die von den einzelnen FTDs synchron abgeglichen werden. Diese Listen sind einfache Warteschlangen und werden nach einer FCFS-Strategie geleert. Die Jobverteilung basiert auf dem völlig dezentralen, softwareimpementierten Prinzip der Job-Anziehung (***Job Attraction***). Um einen Job zugeteilt zu bekommen, müssen sich alle freie Rechnerknoten um diesen bewerben, wobei die bei allen gleiche Reihenfolge der einlaufenden Bewerbungsbotschaften den Ausschlag gibt.

Bevor dieser Wettlauf beginnen kann, muß die Eingabe des Benutzers daraufhin untersucht werden, ob dieser einen neuen Job zu starten beabsichtigt, und welche Zuverlässigkeitsanforderungen er dafür stellt. Diese Analyse-Aufgabe fällt dem Terminal-Handler-Prozeß TH zu. Zur Wahl der Zuverlässigkeit ist die Kommando-Shell-Syntax erweitert worden um einen Parameter ***Fehlertoleranzgrad***. Er gibt die Anzahl der Jobexemplare an, deren Ausfall toleriert werden kann, ohne die Jobresultate zu verfälschen. Ein Job “meinprogramm” wird durch den Aufruf “meinprogramm #t#” als fehlertolerant mit dem Grad t definiert. Auf diese Weise kann man unterscheiden zwischen nicht fehlertoleranten Jobs (t=0), Fehlermaskierung mit $t \geq 1$ tolerierbaren unabhängigen Fehlern, und reiner Fehlererkennung (mit dem Aufrufparameter t=-1) für einen Fail-Save-Betrieb. Die Anzahl der kooperierenden Jobreplikate, im weiteren Kollegen genannt, hängt vom verlangten Fehlertoleranzgrad und dem gewählten Diagnosemodell ab.

Weitere Jobverwaltungsaufgaben betreffen das lokale Starten von Jobs und die globale Überwachung von deren Terminierung. Abb. 36 soll dazu dienen, die über die Lebenszeit eines Jobs anfallenden Verwaltungsaufgaben zu illustrieren. Die schon erwähnten vier Phasen sind darin als

Eingabeanalyse, Job-Anziehung, Jobstart, Jobbeendigung

voneinander unterscheidbar. Die erforderlichen Aktionen werden arbeitsteilig von den Systemprozessen vorgenommen und mithilfe von Botschaften koordiniert. Wegen des homogenen Botschaftsformats ist es in der Darstellung nicht erforderlich, zwischen Schwer- und Leichtgewichtsprozessen zu unterscheiden.

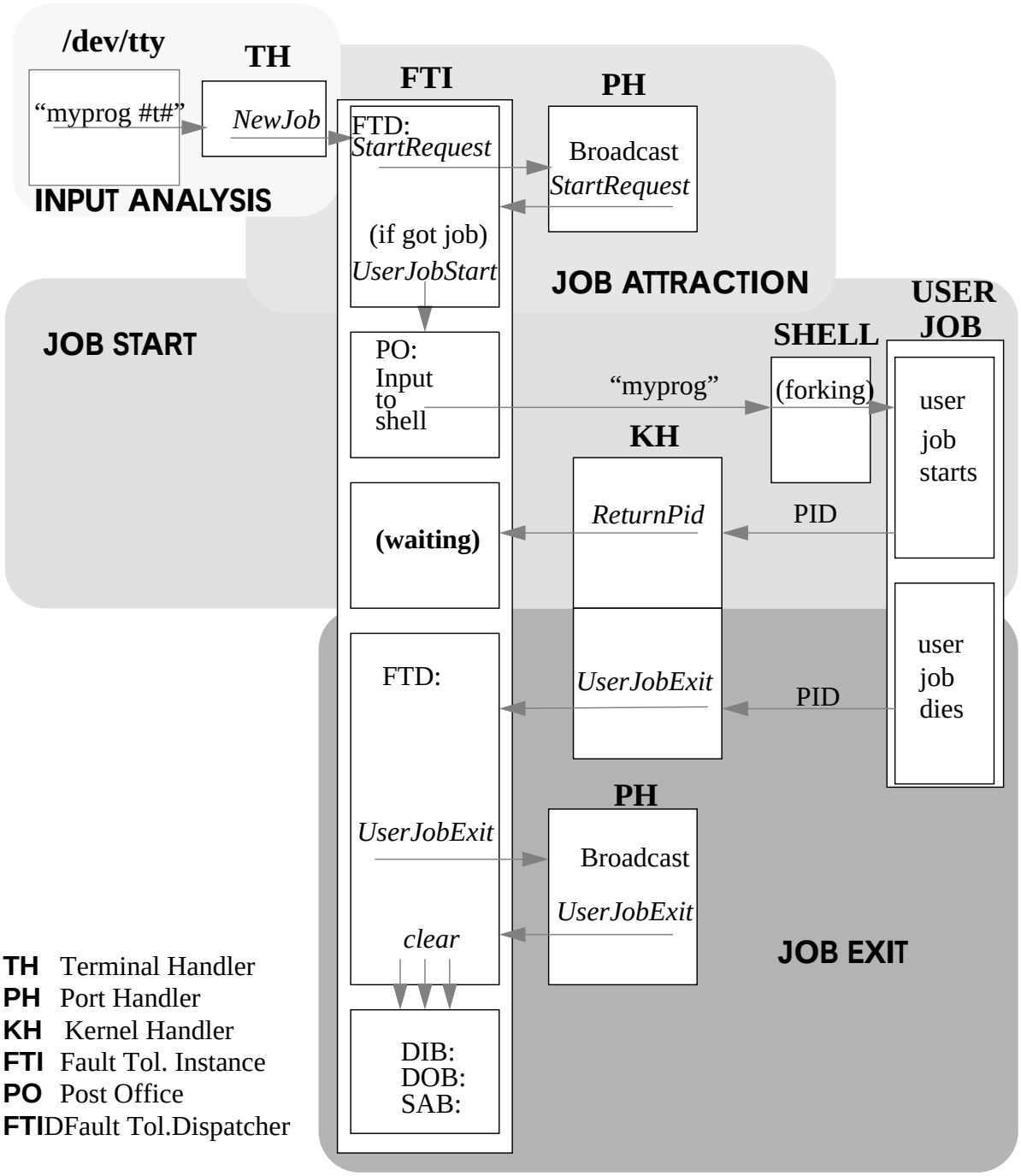


Abb. 36 Meldungsverkehr zur Jobverteilung

Die Eingabedaten des Benutzers werden den Terminal-Handlern aller Knoten gemeinsam zugeführt. Haben sie ein Benutzerkommando erkannt, formatieren sie eine Nachricht *NewJob* und senden sie zum FTD-Clerk des lokalen FTI-Prozesses. Die Terminal-Handler können ein Kommando von normalen Jobeingaben unterscheiden, da eine Konvention zur Identifikation

getroffen wurde: alle Eingaben tragen eine Kennzeichnung für den Job, für den sie bestimmt sind. Die Kommandoshell des Benutzers ist ein Job mit der festen Identifkationsnummer 0. Die Kennzeichnung geschieht entweder durch einen Präfix, den der Benutzer selbst eingeben muß, oder aber, in Verbindung mit einer graphischen Benutzeroberfläche, automatisch durch das Fenstermanagement im Endgerät (globale Ressource), das die Zuordnung von Fenstern zu Jobs verwaltet (Siehe dazu Kapitel 6.3).

Der FTD-Clerk erzeugt einen neuen Job-Kontrollblock (JCB) und trägt dort Programmnamen und Fehlertoleranzgad ein, die er der *NewJob*-Nachricht entnommen hat. Der neue Job erhält eine einzigartige globale Job-Identifizierungsnummer (JID). Der JCB wird darauf in die JCB-list eingereiht. Falls die Replikationserfordernisse erfüllt werden können, bewerben sich freie Knoten im System mit einer Broadcast-Nachricht *StartRequest*, die an alle FTD-Clerks im System, also auch an den Bewerber selbst gerichtet ist. Nach Erhalt einer solchen Botschaft, die vom Port-Handler-Prozeß PH entgegengenommen und der FTI (über die Handler-Pipe) zugeführt wurde, kennzeichnen die FTDs in ihrem JCB-Eintrag die Bewerber als Bearbeiter, falls die Maximalzahl von Jobkollegen noch nicht überschritten ist. Im Falle, daß die Kollegenliste komplett ist, werden weitere Bewerbungen für diesen Job ignoriert. Die zu spät gekommenen Bewerber nehmen dies zum Anlaß, sich für den nächsten Job zu bewerben, falls in ihrer JCBlist noch weitere Einträge existieren.

Wenn ein FTD-Clerk einen Job für den Knoten erlangen konnte, veranlaßt eine *UserJobStart*-Nachricht das Post-Office dazu, mithilfe der Kommando-Shell des Benutzers den Job lokal zu starten. Der Standard-Eingabekanal der Anwenderprozesse ist zum Post-Office hin umgelenkt. Da die Shell selbst nichts weiter als einer von diesen Anwenderprozessen ist[1] - als solcher wird er auf seine Systemaufrufaktivitäten hin überwacht - läßt sich ein neuer Job einfach dadurch starten, daß das Post-Office die ursprüngliche Kommando-Zeichenfolge, vom Fehlertoleranzgrad befreit, in den Anwender-Eingabekanal (User Pipe) schreibt. Der *fork()*-Systemaufruf der Shell wird vom Kernel-Handler erkannt. Im Zuge von dessen Bearbeitung wird die lokale Prozeß-ID des Anwenderjobs (des Shell-Prozeß-Kindes) extrahiert und in einer Nachricht *Return-PID* dem FTD-Clerk mitgeteilt, der auf diese Weise die Zuordnung von globaler JID zu lokaler PID verwalten kann. Diese Aufgabe kann nicht dem KH-Prozeß überlassen werden, da er wegen seiner nur eingerichteten Nachrichtenverbindung zur Fehlertoleranzinstanz keine Kenntnis über globale Systemdaten hat. Die FTL wartet nun auf weitere Aktionen des Anwenderjobs.

Der JCB-Eintrag verbleibt solange in der Jobwarteschlange, bis *UserJobExit*-Meldungen von allen beteiligten Kollegen beim FTD-Clerk eingetroffen sind. Sie kommen entweder von den entfernten Kollegen über den Port-Handler oder aber vom lokalen Kernel-Handler, nachdem er den *exit()*-Systemaufruf des Anwenderjobs bei dessen Terminierung erkannt hat. Danach müssen noch alle möglicherweise noch vorhandenen und sich auf den Job beziehenden Listeneinträge bei anderen Clerks gelöscht werden. Dazu dienen die Nachrichten *ClearDIB*, *ClearDOB*, *ClearSAB* an die jeweiligen Clerks.

5.6.2 Eingabeverteilung zu den Jobs

Die Verwaltung des Eingabepuffers DIB übernimmt der gleichnamige Clerk. An der Bearbeitung von Eingabedaten sind außerdem noch der Terminal-Handler und der Kernel-Handler beteiligt. Die vom physikalischen Gerät kommenden Eingabedaten werden im TH-Prozeß als

1. Vorteilhaft an diesem Verfahren ist, daß der Benutzer seine persönliche Shell unverändert übernehmen kann.

Eingabedaten für einen bestimmten Job identifiziert, und in eine Botschaft *UserInput* verpackt an den DIB-Clerk geschickt. In der lokalen Fehlertoleranzinstanz sind drei Phasen der Bearbeitung zu erkennen (Abb. 37). Die Zwischenlagerung (Store) im DIB aufgrund der *UserInput*-Nachricht, das Eintreffen einer Leseanforderung vom Anwenderprozeß (Require) und die eigentliche Datenübergabe an den Anwenderprozeß durch Überführung der Daten aus dem Eingabepuffer an die User-Pipe (Retract). Die Leseanforderung wird bei einem Anwender-*read()*-Systemaufruf durch den Kernel-Handler in einer *UserJobRead*-Botschaft angezeigt. Bedingt durch den Asynchronlauf der beteiligten Prozesse, und weil Benutzerdaten jederzeit unkorreliert mit dem Anwender-Programmlauf über die Leitung eintreffen können, kann sich die Reihenfolge der Store- und der Require-Phase umdrehen. Die Eingaben des Benutzers werden aufgrund der mitgelieferten JobID als normale Jobeingaben identifizierbar. Die beim Terminal-Treiber /dev/tty einzeln eintreffenden Zeichen werden in der Basisversion des Experimentalsystems dort aufgesammelt und als ganze Zeilen dem TH-Prozeß übergeben. Diese starke Vereinfachung vermeidet die Gefährdung des Gleichlaufs der Anwenderprozeßreplikate aufgrund von indeterministischen Eingabedaten (siehe Kapitel 3.2), verhindert aber dadurch interaktive Anwendungen. Die damit verbundene Problematik der Implementierung wird in Kapitel 5.8 untersucht.

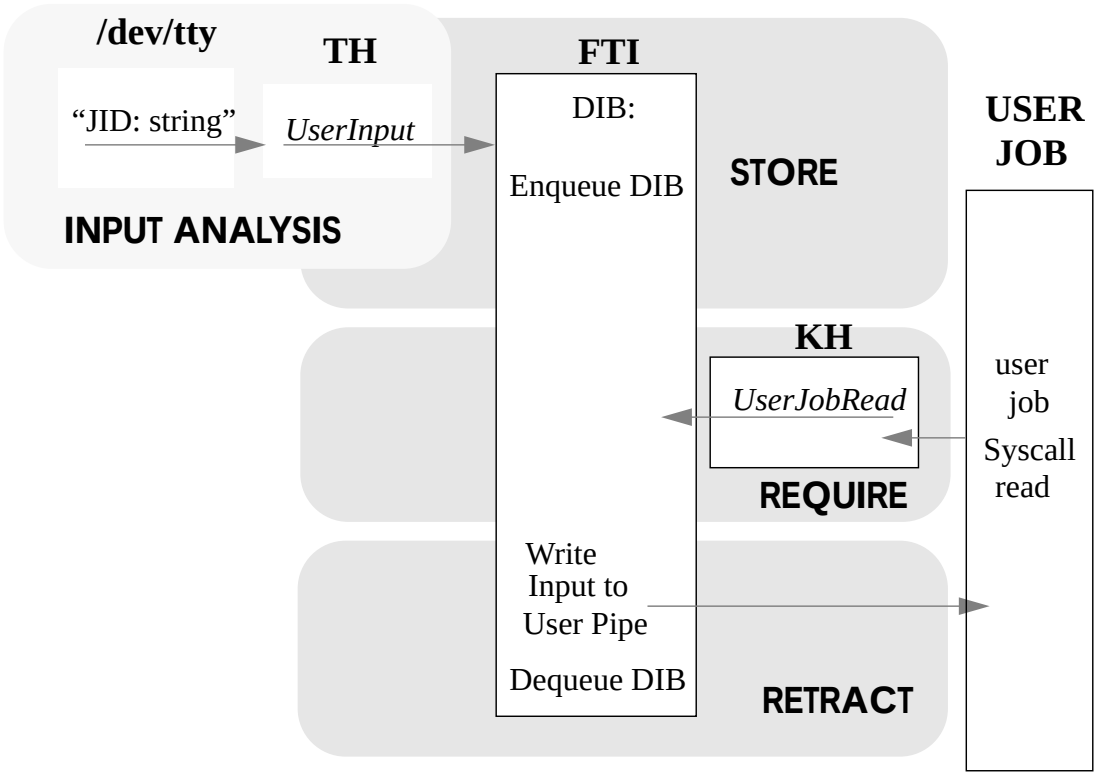


Abb. 37 Botschaftstransfer zur Behandlung der Job-Eingabe

5.6.3 Ressourcenverwaltung

Der Ressorce-Manager RM hat die Aufgabe, Ausgabekollisionen auf die globalen Ressourcen zu verhindern. Die zentrale Datenstruktur in diesem Modul ist die für jede globale Ressource im System separat gehaltene *RM-Liste*, eine Verwaltungsdatenstruktur mit Puffer für auf dem jeweiligen Ressourcen-Bus auszugebende Daten. Ausgabedaten gelangen vom DOB-Clerk mithilfe der Nachricht *OutputData* zum RM. Bevor die Daten physikalisch ausgegeben werden

dürfen, muß zunächst der alleinige Zugriff auf die Ressource abgesichert werden. Dieser Mechanismus des ***Resource Locking*** bedient sich wieder des bereits bekannten Wettlaufprinzips. Stehen auf mehreren Knoten Daten zur Ausgabe bereit, gewinnt derjenige die Ressource, der sich am schnellsten mit der Broadcastbotschaft *ResRequest* um sie beworben hat. Wer gewonnen hat, kann lokal daran erkannt werden, welcher RM nach seiner Bewerbung als erster seine eigene Bewerbungsbotschaft vor denen der anderen erhalten hat. Er ist dann der erste, wenn bei Eintreffen seiner Nachricht die Ressource noch als frei gekennzeichnet ist. Der Datenpuffer der RMliste wird daraufhin vollständig geleert. Abschließend wird die Ressource mit *ResRelease* wieder allen zur Verfügung gestellt.

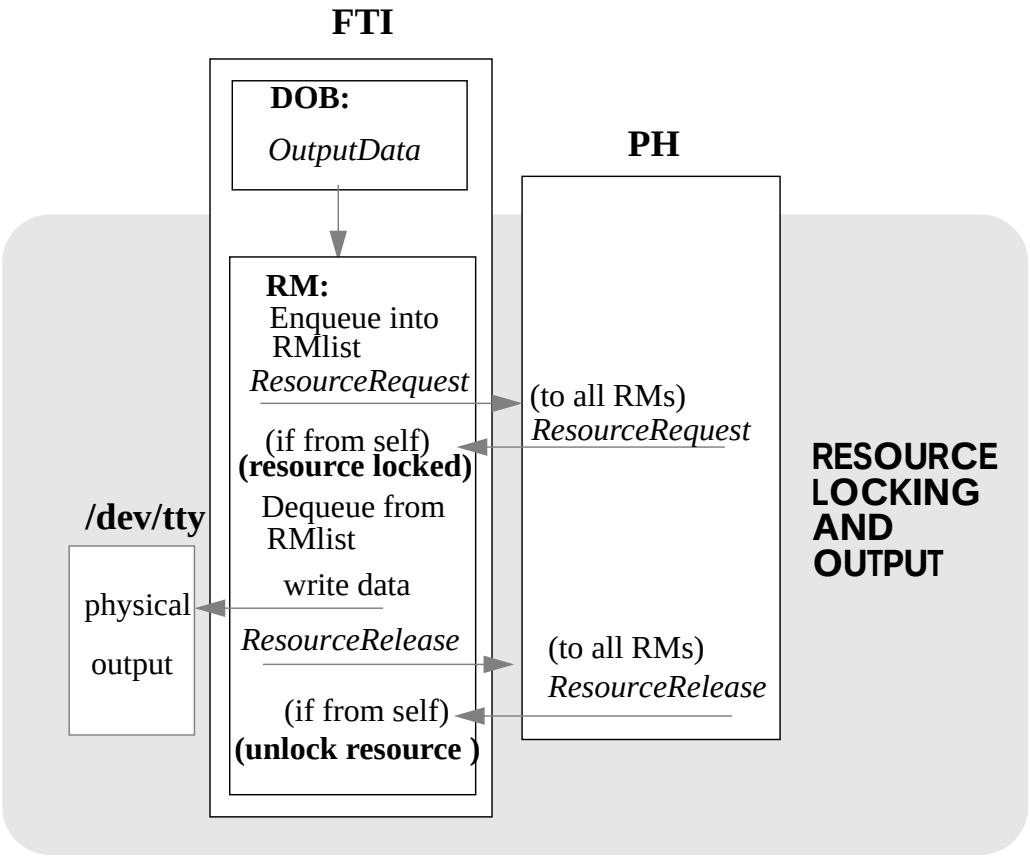


Abb. 38 Ausgabekoordination auf globalen Ressourcen (Beispiel asynchrone Terminalleitung)

Dieser einfache Grundmechanismus ist in der Praxis jedoch mit Problemen verbunden. Diese werden in Kapitel 5.8 behandelt.

5.6.4 Fehlermaskierung

Zur Fehlerunterdrückung müssen die kooperierenden Knoten (Job-Kollegen) sich synchronisieren, und daraufhin muß ein Votierungsvorgang stattfinden. Anstelle etwa eines Mehrheitsentscheides kommt das Verfahren ***End-to- End-Job Result Comparison*** zur Anwendung, im Kern ein verteilter Algorithmus zur vergleichstestbasierten Nachbarschafts-Diagnose, wie sie in Kapitel 4. beschrieben wurde.

Eine beabsichtigte Ausgabe auf eine globale Ressource resultiert in einer Synchronisation der replizierten Prozeßexemplare zum Austausch von Signaturen. Diese sind von den Ausgabeda-

ten durch Kompression abgeleitet. Sie haben eine normierte Länge von 32 Bit; dies trägt dazu bei, den Verkehr auf dem globalen Kommunikationsbus zu reduzieren. Es wurde eine Kompressionstechnik gewählt, die es hinreichend unwahrscheinlich macht, daß unterschiedliche Jobresultate zur selben Signatur führen [DBLDR87]. Dies ist wichtig, da die Systemzuverlässigkeit von der Vertrauenswürdigkeit der Diagnoseergebnisse direkt beeinflußt wird.

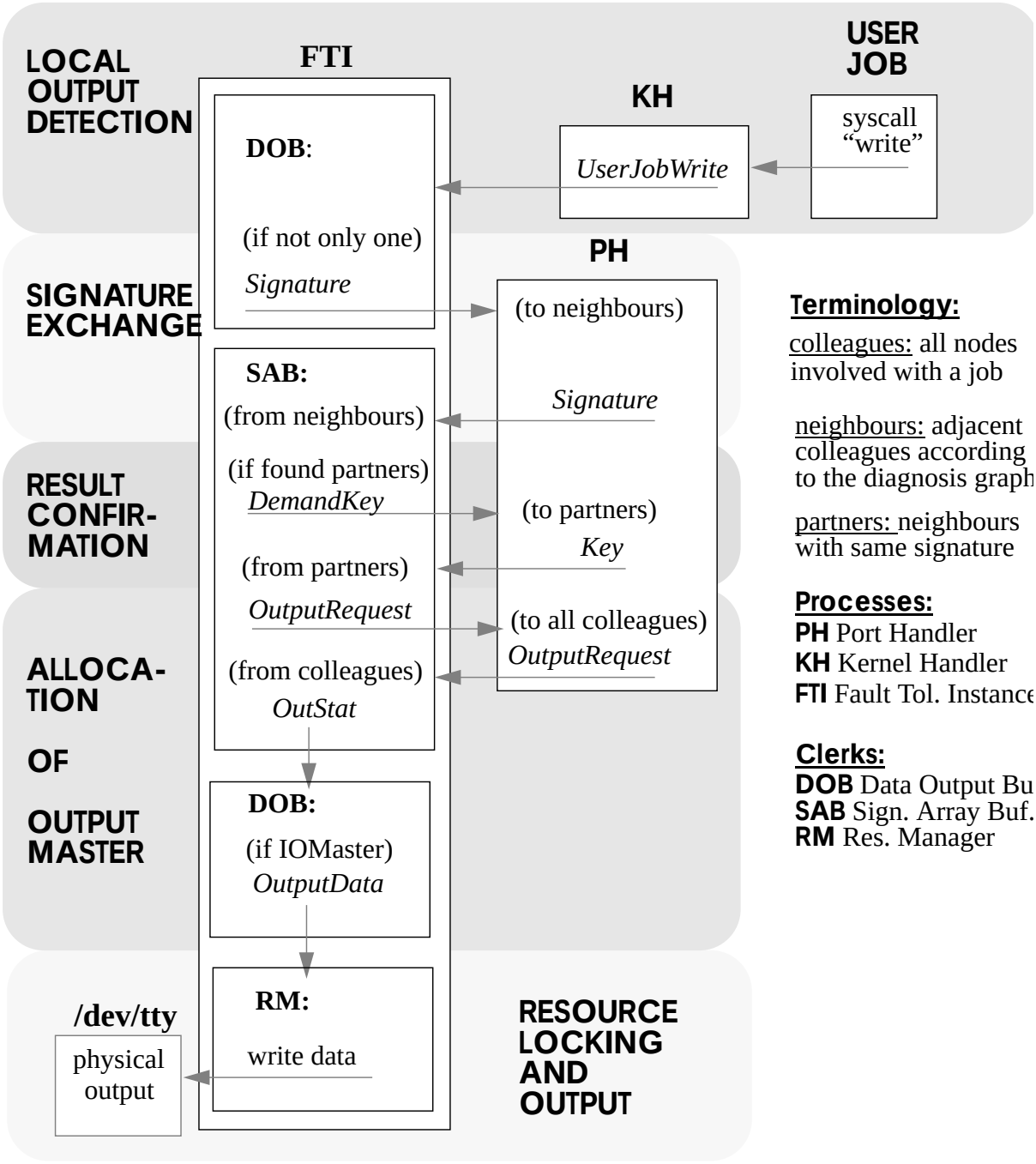


Abb. 39 Meldungsverkehr zur Fehlermaskierung

Die Abb. 39 zeigt die Interaktionen zwischen den lokalen bzw. entfernten Systemprozessen, um mögliche Fehler daran zu hindern, die Systemumwelt zu erreichen. Fünf Phasen sind erkennbar: 1. Erkennung des Ausgabewunsches, 2. Signaturaustausch, 3. Ergebnisbestätigung, 4. die Einigung für die Ausgabeberechtigung, und zuletzt 5. die Ressourcen-Reservierung und

physikalische Ausgabe.

Eine Sequenz zur Ausgabe von Jobergebnissen startet mit einem *write()*-Systemaufruf des überwachten Anwenderprozesses. Vom Kernel-Handler wird daraufhin eine Botschaft *User-JobWrite* an den lokalen DOB-Clerk verschickt. Alle Nachbarkollegen erhalten nun eine *Signatur*e-Nachricht. Die Nachbarschaftsbeziehungen sind, wie in Abb. 40 angedeutet, durch den Diagnosegraphen festgelegt. Signaturen werden vom SAB-Clerk aufgesammelt, und dort mit der eigenen verglichen. Im positiven Falle hält der Knoten sich selbst für fehlerfrei, und offeriert seinem Vergleichspartner einen Schlüssel (*DemandKey*), der dort modifiziert wird und mit der Botschaft *Key* zurückgeschickt wird. Dies schafft zusätzliche Sicherheit vor Knoten, die sich selbst fälschlicherweise für korrekt halten. Letztendlich soll eine Ausgabeberechtigung nicht ohne ausdrückliche Bestätigung durch einen anderen korrekten Knoten erlangt werden können. Die zusätzliche Sicherheit wird dadurch gewonnen, daß der Schlüssel so mit der Adresse der Ausgaberoutine kodiert ist, daß ohne die korrekte Signatur die Ausgabeprozedur nicht aufgerufen werden kann. Gleichzeitig stellt der über den Signaturaustausch hinausgehende Botschaftsverkehr als Positiv- bzw. Negativ-Bestätigung ein Grundelement des mit Zeitschranken arbeitenden Diagnosealgorithmus dar, wie in Kapitel 4.2 erläutert wurde.

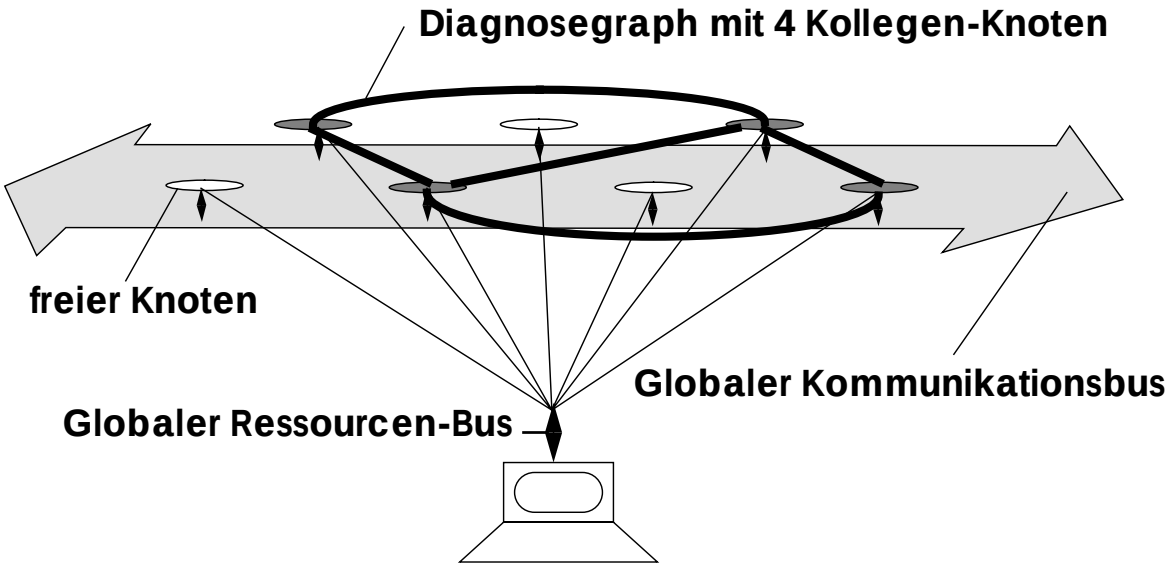


Abb. 40 Jobkollegen und ihre Diagnosebeziehungen eingebettet in ein System mit 7 Knoten

Nach dem Erhalt eines gültigen Schlüssels werden alle Kollegen davon informiert, daß der Knoten sich für berechtigt hält, die Ausgabe vorzunehmen. Dies geschieht mit der Botschaft *OutputRequest*. Der erste Knoten, der seine eigene Nachricht zurückbekommt, hat den Wettlauf um die Rolle als Ausgebender (*IO Master*) gewonnen, die anderen werden zu Beobachtern, falls sie sich als korrekt herausgestellt haben. Ein Beobachter kann die Aufgabe übernehmen, die physikalische Ausgabe des IOMasters auf Richtigkeit zu überwachen, um ggfs. nach Absprache mit den anderen Beobachtern korrigierend einzugreifen. Diese Maßnahme schafft zusätzliche Sicherheit für den Fall, daß zwischen Rollenverteilung und Ausgabe der IOMaster defekt wird, kann allerdings den Verlust der Integrität nicht verhindern. Die Korrekturmaßnahme ist nicht bei allen Anwendungen sinnvoll, und dieser Beobachtungs-Mechanismus gehört deshalb nicht zu den Grundfunktionen des Fehlertoleranzsystems. Im Experimentalsystem ist er deshalb auch nicht realisiert.

Nach der Rollenverteilung teilen SAB's ihren erworbenen Status ihren DOB's mit (Botschaft *OutStat*). Bevor nun der IO-Master-Knoten seine Ausgabe vornehmen kann, muß die globale Ressource vor dem Zugriff anderer parallel laufender Jobs geschützt werden. Dazu dient der bereits in Kapitel 5.6.3 beschriebene Ressource-Locking-Mechanismus.

5.6.5 Funktionen der Zeitüberwachung

Die zuvor beschriebenen nachrichtengesteueten Fehlertoleranzmechanismen werden durch Funktionen der Zeitüberwachung unterstützt, um Unterlassungsfehler tolerieren zu können, die sich durch Ausbleiben von Reaktionen (Nachrichten) fremder Knoten, etwa bedingt durch deren Ausfall, bemerkbar machen, und die anderenfalls zu einer Blockierung des Betriebs führen würden. Außer den Zeitüberwachungsfunktionen der höheren ATOS-Schicht FTL, die hier beschrieben werden, gibt es noch weitere in den unteren Betriebssystemschichten, die allerdings ausschließlich dazu dienen, den dort erbrachten Dienst abzusichern. Z.B. gewährleisten sie dort die geforderten Eigenschaften der Zuverlässigkeit und Reihenfolgekonsistenz für den globalen Rundspruch im Kommunikationssystem. In der FTL wird der Zeitverwaltungsdienst durch Clock-Clerks auf jedem Rechnerknoten realisiert. Zur Zeitüberwachung werden die Dienste *setTimeOut*() und *clearTimeOut()* angeboten. Die Zeitüberwachungsfunktionen stützen sich wie alle anderen System-Funktionen der FTL auf Botschaftsaustausch. Der Clock-Clerk meldet abgelaufene Zeitüberwachungen mit einer speziell markierten Botschaft dem Auftraggeber zurück, falls dieser nicht zuvor seinen Auftrag zurückgenommen hat. Mit dem Setzbefehl gibt der Auftraggeber-Clerk den Typ dieser von ihm im Falle des Ablaufs der gewählten Zeitschranke erwarteten Botschaft an; sie entspricht exakt dem Botschaftstyp der ausgebliebenen Botschaft, sodaß die Information automatisch der richtigen Bearbeitungsprozedur im Clerk zugestellt wird.

Aus Gründen der Übersichtlichkeit sind Zeitüberwachungsfunktionen in den Darstellungen Abb. 36 bis Abb. 39 weggelassen. Die wichtigsten Zeitschranken listet die Abb. 41 auf:

Mechanismus	Clerk	Aktion
Job-Dispatching	FTD	Warten auf *StartRequest* von Kollegen
Job-Dispatching	FTD	Warten auf *UserJobExit* von Kollegen
Maskierung	SAB	Aufsammeln der Signaturen
Maskierung	SAB	Warten auf Keypartner
Ressoucen-Verwaltung	RM	Überwachung der Blockade

Abb. 41 Zeitschranken

Die Dimensionierung von Zeitschranken ist bei loser Kopplung der Replikate, wie im vorliegenden Fall, mit Problemen verbunden, wenn das Anwenderprogramm-Laufzeitverhalten dem System unbekannt ist (und nach Konzeptanforderung auch sein soll). Um Maximalabweichungen überhaupt angeben zu können, muß das System eine gewisse Grundhomogenität aufweisen, sowohl in der Architektur, als auch in der Prozeßlandschaft auf den Kollegenknoten zur Laufzeit von Anwenderjobs, um die Lastverhältnisse vergleichbar zu halten. Insbesondere um letzteres kalkulierbar zu machen, ist das ATTEMPTO-System auf einen einzigen Benutzer be-

schränkt. Hintergrundjobs, wie in UNIX möglich, werden genau wie andere Anwenderjobs behandelt, also zur echt parallelen Ausführung auf Rechnerknoten verteilt. Dies reicht jedoch noch nicht aus. Die lose Kopplung macht es notwendig, für absolute Zeitschranken sehr hohe feste Werte zu benutzen, und führt so zu einer sehr zeitineffizienten Fehlererkennung. Um dies zu verbessern, kann man ***Zeitschranken*** für Jobverwaltungsfunktionen ***relativieren***. Wenigstens die individuelle Verzögerung des Starts von Jobreplikaten - besser noch die aktuelle Verzögerung bei der letzten Aktionssynchronisation - muß bei der Zeitüberwachung von deren Folgeaktivitäten mitberücksichtigt werden.

Außerdem kann es auch zur ***Akkumulation von Zeitüberwachungsperioden*** kommen, wenn es den einzelnen Replikaten erlaubt ist, mehr als einen Synchronisationsabschnitt vorauszueilen (Puffersynchronisierung mit k>1, siehe Kapitel 3.1). In ATTEMPTO erhalten deshalb *StartRequest*-Nachrichten beim Eintreffen im Porthandler-Prozeß einen Zeitstempel, um jobspezifische Zeitschranken definieren zu können, und die Puffertiefe bleibt auf k=1 begrenzt, die Synchronisation ist also relativ straff.

Das Setzen von Zeitüberwachungsschranken im Zusammenhang mit der Fehlermaskierung ist abhängig vom Maskierungsmechanismus, also dem gewählten verteilten Diagnosealgorithmus. Beim Aufsammeln der Kollegen-Signaturen wird eine Zeitüberwachung gestartet, wenn genau t Signaturen ausstehen (t ist der Fehlertoleranzgrad). Dies garantiert auf Basis des Fehlermodells, daß ein Konsens für einen Fortschritt des Programmlaufes immer möglich ist.

5.7 Fehlermodell und Fehlerbehandlung

Ein fehlermaskierendes Verfahren stellt nur geringe Anforderungen an das Fehlermodell, hauptsächlich, weil die korrekte Systemfunktion keiner ausführlichen absoluten Spezifikation bedarf, sondern Fehlverhalten relativ zum korrekten Verhalten beurteilt wird. Die Korrektheit ermittelt sich in unserem Fall aus den Fehlerannahmen im Vergleichstestmodell; die *Fehlersymptomverschiedenheit* ist hier das entscheidene Kriterium; die Fehler sind unabhängig, und zufällig korrektes oder mit anderem inkorrekten übereinstimmendes Erscheinungsbild der Fehler gilt als ausgeschlossen. Was die Natur der Fehler anbelangt, sind pauschale Fehlertypisierungen wie in Abb. 42 als Klassifizierung ausreichend.

Grundsätzlich können nur Hardwarefehler toleriert werden. Für eine maskierende Softwarefehlertoleranz, wie im *N-Version-Programming* [AvCh78], wären diversitäre Jobreplikate notwendig. Wenngleich ein solches Konzept auch leicht integrierbar wäre, müßte dafür die Forderung nach Binärcodetransparenz aufgegeben werden. Zwar ist die automatische Generierung von diversitären Programmversionen bis zu einem gewissen Grade möglich, jedoch nur quelltextnah, z.B. mithilfe eines modifizierten Compilers, oder noch eingeschränkter allein auf Hochsprachenebene durch Einsatz eines Präcompilers [Gad92].

Das Redundanzkonzept gestattet die Tolerierung von Mehrfachfehlern. Ihre Anzahl entspricht dem gewählten Fehlertoleranzgrad. In allen Fällen ist Fehlerunabhängigkeit die Voraussetzung für eine sinnvolle Maskierung. Diese Eigenschaft hängt von der Einheitlichkeit der Beeinflussungsmöglichkeiten ab. In realen Systemen wird Unabhängigkeit z.B. begünstigt durch physikalische Komponententrennung - örtlich getrennte Aufstellung, Getrenntspeisung, galvanische Kopplung im Kommunikationsmedium etc. Lokalität bewirkt allgemein Isolation und schützt so auch vor der Ausbreitung interner pyhsikalischer Fehler auf andere Komponenten. Gegen Entwurfsfehler ist bei homogenen Systemen allerdings nicht vorgesorgt, insbesondere wenn sie starr synchronisiert sind (Taktsynchronität Kapitel 3.1). Eine lose Kopplung der Replikate - wie im ATTEMPTO-System - kann hier jedoch Vorteile bringen. Insbesondere können Entwurfsfehler, die belastungsabhängig auftreten, z.B. Betriebssystemfehler beim Betrieb an Belastungsgrenzen, bei entsprechender Asynchronität toleriert werden, da hier die Wahrscheinlichkeit des gemeinsamen zeitgleichen Auftretens geringer ist.

Bzgl. der Fehlerdauer werden in erster Linie *temporäre Fehler*[1] betrachet. Dies ist gerechtfertigt, da sie weitaus wahrscheinlicher vorkommen als permanente [Siew82]; eine unmittelbare Ausgrenzung fehlerhafter Komponenten (Knotenexemplare) wird nicht vorgenommen, stattdessen werden Fehler während der restlichen Joblaufzeit nur ohne Unterscheidung in permanente oder transiente in sogenannten Fehlerfrequenzlisten notiert und wie üblich kompensiert. Häufen sich Verdachtsmomente durch Anstieg der Fehlerfrequenz, beginnt der betroffene Knoten mit Selbsttests, um möglicherweise permanente Fehler aufzudecken.

Dieses Verfahren wirkt sich bei selbstheilenden temporären Fehlern vorteilhaft aus. Dennoch müssen permanente Fehler als solche diagnostiziert werden. Zum einen wird dadurch der Wartungsvorgang beeinflußt, der sich zur Fehlerbehebung anschließen muß, zum anderen aber bleiben fälschlicherweise für transient gehaltene permanente Fehler latent auch über die Joblebenszeit hinaus im System vorhanden. Beim Starten eines neuen Jobs unter Teilnahme eines

1. Zwischen transienten (externen verursachten) und intermittierenden (internen) wird nicht unterschieden. Gelegentlich wird allerdings die Bezeichnung “transient” im Sinne von “temporär” verwendet.

solchermaßen fehlerhaften Knotens ist die erreichbare Zuverlässigkeit erheblich reduziert, da die dem Fehlertoleranzgrad entsprechenden Redundanzerfordernisse schon von Beginn an mißachtet werden. Solche Fehlerzustände, die einen Knoten als korrekt kooperierend erscheinen lassen, resultieren aber viel wahrscheinlicher aus korrumpierten Daten als aus Programmflußabweichungen. Ein fehlerhafter Knoten ist daher möglicherweise in der Lage, sich selbst als defekt zu diagnostizieren, worauf er von sich aus umfangreiche Selbsttests starten wird. Solange er sich seines fehlerfreien Zustand nicht sicher ist, wird er sich von weiteren Jobbewerbungen zurückhalten.

Unabhängig von Verdachtsmomenten können Knoten, die aktuell keinen Job ausführen, ihre Rechenkapazität für Selbsttests nutzen, um die Gefahr latenter Fehler zu verringern. Sie laufen zyklisch mit niedriger Priorität ab, und können jederzeit durch eintreffende Jobwünsche des Benutzers eingestellt werden.

Wie Abb. 42 zeigt, kommen noch weitere Mechanismen zur Fehlerdiagnose zum Einsatz. Fremdtests werden vor allem im Kommunikationssystem und beim Eintreffen von Botschaften in den lokalen Fehlertoleranzclerks durchgeführt. Dazu gehören vom Meldungsinhalt abhängige Plausibilitätstests, Formatüberprüfungen nach unterschiedlichen Akzeptanzkriterien und Zeitüberwachungen auf der höheren Schicht der FTL wie auch im Kommunikationstreiber. Dateninhalte in Interprozessornachrichten sind durch Signaturen (Checksummen) abgesichert.

Nicht nur während des Betriebs sondern auch *off-line* ist zur Wartungsunterstützung eine Fehlerdiagnose vorgesehen - eine Maßnahme zur Steigerung der Systemverfügbarkeit. Das Expertensystem DIAMONT [Phil91] versucht durch Beauftragung zur Durchführung von Selbsttests defekte Subkomponenten zu lokalisieren. Dabei geht es zur Auswahl der zu testenden Knoten von den on-line geführten Fehlerfrequenzlisten aus, um Testaufwand zu sparen. Tests auf unterschiedlichen Knoten können zudem parallel ausgeführt werden. Um die Diagnose zu beschleunigen, kann das System DIAMONT auch selbst verteilt ausgeführt werden.

Die Fehlerbehandlung ist in dem maskierenden System relativ einfach. In der Basisversion ist keine Defektkomponentenausgrenzung vorgesehen. Eine erweiterte Version nutzt die Ergebnisse der Selbsttests zur Selbstausgrenzung im Fehlerfall. Auch die Zwangsausgrenzung von als defekt diagnostizierten Komponenten kommt in Betracht. Unter bestimmten Randbedingungen kann sie eigenständig vorgenommen werden, z.B. werden Kommunikationspartner, auf deren Mailbox kein pysikalischer Zugriff mehr möglich ist, nach mehrmaligen Tests aus der Menge der Kommunikationspartner ausgeschlossen. Bei weniger eindeutiger Fehlererkennung und ausreichendem Verdacht, angezeigt durch Überschreiten eines Schwellwertes in der Fehlerfrequenz, wird von dem Knoten, der dies erstmalig beobachtet, eine vollständige Systemdiagnose eingeleitet. In der Regel dient die verteilte Systemdiagnose allein zur unmittelbaren Fehlerunterdrückung und wird nur unvollständig durchgeführt. Eine dezentrale t-Diagnostizierbarkeit, wie in Kapitel 4.1 definiert, verlangt, daß alle intakte Einheiten zu einem konsistenten Diagnosebild des Gesamtsystems gelangen. Tatsächlich gestattet der ATTEMPTO-Diagnosealgorithmus jedem Knoten zunächst nur, die unmittelbaren Nachbarn im Diagnosegraphen und sich selbst zu diagnostizieren. Zur vollständigen Diagnose der gesamten Jobkollegen ist noch ein weiterer Schritt im Diagnosealgorithmus nötig, der im Austausch und Abgleich der lokal ermittelten Fehlermuster zwischen Nachbarn besteht. Danach ist eine konsistente Grundlage geschaffen, um den verdächtigen Knoten aus dem System zu entfernen. Dies geschieht durch Anlegen eines HALT-Signals von außen über ein Kontrollinterface im globalen VMEbus-Addreßbereich.

Fehlermodell

Fehlertypen
- Hardware-Fehler
- transiente (hauptsächlich)
- permanente
- auch (unabhängige) Mehrfachfehler

Fehlererkennung und -lokalisierung

Fremdtests
- Zeitüberwachung
- Plausibilitätstests
- Signaturvergleich

Selbsttests
- On-line, zyklisch, mit niedriger Priorität, bei Verdacht auf perman. Fehler

Fehlerdiagnose
- on-line, zur Fehlermaskierung:
  verteilte Systemdiagnose auf Basis von (t-optimalen) Diagnosegraphen
- lokal on-line, Selbstdiagnose
- off-line, zur Unterstützung der Reparatur,
  - aufgrund von Selbsttestergebnissen zur Lokalisierung defekter Subkomponenten
  - ausgehend von on-line geführten Fehlerfrequenzlisten

Fehlerbehandlung

Defektkomponentenausgrenzung
- Basisversion: gar keine
- Selbstausgrenzung nach Selbstdiagnose
- Ausgrenzung durch intakte Knoten nach Abgleich auf Basis von Schwellwerten in der Fehlerfrequenz

Reintegration reparierter Defekt-Komponenten
- nur off-line

Abb. 42 Fehlertoleranzeigenschaften

Die Selbstausgrenzung findet nicht immer nur als Folge eines expliziten Selbsttests statt, sondern auch dann, wenn implizite Fehlererkennungsmechanismen in der Hardware (Busfehler, Addressfehler, Parity-Fehler im Speicher, unerwartete Ausnahmen allgemein) und im lokalen

Betriebssystem (Ausnahmebehandlungen als Folge der Hardwarefehlerdetektion und interne Bearbeitungsfehler im Betriebsystem) wirken. In der Regel wird daraufhin ein Neustart des betroffenen Knotens eingeleitet. Wünschenswert wäre eine Reintegration des Knotens nach dem Wiederanlauf, falls dieser möglich und ein umfangreicher *Power-Up*-Selbsttests mit positivem Endergebnis dies erlauben würde. Dies ist jedoch zur Zeit nicht möglich, und kann nur - ebenso wie die Reintegration reparierter Defektkomponenten - durch Neustart des gesamten Fehlertoleranz-Systems erreicht werden.

Wie zuvor bereits deutlich wurde, gibt es unterschiedliche Klassen von Selbsttestfunktionen:

- Der ***Power-Up-Selbsttest*** dient dazu, beim Systemstart eine Grundkonfiguration intakter Knoten zu bestimmen. Er ist der umfangreichste Test mit der höchsten Testüberdeckung und der längsten Laufzeit. Selbsttests verlangen immer eine als ideal vorausgesetzte Basis (*Hardcore*), von der ausgehend nach und nach weitere Systemkomponenten ausgetestet werden, wobei die Funktionen des Hardcore und damit die Testvoraussetzungen für die folgenden Tests stetig anwachsen. Diese Vorgehensweise ist als *Start-Small-Strategie* bekannt [Dent68]. Im allgemeinen ist dafür allerdings detaillierte Hardwarekenntnis, nicht nur auf Board- sondern auch auf Chipebene notwendig, sodaß ohne Unterstützung durch den Bauteilhersteller nur angenäherte Ansätze möglich sind. Häufig ist daher der initiale Hardcore schon sehr umfangreich und hochintegrierte Bausteine können nur grob in ihren Grundfunktionen und nicht im gesamten Leistungsumfang getestet werden.

  Erkannte Fehlzustände werden je nach verbleibenden Möglichkeiten auf einer Systemkonsole, in einer Testprotokoll-Datei, und mit entsprechenden Anzeigeelementen (Fehler-LED) auf den Rechnerbaugruppen angezeigt. Die Systemkonsole kann an das Diagnosesystem DIAMONT angebunden sein.
- ***On-Line-Tests*** werden vom Diagnosesystem initiiert. Dazu muß ein relativ umfangreicher *Hardcore* im System vorausgesetzt werden. Dazu gehört neben einem ausreichend fehlerfreien lokalen Betriebssystem vor allem die Netzbetriebsfähigkeit. Ist sie nicht mehr gegeben, kann die Diagnose ersatzweise über die Konsolschnittstelle fortgeführt werden. U.U. ist dazu ein manueller Eingriff des Wartungspersonals erforderlich. Manuelle Mithilfe zur Hardwarediagnose ist in DIAMONT auch sonst ein übliches Mittel zur Testdatenaquisition.
- ***Transparenztests*** sind diejenigen Tests, die die Fehlertoleranzschicht in der Aufdeckung von permanenten Fehlern behilflich sind.

On-Line- wie auch Transparenztests sind, zusammengefaßt in einem Testtreiber */dev/test*, Kernfunktionen im lokalen Betriebssystem. Randbedingungen des Multiprozeßbetriebs bewirken Einschränkungen in der Testfunktionalität, sodaß diese Tests nur eine Untermenge der *Power-Up*-Selbsttests darstellen. Gleichzeitig muß auf ein befriedigendes Laufzeitverhalten bei jedoch noch möglichst großer Fehlerüberdeckung geachtet werden. Die Vertrauenswürdigkeit von Tests kann im System DIAMONT modelliert werden. Das Testselektionsverfahren beruht auf logischen und heuristischen Inferenzen und ist darum bemüht, Test mit der höchstgewinnbringenden Aussage zu bevorzugen und dabei gleichzeitig ein größtes Maß an Testparallelität zu erreichen. Allgemein verfügt das Diagnosemodell von DIAMONT über Modellierungsmöglichkeiten für den strukturellen und funktionalen Zusammenhang der Einzelkomponenten inhomogener Multiprozessor- bzw. Mehrrechnersysteme. Die Integration von Tests in das Fehlertoleranzkonzept von ATTEMPTO und die Modellierung und Anbindung an das Diagnosesystem wurden im Rahmen einer Diplomarbeit [Oppm93] durchgeführt.

5.8 Spezielle Implementierungsprobleme

Die kooperative Erbringung von Systemfunktionen durch lokale Systemprozesse und auf Rechnerknoten verteilte Fehlertoleranzinstanzen bewirkt einen hohen Grad an Nebenläufigkeit in ATTEMPTO. Dies führt zu Problemen bei der praktischen Implementierung, die in besonderer Weise berücksichtigt werden müssen. Vor allem beziehen sich diese auf die Bewältigung von Nichtdeterminismus im Kommunikationssystem und bei der Verteilung von Eingabedaten; dieses letztgenannte Problem wurde bereits in Kapitel 3.2 vorgestellt. Hinzu kommen noch Unterschiede in den lokalen Laufzeitumgebungen der Jobreplikate. Die wichtigsten dieser Probleme werden im weiteren näher behandelt.

5.8.1 Probleme mit Nebenläufigkeit im Kommunikationssystem

Im Experimentalsystem gibt es insgesamt drei Quellen für nebenläufige Aktivitäten, die die Bearbeitung einer lokalen Fehlertoleranzinstanz beeinflussen können:

- Aktionen des Benutzers (der Bedienperson)
- Eintreffen von Nachrichten von anderen Rechnerknoten
- Aktivitäten der überwachten Anwenderprozesse.

In der Modulstruktur spiegelt sich dies wider in der Anzahl der Handler-Prozesse, die den lokalen FTI-Prozeß unterstützen. Jede hinzukommende globale Ressource wird den Grad der Nebenläufigkeit erhöhen. Alle Aktivitäten im System werden durch Botschaften angezeigt, die unbestimmt durchmischt in den lokalen Post-Offices eintreffen. Die globale Nachrichtenreihenfolge kann zwar für Rundspruchnachrichten im Interprozessor-Kommuniationssystem gesichert werden, für das lokale Kommunikationssystem gilt dies jedoch nicht. Dies wirkt sich allerdings nur dann schädlich aus, wenn Kausalitäten zwischen aufeinanderfolgenden Aktivitäten verletzt werden. Genau dies trifft jedoch auf die folgenden drei Fälle zu. In den beiden ersten vermischen sich Interprozeßnachrichten mit lokalen Botschaften, bei dem letzten handelt es sich um rein lokale Aktivitäten. Ein viertes Problem zeigt eine Schwäche des einfachen Wettlaufprinzips (Kapitel 5.6.3) zur Ressourcenverwaltung.

- **StartRequest - Botschaft eines fremden Knotens überholt eigene NewJob-Meldung.**

  Die Broadcastnachricht *StartRequest* kann nur die Ordnung von Job-Kontroll-Blöcken (JCB) in den dezentralen Job-Listen koordinieren, angelegt wird ein solcher JCB aber aufgrund einer dezentralen Erfassung des Benutzerkommandos über die globale Ressource. Kommt es wegen der Entkopplung von der Eingabe durch das lokale Betriebssystem und den Terminalhandler-Prozeß zu einer Verspätung der *NewJob*-Nachricht, trägt die fremde *StartRequest*-Nachricht eine unbekannte Jobreferenz. Deshalb wird in diesem Fall präventiv ein JCB angelegt, der bei Eintreffen der *NewJob*-Nachricht im Namensfeld aktualisiert werden muß. Die *StartRequest*-Nachricht muß den Fehlertoleranzgrad mitführen, damit die Kollegenliste konsistent verwaltet werden kann.

- **Terminierung des eigenen Job-Replikats vernichtet die Jobreferenz.**

  Dies ist der umgekehrte Fall: der JCB ist lokal schon vernichtet (aufgrund einer lokalen *UserJobExit*-Nachricht), dennoch treffen Interprozeßnachrichten ein, die sich noch darauf beziehen. Um dieser Situation gerecht zu werden, wird die Lebensdauer eines Jobs (der dezentralen Job-Kontroll-Blöcke) global synchronisiert, wie in Kapitel 5.6.1 erläutert. Diese Lösung wird der Alternative vorgezogen, die “zuspät” kommenden Interprozeßnachrichten einfach zu verwerfen, weil so globale Jobbelegungszustände leichter verwaltet werden können. Fehlerhafte Nachrichten lassen sich besser abgrenzen, und

Zeitschranken für den künftigen Ablauf leichter dimensionieren (siehe dazu Kapitel 5.6.5).

- **Noch vorhandene Job-Eingabedaten in der UserJob/Shell-Pipe beim asynchronen Job-Exit.**

  Benutzereingaben sind zunächst nicht koordiniert mit der Lebensdauer eines Jobs, erst durch eine für den Benutzer sichtbare Systemreaktion auf das Jobende kann eine Rückkopplung durch den Benutzer selbst stattfinden. Zudem muß die Anzahl der vom Anwender-Job geforderten und vom Benutzer angebotenen Eingabedaten nicht zwangsläufig übereinstimmen. Etwaige überzählig im System gespeicherten Daten müssen entweder vernichtet werden, oder sie können für die Eingabeanforderung nachfolgender Jobs zurückbehalten werden. Dieses letztgenannte Verfahren ist i.a. bei Zwischenpufferung von Benutzereingaben in Monoprozessor-Betriebssystemen die Regel, so auch in UNIX. Der Nachfolge-Job ist hier die Kommando-Shell des Benutzers. Durch den Eingabevorlauf kann auch unbeabsichtigt ein Job gestartet werden. Man nimmt dies jedoch in Kauf, weil bei absichtlichem Kommando-Vorlauf die Vorteile überwiegen (schnelle Befehlseingabe). Diese Verhältnisse können für das ATTEMPTO-System nicht unverändert übernommen werden, denn hier kann sich dies in zweierlei Hinsicht fehlerhaft auswirken. Zum einen muß immer gewährleist sein, daß alle Knoten dieselben Kommandos erhalten. Dieses Problem der Quellkongruenz ist zunächst ganz allgemein zu lösen (siehe dazu auch das folgende Kapitel). In diesem speziellen Fall des Eingabeüberschusses ist diese Konsistenzforderung allerdings nicht nur möglicherweise, sondern sogar immer verletzt, wenn der Replikationsgrad für den zuletzt gelaufenen Job kleiner war als die Gesamtknotenzahl, denn nur bei den letztbeteiligten Kollegen gibt es diese überzähligen Eingabedaten. Zum anderen - und das ist das wesentliche - darf ohne gezielte Steuerung durch die FTL überhaupt kein Anwenderjob gestartet werden. Dies wäre aber immer der Fall, da die überzähligen Shell-Eingabedaten nie in einer *NewJob*-Nachricht erfaßt worden sind. Der einzige Ausweg besteht darin, bei der Jobbeendigung die UserJob/Shell-Pipe zu entleeren. Ein *read*()-Systeaufruf des Shell-Prozesses darf erst dann befriedigt werden, wenn sich in ihrem Eingabekanal abgesicherte Kommandos befinden, für die bereits eine erfolgreiche Bewerbung durchgeführt worden ist.

- **Reihenfolgeverletzung von Jobausgaben fehlertoleranter Jobs bei häufigem Ausgabewechsel.**

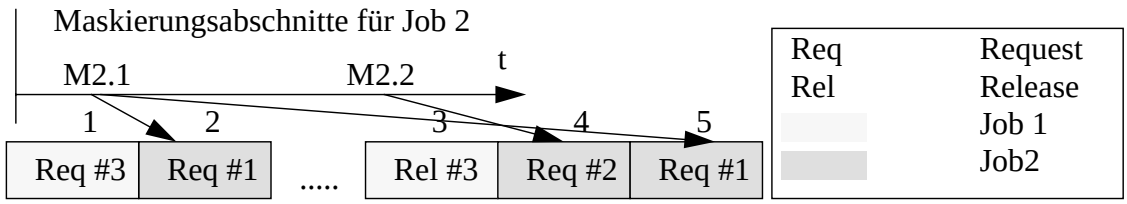


Abb. 43 Ausgabevermischung eines fehlertoleranten Jobs

Die Ausgabe der Jobergebnisse erfolgt abgekoppelt vom Maskierungsvorgang. Bei entsprechender Belastung der Systemressource, z.B. durch ausgabeaktive Parallel-Jobs, können sich Ausgabedaten ansammeln. Bei fehlertoleranten Jobs wird nach jedem Maskierungsvorgang ein neuer *Output-Master* (siehe Kapitel 5.6.4) zwischen den Kollegen durch einen Bewerbungswettlauf ausgehandelt, sodaß sich die Ausgabedaten auf ver-

schiedene Ressource-Manager aufteilen. Diese werden dann unabhängig von der durch den Anwenderprogrammlauf vorgegebenen Reihenfolge miteinander um die Ausgabe konkurrieren. Das Problem der Reihenfolgeverletzung entsteht ursächlich dadurch, daß erfolglose Bewerbungen um die globale Ressource verworfen werden und wiederholt werden müssen. Abb. 43 zeigt ein Beispiel dafür: die Request-Nachricht des Outputmasters Knoten #1 für die Daten im Maskierungsabschnitt 1 des Jobs 2 verliert den Wettlauf gegen einen fremden Job (Req #3 für Job 1). Bis zu dessen Freigabe (durch Rel #3) wird eine weitere Maskierung durchgeführt (M2.2). Für die zugehörige Ausgabe ist Jobkollege Knoten #2 zuständig, der den neuerlichen Wettlauf nach der Freigabe gewinnt.

Das Problem könnte durch dezentrale Zwischenspeicherung der Anforderungen und Befriedigung in der Reihenfolge ihres Eintreffens gelöst werden (Variante 1), oder aber indem Buch geführt wird über die aktuelle Jobausgabesequenz (durch Nummerierung), sodaß zufrühe Bewerbungen zurückgehalten werden können (Variante 2). Für die letztgenannte Variante ist ein Rückopplungsmechanismeus nötig. In [Grill93] wird ein Algorithmus zur Ressourcensteuerung vorgestellt, der die Variante 1 verfolgt. Er zielt gleichzeitig auf minimale Anzahl von Bewerbungsbotschaften ab, um den Ausgabedurchsatz zu erhöhen, und beinhaltet ein (rudimentäres) Abstimmungsverfahren für verbesserte Robustheit.

5.8.2 Niedrige Betriebssystem-Dienste zur Ein/Ausgabe

Hierunter sind Dienstleistungen des lokalen Betriebssystems zu verstehen, soweit sie sich auf asynchrone Ein/Ausgabe von/auf globale Ressourcen beziehen. Dieser Themenkomplex wurde bereits in Kapitel 3.2.2 und Kapitel 3.2.3 unter dem Begriff der Quellkongruenz erläutert. Mit naturbedingtem Indeterminismus ist vor allem bei der nichtblockierenden Eingabe und bei der Signalbearbeitung zu rechnen. Doch auch die durch das Betriebssystem vom Prozeß abgekoppelte Ausgabe kann problematisch werden. Die einzelnen Probleme werden im folgenden aus einem Implementationsblickwinkel näher betrachtet.

- **Terminal-Treiber-Line-Discipline: Raw-Mode-Input.**

  Die replizierten Sensoren zur Eingabeerfassung für die globale Ressource “Terminal” im ATTEMPTO-Experimentalsystem bestehen aus der Schnittstellenhardware und Gerätetreibern im Betriebssystem UNIX. Die Betriebsart des Terminaltreibers, die sogenannte *Line Discipline*, läßt sich wählen. Ein Anwenderprozeß stellt sie ein mithilfe eines gerätespezifischen *ioctl()*-Systemaufrufes. Für Terminaltreiber gibt es vielfältige Einstellmöglichkeiten. Zwei davon erweisen sich als besonders problematisch: automatische Echoerzeugung für Eingabezeichen und unformatierte zeichenorientierte Eingabe. Im Gegensatz zur zeichenorientierten Eingabe (*raw mode input*) gibt es die zeilenorientierte Eingabe (*canonical mode* oder *line mode*), die es gestattet, eine unbestimmte Anzahl von Zeichen zu lesen in Form einer Zeichenkette, die mit einem speziellen Zeilenende-Charakter abgeschlossen ist. Im raw-mode findet in der Regel keinerlei Interpretation einzelner Zeichen statt, während im canonical mode i.a. zahlreiche Sonderbehandlungen, z.B. zur Zeilenedition, durchgeführt werden, bevor die komplette Zeile an den Anwenderprozeß weitergegeben wird. Im raw mode besteht die Wahlmöglichkeit, eine bestimmte definierbare Anzahl von Zeichen aufzusammeln, oder eine bestimmte Zeitlang auf eintreffende Zeichen zu warten. Darüberhinaus lassen sich diese Möglichkeiten auch kombinieren: der *read*-Systemaufruf terminiert dann, wenn eine der beiden Bedingungen erfüllt ist. Die Anzahl der tatsächlich gelesenen Zeichen wird dem Prozeß mit zurückgegeben.

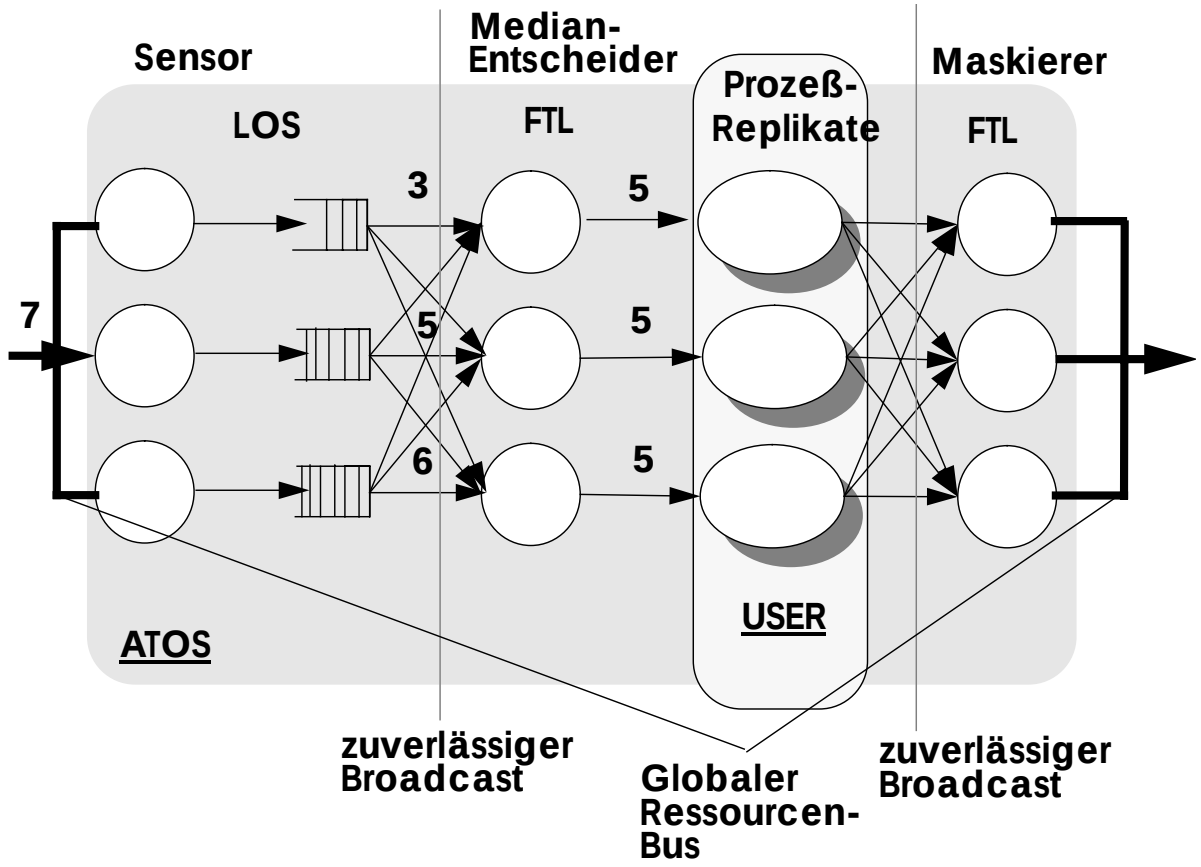


Abb. 44 Abgleich der Anzahl lokal eingetroffener Zeichen zur Elimination von Indeterminismus

Insbesondere die zeitbegrenzte Erfassung ist eine Quelle für Indeterminismus. Aber auch die Anzahlbegrenzung erweist sich als kritisch, da der Wert Anzahl=0 zugelassen ist; dadurch ergibt sich nichtblockierendes Lesen[1]: der Eingabepuffer im Terminaltreiber wird abgetastet (“Polling”). Wegen der asynchronen Abspeicherung von Eingabezeichen in den lokalen Eingabepuffern durch die Treiber, und der unkoordinierten Pollvorgänge lassen sich ungleich verteilte Eingabedatenportionen nicht vermeiden; bei zyklischem Pollen wird dadurch sogar die Anzahl der read-Systemaufrufe indeterministisch. Ein Abstimmungsverfahren zwischen den Jobkollegen ist also in diesem Fall unumgänglich. Es kann jedoch ausgesetzt werden, wenn die Anwendung im Zeilenmodus arbeitet. Dazu muß der Job auf einen *ioctl()*-Systemaufruf hin überwacht werden.

Das Abstimmungsverfahren soll dazu dienen, die Anzahl der mit jedem *read*-Systemaufruf zurückgegebenen Zeichen global festzulegen. Dafür kommt ein Votierungsverfahren für nichtdeterministische Daten in Betracht, z.B. auf der Basis eines Medianentscheids. Dieses Grundprinzip ist in Abb. 44 für einen Jobreplikationsgrad n=3 dargestellt. Die Maskierung ist trotz des Indeterminismus möglich, weil sich die Daten in einem geordneten Intervall anordnen lassen [Echt90]. Das Verfahren ist robust: das Medianelement ergibt immer einen sinnvollen Wert, wenn die Zahl n (der Replizierungsgrad) der der Maskierungsentscheidung zugrundeliegenden Daten ungerade ist, und nicht mehr als t=(n-1)/2 fehlerhafte Daten vorliegen. Allerdings entspricht dieses Modell nicht demje-

1. *Non-blocking I/O* ist in UNIX auch eine eigenständige Betriebsart, die gleich beim Öffnen eines Kanals eingestellt werden kann.

nigen, das der Fehlermaskierung in ATTEMPTO zugrundeliegt. Wegen des Vergleichstestverfahrens für die verteilte Systemdiagnose ist hier t=n-2, wobei n auch gerade Werte annehmen kann. Betrachtet man den Beispielfall des Maskierungsentscheides zur Eingabeabstimmung mit n=4 Kollegen, und t=2 Fehlern anhand des Intervalls

$$< 0,0,1,1>,$$

erkennt man, daß keine sichere Entscheidung getroffen werden kann. Hierbei ist angenommen, daß ausgefallene Knoten einen Wert 0 zum Intervall beitragen, 0 aber auch ein korrektes Ergebnis sein kann. Außerdem können auch Werte > 0 falsch sein. Eine Fehlentscheidung wäre fatal: wenn die beiden linken Werte fehlerhaft sind, tritt bei zyklischem Pollen möglicherweise ein *Lifelock* auf, falls die Entscheidung 0 ist, im umgekehrten Fall, bei zwei Fehlern rechts, sind bei einer Fehlentscheidung für 1 überhaupt keine Daten vorrätig, und der Inhalt des Anwenderdatenpuffers ist unbestimmt.

Zusätzliche Fehlererkennungsmechanismen sind also für ein zuverlässiges Abstimmungsergebnis notwendig. Dazu muß das Fehlermodell zunächst präzisiert werden. In einem Fail-Stop-Fehlermodell (Nachrichteninhalte sind immer korrekt oder treffen nicht ein) wären Werte > 0 immer vertrauenswürdig und somit eine korrekte Entscheidung kein Problem. Bei möglichen Replikationsfehlern würde der Maskierungsentscheid in dezentralen Maskierungsinstanzen erheblich erschwert (siehe Kapitel 3.3). Replikationsfehler sind aber im ATTEMPTO-Fehlermodell wegen der zuverlässigen Broadcast-Kommunikation nicht angenommen, und die Fail-Save-Eigenschaft wird nicht vorausgesetzt (Kapitel 5.7).

Ein für das ATTEMPTO-System geeigneter Abstimmungsalgorithmus definiert [Klau92]. Der Maskierungsentscheid berücksichtigt dabei sowohl Anzahl wie auch Inhalt der eigentlichen Lesedaten. Dabei wird fehlerhafter, aber gleicher Dateninhalt (Fehlersymptomgleichheit) als geringwahrscheinlich von der Betrachtung ausgenommen. Der Algorithmus bemüht sich, die längste Eingabezeichenkette zu ermitteln, für die zwischen wenigstens zwei Kollegen Übereinstimmung herrscht.

Im Experimentalsystem ist der Abstimmungsalgorithmus nicht implementiert; raw-mode-Eingabe in fehlertoleranten Programmen ist nicht erlaubt. Deshalb muß auch der *ioctl()*-Systemaufruf nicht überwacht werden. Diese Einschränkung kann nur im experimentellen Prototyp hingenommen werden. Sie hat aber hier noch weiteren Implementationsaufwand erspart. Grundsätzlich muß es natürlich auch möglich sein - zumindest für eine fenstergesteuerte Benutzeroberfläche - , daß verschiedene parallel laufende Anwenderjobs auch unterschiedliche Betriebsarten des Terminaltreibers wählen können. Alle Jobs kommunizieren jedoch über diesselbe Terminalleitung. Da der raw-mode eine funktionale Untermenge des canonical-modes ist, muß die physikalische Schnittstelle im raw-mode betrieben werden.

Der Terminaltreiber ist also fest auf diese Betriebsart eingestellt, und der Terminalhandler-Prozeß sammelt die Eingaben zeichenweise in unbestimmter Anzahl, wie sie eintreffen, identifiziert ihre Jobzugehörigkeit in den Nachrichten zum FTI-Prozeß und veranlaßt ihre erneute Zwischenspeicherung im DIB. In der Fehlertoleranzinstanz müssen nun alle Treiberfunktionen der line-discipline nachgebildet werden, damit die Eingabezeichen jobspezifisch behandelt werden können. Neben dem unerwünscht hohen Aufwand

und der zu erwartenden Leistungseinbuße bedeutet dies auch die Verschlechterung der Portierbarkeitseigenschaften für die Fehlertoleranzbetriebssoftware.

- **Terminal-Treiber-Line-Discipline: Echo-Behandlung.**

  Dies ist der schon zuvor erwähnte zweite problematische Betriebsparameter. Echos werden aus Performanzgründen bereits auf unterer Betriebssystemebene erzeugt, insbesondere um dem Benutzer eine akzeptabel schnelle Systemantwort auf seine Eingabe zu geben. Schon weil in ATTEMPTO unterschiedliche Jobs unterschiedlichen Echobetrieb verwenden dürfen, kann diese Funktion nicht weiter im LOS behalten werden. Stattdessen bieten sich verschiedene Alternativen an:

  - Erzeugung durch die FTI beim Ablegen der Zeichen im DIB,
  - Erzeugung im Endgerät.

  Die letztgenannte Alternative ist nur bei “intelligentem” Endgerät realisierbar, das die Einstellbefehle der FTL verarbeitet. Bei einer fenstergesteuerten Benutzeroberfläche bietet sich diese Lösung an, da das Antwortzeitverhalten für Echos sehr gut ist. Gleichzeitig bedeutet dies natürlich einer Verlagerung von Systemfunktionen auf eine außerhalb des betrachteten Fehlerbereichs des Maskierungssystems liegende zentrale Instanz und damit eine Verringerung der Systemfunktionalität und möglicherweise verringerte Robustheit. Bei Ausfall des Endgerätes sind jedoch auch keine Echozeichen erforderlich. Die Echoerzeugung selbst verlangt wegen der vergleichsweise geringen Bedeutung dieser Funktion auch keine hohe Zuverlässigkeit.

  Bei einem unintelligenten asynchronen Datenterminal wird sich bei der Erzeugung der Echos in der Fehlertoleranzschicht ein unbefriedigendes Antwortzeitverhalten ergeben. Messungen zeigen, daß die Verzögerung im Bereich von mehreren hundert Millisekunden liegen kann, bei hoher Systembelastung sogar noch erheblich mehr. Der Grund dafür ist hauptsächlich die Ermittlung des für das Echo zuständigen FTI-Kollegen und die erforderliche Ressourcenkontrolle. Unterliegen Echos wie normale Jobausgaben dem üblichen Fehlermaskierungsmechanismus, wird die Zuständigkeit für die Echoausgabe wie gewöhnlich durch einen Bewerbungswettlauf der intakten Kollegen entschieden. Dieses Verfahren paßt sich sehr homogen in das bestehende System ein, verhindert fehlerhafte Echos, führt aber zu hohen Verzögerungen. Zur Verbesserung des Antwortzeitverhaltens kann z.B. auf die Maskierung verzichtet werden, und damit allerdings auch die Zuverlässigkeit verringert werden. Selbst auf das Bewerbungsverfahren als Echo-Master kann verzichtet werden, indem eine feste Zuordnungvorschrift auf die Kollegenmenge definiert wird, die für den Fehlerfall Ersatzfestlegungen trifft. Im Umschaltmoment treten allerdings dann Inkonsistenzen auf. In allen Fällen müssen Echos durch die Fehlertoleranzschicht geleitet werden und das Umschalten der Echobetriebsart ist zu überwachen. Für den experimentellen Prototypen mit unintelligentem Terminal ist Echobetrieb fest vorgeschrieben. Das Echo kann entweder im Endgerät erzeugt werden, falls dieses dort möglich ist, oder diese Aufgabe kann dem Terminaltreiber eines festgelegten Knotens dauerhaft angelastet werden.

- **Signale vom Terminal**

  Die Problematik der zustandsgleichen Einphasung von Signalen in den Programmfluß der Prozeßexemplare wurde bereits in Kapitel 3.2.3 ausführlich behandelt. Dort wurde auch bereits deutlich, daß bei hochgradiger Asynchronität wie im ATTEMPTO-System erhebliche Schwierigkeiten auftreten. Die infrage kommende Ereignissynchronisierung

mit Vorlaufausgleich ist ohne Hardwareunterstützung nur mit sehr aufwendigen zusätzlichen Checkpointing-Mechanismen machbar. Aus diesem Grunde wird eine Einschränkung der ursprünglichen Transparenzforderung für Anwendungsprogramme hingenommen. Prozesse dürfen keine eigendefinierte Signalbehandlung durchführen (d.h. Programme dürfen keinen *signal()*-Systemaufruf enthalten). Die Standard-Signalbehandlung - Prozeß-Terminierung - kann jedoch zugelassen werden, da der Folgezustand der Prozeßreplikate nach der Signaleinwirkung deterministisch ist. Es muß jedoch dafür Sorge getragen werden, daß etwaige schon begonnene Fehlertoleranzmechanismen, z.B. zur Fehlermaskierung, konsistent zuende geführt oder geordnet abgebrochen werden. Dazu muß das Signal vom Benutzer in eine Nachricht transformiert werden, die der FTL zugeleitet wird. Der sonst übliche Weg, das Signal schon im Terminal-Treiber aus dem Eingangsdatenstrom zu extrahieren, und die Weiterleitung direkt im LOS-Kern vorzunehmen, muß gesperrt werden.

- **Konkurrierendes Lesen verwandter Prozesse vom selben Kanal**

  In UNIX ist es durchaus möglich, daß mehrere Prozesse quasiparallel lesend auf ein- und denselben Eingabekanal zugreifen; das Prozeß-Scheduling bestimmt dann die Datenaufteilung mit. Dieser inhärente Indeterminismus pflanzt sich in der ATTEMPTO-Multiprozessorumgebung fort. Da das Prozeß-Scheduling eine auschließlich lokale Systemfunktion ist, ist die Abweichung einzelner Prozeß-Replikate vom gemeinsamen Programmfluß vorhersehbar. Die Gefährdung besteht auch, wenn der Eingabekanal lokal ist, z.B. eine Pipe oder ein Gerät im lokalen Zuständigkeitsbereich. Um dieses Problem im Maskierungssystem zu lösen, wäre der Abgleich aller Lesedaten - und nicht nur der von globalen Ressourcen zur Lösung des zuvor beschriebenen Problems der asynchronen nichtblockierenden Leseoperationen - erforderlich. Wegen der hohen Kosten wird deshalb im ATTEMPTO-Experimentalsystem darauf verzichtet. Konkurrierendes Lesen ist daher für fehlertolerante Jobs nicht zugelassen.

- **Ausgabekollision bei großen Schreibportionen**

  kann auftreten, wenn keine Rückkopplung über asynchron ausgegebene Daten zurück zum schreibenden Prozeß - dem ausgebenden Ressource-Manager - besteht. Dies ist in UNIX die Regel. Falls das Betriebssystem keine Möglichkeit zur synchronen Datenausgabe anbietet - im für das Experimentalsystem verwendeten Basis-Betriebsystem ist dies gegeben - muß der Mechanismus zum Ressource-Locking erweitert werden, z.B. indem eine abhängig von der auszugebenden Zeichenmenge geschätzte Wartezeit verstreichen muß, bevor die Ressource für andere Knoten freigegeben wird. Sichere Wartezeiten verlangen jedoch eine pessimistische Schätzung, sodaß die Ausgabefrequenz über Gebühr verringert wird.

5.8.3 Aktuelle Laufzeitumgebung

Die aktuelle Laufzeitumgebung eines Jobs wird bestimmt durch seine Einbettung in das jeweilige lokale Betriebssystem, das nur begrenzt durch die erweiterten ATOS-Systemfunktionen über eine globale Systemsicht verfügt, und durch eine künstliche Umgebung, repräsentiert durch Umgebungsvariablen, die unter lokaler Verwaltung der replizierten Kommandointerpreter (Shell-Prozesse) stehen.

Die

- **lokalen Systemdienste**

  können den Gleichlauf der Prozeßreplikate gefährden, wie bereits in Kapitel 3.2.1 be-

schrieben. Neben den synchronen Systemaufrufen für Informationsdienste, die lokale Kerndaten referenzieren und deshalb global zu synchronisieren sind, darf es nur unter drastischen Einschränkungen für die Funktionalität eines fehlertolerant auszuführenden Programmes Datenverkehr über lokale Schnittstellen geben. Das hauptsächliche Problem bei interaktivem Allzweckbetrieb stellt hier das Dateisystem dar, das in der experimentellen ATTEMPTO-Realisierung nur aus völlig unabhängigen lokalen Einzel-Dateisystemen besteht. Für den praktischen Gebrauch ist hier unbedingt ein geschlossenes Konzept für ein verteiltes Dateisystem, das auch Fehlertoleranzeigenschaften beinhaltet, vonnöten. Für den Prototypen ist dieser Gesichtspunkt ausgeklammert worden. Ähnliches gilt auch für die

- **Umgebungs-Variablen.**

  Außerhalb des Adreßraums des Anwenderprozesses gelagert, für diesen aber durch den Kommandointerpreter beim Start verfügbar gemacht, befinden sich Daten, die die Ausführung des Programmes beeinflussen können. Diese in UNIX Umgebungs-Variablen genannten Daten (*environment variables*) sind zunächst beliebiger Art und können frei definiert werden. Hauptsächlich dienen sie jedoch nach Konvention dazu, für die Shell selbst und für die Anwenderprozesse eine Plattform zu schaffen, die von Systembesonderheiten abstrahiert oder allgemeine Systeminformation für die Kommandoausführung bereithält. Dazu gehören benutzerbezogene Daten (USER-Name, HOME-Verzeichnis, etc.) Maschinen- und Netzwerkdaten (HOST-Name, Maschinentyp,..), Anwenderprogrammschnittstellen (Lagerorte von speziellen Dateien, Programmnamen, ...) und solche Daten, die die Shell direkt benötigt, wie z.B. der aktuelle Pfadname (PWD) oder Suchpfade zum Auffinden von zu startenden Programmen (PATH). Weichen die Umgebungsvariablen voneinander ab, ist der deterministische Gleichlauf der Prozeßreplikate in Gefahr. Insbesondere, wenn Suchpfade oder Arbeitsverzeichnisse nicht übereinstimmen, kann es dazu kommen, daß Jobreplikate überhaupt nicht gestartet werden.

  Mit der Replizierung der Prozeßexemplare sind also auch die Umgebungs-Variablen aneinander anzugleichen. Da auf den ATTEMPTO-Rechnerknoten jedoch früher gestartete Jobs in unterschiedlichen Umgebungen abgelaufen sein können, stellt sich die Frage, welche Umgebung für die sich zufällig zusammenfindenden Kollegen eines neuen Jobs relevant sein soll. Diese Entscheidung wird auch durch die Ausgestaltung der Benutzeroberfläche mitbestimmt. Im Experimentalsystem muß der Benutzer vor Start eines fehlertoleranten Jobs eine homogene Laufzeitumgebung selbst hergestellt haben. Ein automatischer Abgleich ist nicht implementiert.

  Das gewählte Konzept zum lokalen Starten von Anwender-Jobs (siehe Kapitel 5.6.1) durch den gewohnten unveränderten Kommandointerpreter (Shell) gestattet die freie Wahl desselben. Entgegen der idealen Vorstellung können dessen Funktionen jedoch nicht völlig transparent für die Fehlertoleranzschicht sein, schon deshalb, weil die Shell die Umgebungsvariablen verwaltet. Darüberhinaus machen jedoch

- **Eingebaute Shell-Kommandos**

  allgemein Probleme. Der Terminalhandlerprozeß TH müßte Kenntnis über diese besitzen, um eingebaute Kommandos von vornherein von Anwenderprogrammen für den FTD-Clerk unterscheidbar zu machen, da ihre Behandlung in der FTL nicht gleich sein kann. Für eingebaute Kommandos werden keine Prozesse erzeugt, die FTL setzt dies jedoch für Anwenderprozesse voraus. Mit jedem Kommando, angezeigt durch die Botschaft *NewJob*, ist ein Job-Kontrollblock (JCB) verbunden, der solange in der Jobwarte-

schlange bleibt, bis die mit dem Job assoziierte Prozeßgruppe terminiert. Dies wird anhand des *exit()*-Systemaufrufes überwacht, der bei eingebauten Kommandos natürlich ausbleibt. Als Folge davon wird der JCB nie aus der Jobwarteschlange ausgekettet, der bearbeitende Knoten gilt als belegt und ist damit permanent blockiert. Neben der Fähigkeit, eingebaute Kommandos der jeweiligen Shell zu identifizieren, müßte der TH-Prozeß außerdem die komplette Syntax der Kommandosprache beherrschen, um zweifelsfrei Shell-Eingaben von reinen Jobnamen trennen zu können.

Die Verwendung eines unveränderten Standard-Kommandointerpreters ist im ATTEMPTO-System also möglich, dessen Funktionen müssen aber im Fehlertoleranz-Betriebssystem zusätzlich nachgebildet werden. Im Prototyp ist darauf jedoch verzichtet worden. Stattdessen sind diese Aufgaben über die Systemschnittstelle zur Umwelt hin auf den Benutzer verlagert worden. Er muß Shell-Eingaben als solche besonders kennzeichnen (“Eingabe für den Job #0” anstelle von “Kommandoeingabe”). Sie werden daraufhin ohne weitere Verwaltung jedem Shellprozeß im System zugeleitet und also auf allen Knoten gleich bearbeitet. Auf diese Weise ist es leicht möglich, die notwendige homogene Laufzeitumgebung für fehlertolerante Jobs einzurichten, wie im vorigen Abschnitt gefordert wurde.

6. ATTEMPTO: Systembewertung

Die Bewertung eines jeden technischen Systems hat zwei Aspekte: die **Validation** beurteilt die Fähigkeit des Systems, adäquat die Anforderungen des Betreibers erfüllen zu können (ist das richtige System gebaut worden?), die **Verifikation** dient der Überprüfung der korrekten Implementation gemessen an der Systemspezifikation (ist das System richtig gebaut worden?). Daneben spielt ein quantitativer Aspekt noch eine Rolle: die **Evaluierung**, d.h. die Beurteilung von Systemeigenschaften anhand von meßtechnisch erfaßten Daten (wie gut ist das System?).

Verifikation ist auf zwei Arten möglich: die sogenannte ***statische Verifikation*** (Lap90) versucht, unabhängig von der Existenz des eigentlichen Programmes eine formale Analyse des Systems durchzuführen, während für die ***dynamische Verifikation*** das Programm selbst ausgeführt wird. Gibt man konkrete Eingabewerte vor, spricht man im letzteren Falle von Verifikation durch Testen.

Allgemein wird man für ein softwareimplementiertes fehlertolerantes System besonders bemüht sein, die Korrektheit des Programmes nachzuweisen, da hiervon direkt die Systemeigenschaft Zuverlässigkeit abhängt. Da es unmöglich ist, ein komplexes Programm allein durch Eingabevariation erschöpfend zu testen, kommt dem Korrektheitsbeweis durch formale Verifikation besondere Bedeutung zu. Wenngleich dies als unumstritten gilt, finden sich nur wenige Veröffentlichungen, die von formaler Verifikation bei existierenden fehlertoleranten Systemen berichten. Positiv hervorzuheben ist hier das SIFT-System [MSS82], eine Arbeit, die auch in dieser Beziehung für fehlertolerante Systeme Pioniercharakter hat.

Für ATTEMPTO soll im Zusammenhang mit der vorliegenden Arbeit ebenfalls nur der konventionelle Ansatz zur Verifikation durch Testen verfolgt werden. Da ATTEMPTO kein kommerzielles Produkt ist, stünde nur geringer Gewinn durch einen formalen Korrektheitsbeweis erheblichem Aufwand gegenüber. Der Aufwand hängt ab von der Größe des Programmsystems, und macht bei der vorliegenden Systemkomplexität die Unterstützung durch Werkzeuge erforderlich. Tatsächlich sind auch die Vorraussetzungen für eine formale Verifikation relativ schlecht, da nur eine informelle Spezifikation existiert; darüberhinaus liegen nicht-verifizierbare Komponenten (LOS, alle Entwicklungswerkzeuge wie z.B. Compiler) vor. Sieht man einmal davon ab, würde sich eine klassische Programmverifikation[1] anbieten [BaKeWi90], da es im vorliegenden Fall ein konkretes imperatives Programm bereits gibt. Diese Methode verlangt eine *formale operationelle Spezifikation* (in Form von Vor- und Nachbedingungen für die Transformation von Ein- in Ausgabedaten durch das Programm) in einer Spezifikationssprache, die von entsprechenden Werkzeugen verstanden wird. Meist wird für die automatische Programmverifikation ein zweistufiges *Deduktionssystem* verwendet: eine erste Stufe (*Verification Condition Generator*) erzeugt aus der ursprünglichen Spezifikation und Implementation die Verifikationsbedingungen als ein System mathematisch-logischer Formeln (i.d.R. der Prädikatenlogik erster Stufe), für das ein zweites Deduktionssystem, der *automatische Beweiser,* den Nachweis der Gültigkeit erbringt.

Neben der operationellen Spezifikation gibt es auch solche, die abstrakte Datentypen beschreiben (*axiomatische, algebraische und konstruktive Spezifikation* [BaKeWi90]). Solche Spezifikationen werden auch durch deklarative Programmiersprachen (z.B. SIMULA, PROLOG) ermöglicht, die sich damit im Grunde als Spezifikationssprachen eignen. Die konstruktive Spe-

1. Ein solches klassisches Verfahren wurde auch in SIFT durchgeführt.

zifikation ist besonders interessant, da sie eine automatische Übersetzung in einen ausführbaren Algorithmus[1] erlaubt. Die Entscheidung für einen solchen Weg muß schon in einer frühen Phase der Entwicklung gefällt werden.

Für eine hohe Wirksamkeit sollten in allen Phasen der Software-Entstehung neben den klassischen Verfahren des Software-Engineerings zur Qualitätsicherung auch die Methoden für eine formale Spezifikation und Verifikation integriert werden. Besonders wichtig sind die Problemanalyse und der Entwurf, da hier die Realitätskonformität des Spezifikationsmodells am stärksten beeinflußt wird. Spezifikationsfehler sind ganz allgemein ein Problem der formalen Verifikation. Zudem kann formale Verifikation Testen nur ergänzen, nicht aber ersetzen.

Wegen der schon fortgeschrittenen Entwicklung beschränken wir uns also darauf, die Korrektheit der Implementation durch (funktionelles) Testen nachzuweisen. Um dies zu erleichtern, wurde schon beim Entwurf des ATTEMPTO-Systems auf gute Testbarkeitseigenschaften Wert gelegt. Sie resultiert, wie auch die Portabilität und Konfigurierbarkeit, aus der Modularität des Systems. Die Systembewertung betrachtet diese Modularitätseigenschaften im Detail, beurteilt außerdem noch die Eignung der Standardhardware für ein Fehlertoleranzkonzept, bewertet die Nutzbarkeit und die Zuverlässigkeit. In Kapitel 6.5 schließlich wird das Leistungsvermögen evaluiert.

6.1 Hardware

Die Anforderungen, die das ATTEMPTO-Fehlertoleranzkonzept an die Systemhardware stellt, sind vom Grundsatz her sehr gering. Eine der grundlegenden Systemprämissen ist ja die leichte Realisierbarkeit des Konzeptes mithilfe von Standard-Hardwarekomponenten. Konzept und Architekturprinzip lassen sich folgendermaßen hinsichtlich ihrer Hardwarerelevanz zusammenfassend charakterisieren (vgl. auch mit Kapitel 5.2 und Kapitel 5.3):

> Das Fehlertoleranzkonzept von ATTEMPTO basiert auf massiver Hardwareredundanz auf der Ebene von ganzen Rechnern als kleinste ersetzbare Einheiten (SRU smallest replaceable units) und ermöglicht in Verbindung mit verteilter Nachbarschaftsdiagnose Fehlermaskierung. Die Hardware ist ein homogenes Mehrrechnersystem bestehend aus N gleichartigen Rechnereinheiten (Knoten), die lose gekoppelt mit Nachrichten kommunizieren und den Benutzerjob autonom und asynchron bearbeiten. Diese Architektur als verteiltes System vermeidet zentrale Einheiten, die sich als zuverlässigkeitskritisch erweisen könnten (SPF *Single Point of Failure*). Einzige Ausnahme ist das gemeinsame Kommunikationsmedium, an das daher entsprechend hohe Zuverlässigkeitsanforderungen zu stellen sind, vorausgesetzt es kommen keine zusätzlichen Maßnahmen für Fehlertoleranz in Betracht, wie etwa redundante Auslegung (z.B. Doppelung) des Kommunikations-Busses.

Der einzige für den Fehlertoleranzbetrieb notwendige Hardwareeingriff bezieht sich auf die Zusammenschaltung der peripheren Ein/Ausgangsleitungen zu Bussen, die die globalen Ressourcen an die Systemkomponenten anbinden. Weitere Hardwarevoraussetzungen sind nicht zu erfüllen, da sowohl Fehlerdiagnose wie -maskierung durch Software realisiert ist.

Das ATTEMPTO-Konzept strebt darüberhinaus eine möglichst kostengünstige Realisierung

1. Möglicherweise allerdings nur in der Form als Interpreter.

durch die Verwendung von OEM-Produkten an, wie Single-Board-Computer, handelsübliche Platten, Standard-Bus- und Aufbausysteme. Diese tatsächliche Hardware-Implementierung bringt jedoch Einschränkungen des idealen Systemkonzeptes mit sich. Insbesondere bei örtlicher Konzentration unter Verwendung von Parallelbussystemen als Kopplungsmedium wie im vorliegenden Experimentalsystem wirken sich die Schwächen des gewählten Bussystems direkt auf die Fehlertoleranzeigenschaften des Gesamtsystems aus. Als SPF bestimmt er u.U. dominierend die Gesamtzuverlässigkeit, und es bedarf daher einer besonderen Bewertung seiner Eigenschaften.

6.1.1 Systembedeutung des VMEbusses

6.1.1.1 Speisung der SBC's

Alle SBC's befinden sich im selben 19"-Aufbaurahmen (Doppeleuropa). Die Versorgungsspannung wird zu den Rechnereinheiten über den einzigen Bus (*Backplane*) geführt. Die VMEbus-Spezifikation [VME87] sieht keine Separat-Speisung für einzelne Baugruppen vor; jedoch existiert die Möglichkeit einer zentralen Notspeisung (nur für eine Betriebsspannung). Der Ausfall der Netz-Wechselspannung wird signalisiert auf der ACFAIL-Leitung, falls der Systemcontroller über eine entsprechende (zentrale, optionale) Detektionseinrichtung verfügt (*Power Fail Detector*). Sie teilt den Spannungsausfall allen beteiligten Baugruppen am Bus mit, damit dort in einer verbleibenden Restzeit bis zum Gleichspannungszusammenbruch dem Benutzer überlassene Sicherungsaktionen durchgeführt werden können. Eine Notspeisung kann eingesetzt weren, um die Restzeit zu verlängern.

Um einen ununterbrochenen Betrieb zu gewährleisten, wäre im Bedarfsfall ein störungsfreier Ersatz aller Spannungsversorgungseinheiten nötig. Im Falle der Hauptversorgungsspannung 5V kann dieser Ersatz nur zentral erfolgen; eine Partitionierung des Gesamtbusses in separat versorgte Teilabschnitte käme einer Serialisierung von Funktionskomponenten gleich, die aus Fehlertoleranzgründen vermieden werden muß, wie noch gezeigt wird. Diese Möglichkeit wird auch wegen des zusätzlichen Aufwandes für eine Separatspeisung der aktiven Busterminierungsnetzwerke nicht näher in Betracht gezogen, die aus übertragungstechnischen Gründen notwendig sind. Passive Busterminierung würde diese Problem vereinfachen; sie ist jedoch aufgrund des beim VMEbus sehr hohen statischen Leistungsverbrauchs nicht wünschenswert.

6.1.1.2 Zentrale Systemsteuerungsfunktionen

Systemsteuerungsaufgaben haben beim VMEbus ausgesprochen zentralen Charakter. Neben der Betriebsspannungsüberwachung ist hier vor allem der zentrale Busarbiter zu erwähnen. Defekte an dieser Stelle können zu einer Busblockierung führen; sie sind jedoch wegen der verhältnismäßig geringen Hardwarekomplexität - im ATTEMTO-System wird ein *Single-Level-Arbiter* verwendet - nur gering wahrscheinlich. Weitere an der Arbitration beteiligte kritische Funktionseinheiten sind die auf den einzelnen Busmastern (SBC's) angeordneten *Bus-Requester*, die einerseits den Bus okkupieren können - dies ist sogar eine zulässige Betriebsart - , andererseits wegen der Single-Level-Arbitrierung in einer *Daisy-Chain* arbeiten. Diese Signalverkettung ist zuverlässigkeitskritisch, da sie z.B. den Totalausfall eines Knotens (etwa durch Unterbrechung dessen Spannungsversorgung) verbietet: permanente Fehler führen zu einer Isolation aller hinter dem Fehlerort in der Kette nachgeordneten Kommunikationseinheiten. Wegen der dynamischen Transition des BUSGRANT-Signals durch alle SBC's liegt auch eine Sensibilität gegenüber transienten Störungen vor, die jedoch mithilfe der üblichen Busfehlerdetektion (durch lokale *Watchdog-Timer*) erkannt, und durch entsprechende Ausnahmebehand-

lungen in den lokalen Betriebssystemen toleriert werden können.

6.1.1.3 Eignung als Kommunikationsmedium

Die Bewertung in dieser Hinsicht ist abhängig von

- den Möglichkeiten zur Fehlererkennung bzw. Korrektur,
- dem erzielbaren Durchsatz,
- der Unterstützung der Protokollimplementierung.

In der Spezifikation [VME87] sind keinerlei Fehlererkennungsmechanismen definiert. Selbst der Datentransport erfolgt ungesichert, d.h. ohne Unterstützung der Hardware, etwa durch eine Paritätssicherung. Daher sind entsprechende Softwaremechanismen in der Schicht des Datenübermittlungsprotokolls erforderlich. Sie basieren auf Checksummenbildung und -prüfung und weiteren Plausibilitätsüberprüfungen; außerdem ist ein robustes Busmapping zum Schutz vor Fehladressierung durch transiente Störer vorgesehen. Dieser Aufwand wirkt sich natürlich negativ auf die Nutz-Datenrate aus. Die reine physikalische Datentransfergeschwindigkeit kann sehr hoch sein - abhängig von Datenbusbreite und Buszyklusfrequenz und damit der jeweiligen CPU-Haupttaktfrequenz. Zudem ist ein Blocktransfer-Modus spezifiziert, den jedoch die vorhandene Hardware nicht unterstützt. Bei den vorliegenden Verhältnissen - 10 MHz Takt, 16 Bit- Datenbus - wären maximal 5 MByte/s transferierbar. Bei der Softwarerealisierung des Protokolls jedoch sind in ATTEMPTO nichteinmal 100 KByte/s zu erwarten[1]. Ein durch die Bus-Hardware unterstütztes *Message-Passing* für optimierten Kommunikationsdurchsatz wäre wünschenswert.

Essentiell für das dezentrale Systemkonzept von ATTEMPTO sind Broadcast-Botschaften. Point-to-Multipoint-Datentransfers gibt es jedoch beim VMEbus nicht. Als einzige Möglichkeit bleibt daher ein selektiver Botschafttransfer sequentiell zu allen Adressaten, wobei die gesamte Sendeaktion atomar gemacht werden muß. Die Realisierung in der Art, daß für die Dauer des Kettentransfers die Buszuteilung ununterbrochen festgehalten wird - der VMEbus gestattet dies - scheidet wegen der Gefahr der Dauerblockade aus. Deshalb wurde das ATTEMPTO-Kommunikationsprotokoll (Kapitel 5.4) definiert, das dedizierte Kommunikationsverbindungen (Mailboxen) und separate Broadcast-Interruptleitungen den Knoten zugeordnet. Die Hardwarevoraussetzungen für die Implementation dieses Protokolls sind:

- Dual-Ported-Ram auf den SBC's zur Realisierung der Mailboxen.

  Darunter wird verstanden, daß der lokale Datenspeicher der SBC's über zwei Wege adressierbar und lesbar bzw. schreibbar ist[2], nämlich über den lokalen Bus und von außen über den globalen VMEbus. Wünschenswert sind hardwareunterstützte Mechanismen zum Speicherschutz, um die Auswirkungen von Fehladressierungen in fremde Mailboxen zu unterbinden. Die Anzahl der Mailboxen ist jedoch sogar bei mittlerer Knotenzahl N schon groß (N^2, da neben den globalen auch die lokalen zu berücksichtigen sind). Daraus resultiert eine ebensogroße Anzahl von notwendigen MMU-Deskriptoren, die häufig jedoch nicht gegeben ist. In unserem Fall (MMU MOTOROLA 68451) sind die Deskriptoren in internen Registern des integrierten Bausteins gehalten, daher nicht erweiterbar und in ihrer Anzahl auf 32 beschränkt. Um die Speicherverwaltung des lokalen Be-

1. Genauere Leistungsdaten des Kommunikationssystems sind in Kapitel 6.5 zu finden.

2. Ein "echtes" Dual-Ported-Ram würde gleichzeitigen Zugriff über beide Wege erlauben, sofern nicht ein- und diesselbe Speicherzelle adressiert ist. Diese Eigenschaft ist i.d.R. bei großen Speichern nicht gegeben und auch für unseren Zweck nicht erforderlich.

triebssystems nicht zu stark zu behindern, wird im Experimentalsystem nur ein Deskriptor verwendet, um den Gesamt-Adreßbereich für die globalen Maiboxen dem lokalen Zugriff zu erschließen. Um Fehladressierungen, wenn nicht zu verhindern, so doch unwahrscheinlich zu machen, ist das robuste Adreßmapping eingesetzt worden (siehe Anhang). Die verbleibende Fehlerwahrscheinlichkeit ist wegen des Gesamt-Hammingabstandes von 7 äußerst gering.

- Nutzungsmöglichkeit für die Interruptbetriebsart *Point-to-Multipoint*.

  Im Gegensatz zu der spezifizierten VMEbus-Interruptbetriebsart, die nur eine Zweierbeziehung (1 Interrupter, 1 Interrupthandler) bei vektorisierten Interrupts vorsieht, gibt es hier zu einer Interruptquelle mehrere Interrupthandler; die Interruptpriorität identifiziert direkt die Quelle. Die Interruptquelle muß den eigengenerierten Interrupt selbst wegnehmen. Daß der Interrupt-Requester dazu in der Lage ist, kann nicht als selbstverständlich vorausgesetzt werden. Gleichzeitig ist es erforderlich, daß der Interrupt-Handler flankensensitiv arbeiten kann, d.h. den kurzen Interrupt-Puls speichern kann, damit keine Interrupts verloren gehen. Dies ist ebenfalls nicht selbstverständlich, da VMEbus-Interrupts statisch (pegelsensitiv) sind. Es muß daher im Einzelfall überprüft werden, ob die jeweilige Hardware die geforderten Eigenschaften des Interruptsystems besitzt. Bei der vorliegenden Hardware ist dies gegeben.

6.1.1.4 Alternativen

Die kritischen Eigenschaften des VMEbusses lassen eine redundante Auslegung des Kommunikationssystems wünschenswert erscheinen. Die vollständige Spezifikation der VMEbus-Norm definiert eine Reihe von zusätzlichen Sub-Bussen, die als Alternative in Frage kämen (z.B. VMSbus, VSB). Die vorhandene Hardware besitzt solche Optionen allerdings nicht. Ein denkbarer Ausweg wäre auch die Verwendung eines seriellen Ersatzkanals; ein solcher ist vorhanden. Diese Schnittstelle entfällt dann allerdings für andere Zwecke[1]. Zur Realisierung eines Busses, auf den die SBC's konkurrierend zugreifen können, wären allerdings Eingriffe in die Hardware nötig.

Bei einer neuerlichen Hardwareauswahl sollte auf das Vorhandensein solcher Ersatzbusse geachtet werden. Erst in jüngster Zeit hat die Integrationsdichte auf VMEbus-Karten soweit zugenommen, daß ein ausreichendes Angebot an solchen SBC's vorliegt. Gleichzeitig haben sich auch andere offene Bussysteme weiter verbreitet. Als relativ gut geeignet kann z.B. der Multibus-II gelten, der über einen dezentralen Arbitrationsmechanismus verfügt und ein hardwareunterstütztes Message- Passing bietet.

6.2 Modularitätseigenschaften

6.2.1 Portabilität

Zur Beurteilung der Portabilitätseigenschaft konnte auf praktische Erfahrung zurückgegriffen werden. Teile der in Modula-2 geschriebenen Fehlertoleranz-Systemsoftware waren als Simulationsversion schon vorhanden, einige Module konnten sogar unverändert übernommen werden. Die Simulation lief auf einer PDP11 unter UNIX Version 7 [Brau87]. Die größten

1. Z.Z. ist die Systemkonsole daran angebunden. In einer anderen Konfiguration befindet sich daran ein Drucker.

Schwierigkeiten im Zusammenhang mit der Portierung auf MOTOROLA 68010 und UNIX SYSTEM V traten aufgrund der unterschiedlichen Datenbusbreite der beiden Prozessoren auf, die unterschiedliche Wertebereiche von Standarddatentypen zur Folge hatten. Im allgemeinen jedoch waren diese Abhängigkeiten in systemnahen Bibliotheken gekapselt und damit gut abgegrenzt. Die mit der Portierung notwendig gewordenen Änderungen haben dazu geführt, solche - meist durch unaufmerksames Programmieren verursachten - Typabhängigkeiten zu bereinigen und können somit neuerliche Portierungen vereinfachen.

Im einzelnen wird die Portabilität durch folgende Eigenschaften des Programmsystems unterstützt:

- Maschinenunabhängigkeit ist bis auf den Kernadaptionsteil gegeben. Solche Abhängigkeiten beziehen sich auf das Programmiermodell des Prozessors. In unserem Zusammenhang nimmt es Einfluß auf den Prozeßkontext (zu speichernde Register), der im Systemcall-Handler des Kerns zu sichern ist, und auf das Vorhandensein eines Software-Trap-Mechanismus für einen zentralen Übergang vom Anwender- in den Kernadreßbereich. Trap-Mechanismen sind jedoch häufig zu finden auch bei anderen als den hier betrachteten Prozessoren und werden auch in vielen Betriebssystemen verwendet. Die Umlenkung geschieht durch Einlagerung einer Zwischenbehandlung (siehe Kapitel 5.5). Die Kern-Modifikation bleibt damit lokal und kontrollierbar.
- Selbst ohne die Kenntnis des Kern-Quellcodes kann wegen der o.a. Eigenschaften die Modifikation leicht vorgenommen worden, was anhand des Experimentalsystems verifiziert wurde. Wegen des begrenzten Umfangs war eine Rückbildung des C-Quellcodes basierend auf einer Dissassemblierung des Maschinencodes tragbar und konnte erfolgreich durchgeführt werden. Für das Basisbetriebssystems des Experimentalsystems (Uniplus+ SYSTEM V) lagen nur sogenannte “Rekonfigurationsrechte” vor, d.h. neben einem monolthischen Kernobjekt mit Symboltabelle waren nur Quellen für die verwendeten Hardware-Gerätetreiber vorhanden. Zum Verständis der Semantik des Systemcall-Handlers im UNIX-Kern war die Unterstützung durch entsprechende allgemeine Fachliteratur [Bach86] sowie die Quellen von fremden UNIX-Systemen ausreichend. Die Gleichheit von Variablen- und Funktionssymbolen sowie weitreichende C-Quellcode-Kompatibilität scheint in UNIX selbst über lange Generationsunterschiede gewährleistet zu sein. So wurde zwischen einem Version 7 - UNIX für eine PDP11 und dem SYSTEM V - UNIX noch sehr starke Ähnlichkeit des System Call Handlers festgestellt.
- Die Botschafts-Schnittstelle zum FTI-Prozeß abstrahiert von Kerndetails und Geräteeigenschaften. Der Modularitätscharakter (bedingt durch die Kapselung der Handlerprozesse als UNIX-Prozesse) unterstützt die Anpassung an die jeweiligen Verhältnisse durch Komplett-Austausch.
- Die Verwendung von MODULA-2 als Hochsprache mit ausgeprägtem Modulcharakter erleichtert eine hierarchische Untergliederung von funktional aufeinander aufbauenden Programmobjekten. So gelingt es leicht, Systemabhängigkeiten in Laufzeitbibliotheken auf der untersten Ebene zu halten und diese im Bedarfsfall auszutauschen.

6.2.2 Testbarkeit

Die Programm-Verifikation durch Testen ist ein wichtiger Teil des Softwareentstehungs- und Lebenssyklus. Bei fehlertoleranten Systemen ist die frühe Testphase zur Beseitigung von Entwurfs- unf Entwicklungsfehlern vor der eigentlichen Inbetriebnahme von größter Bedeutung, da die Zuverlässigkeit als geforderte Systemeigenschaft natürlich direkt von den korrekt imp-

lementierten Mechanismen abhängt.

Die Testbarkeit wird schon durch Entscheidungen der Konzeptions- und Entwurfsphase beeinflußt. Modularität vereinfacht, als ein Mittel zur funktionalen Abgrenzung und hierarchischen Systemgliederung, sowohl die Spezifikation als auch die Verifikation und schafft so die Basis für einen *Top-Down*-Entwurf. Dieses Prinzip wurde in ATTEMPTO konsequent angewandt. Programm- und Prozeßmodule der FTL kapseln rückwirkungsfrei einzelne klar definierte Funktionen, die auf makroskopischer Ebene anhand des Ein/AusgabeVerhaltens leicht verifizierbar sind. Die Begrenzung von Modulinteraktionen auf klare Schnittstellen gestattet darüberhinaus auch eine einfache Testumgebung. Schon entwicklungsbegleitend standen Werkzeuge zum Modultest zur Verfügung. Zur Verifizierung der Fehlertoleranzfunktionen wurden darüberhinaus Fehler in die Module selbst injiziert, bzw. dort simuliert.

Entsprechend der Hierarchie von Programmobjekten (siehe dazu Kapitel 5.3.3) wurde ein abgestuftes Testkonzept entwickelt, das entwicklungsbegleitend zunächst die Bausteinfunktionen (“*bottom-up*” beginnend mit Einzelmodulen, hin zu komplexeren Modulobjekten) einzeln an ihren Schnittstellen und später im Zusammenwirken überprüft. Daß die meisten Bibliotheksmodule ihre Daten kapseln, erleichtert deren Separat-Test. Die Testumgebung für Prozeßobjekte zeigt Abb. 45. Sie kann gleichzeitig auch zum Test der lokalen Betriebssystemerweiterungen (Kernel-Device, Port-Device) dienen.

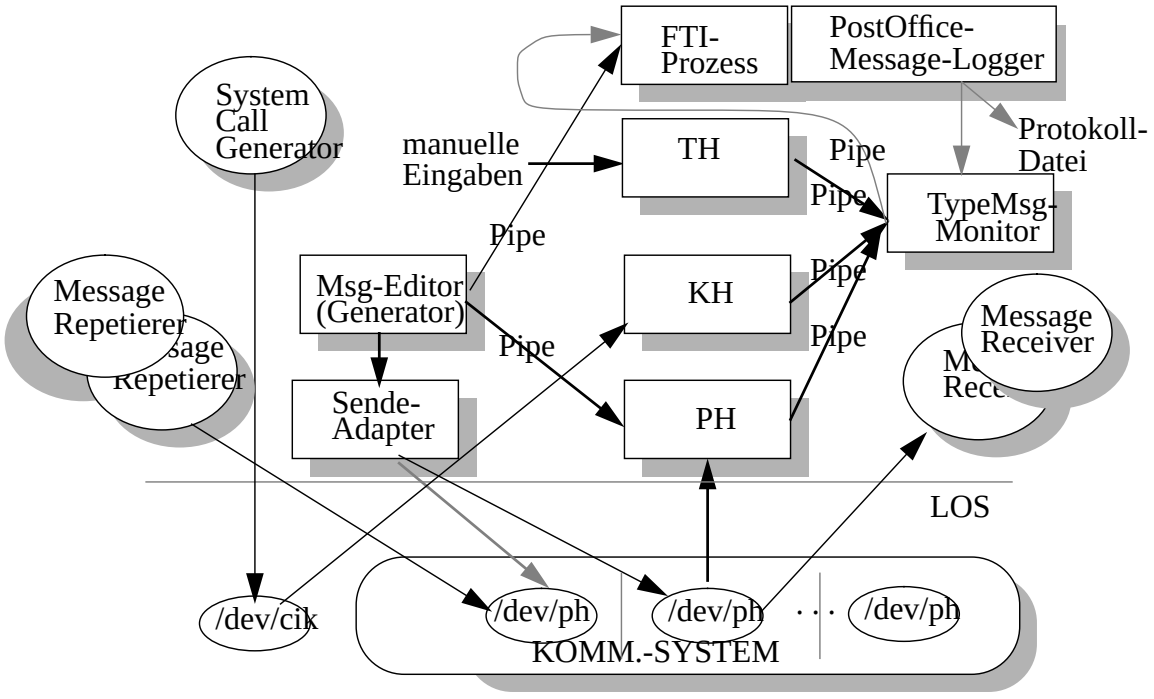


Abb. 45 Allgemeine Testumgebung

Die UNIX-Prozesse der FTL werden für den Fehlertoleranzbetrieb während der Systeminitialisierung so aufgesetzt, daß ihre einzige Eingabeschnittstelle mit ihrem Standardeingabekanal verknüpft ist. Alle Handler-Prozesse besitzen darüberhinaus nur eine einzige Ausgabeschnittstelle über ihren Standardausgabekanal, der im Betrieb mit der gemeinsamen Eingabe-Pipe des lokalen FTI-Prozesses verbunden ist. Handler stellen somit entsprechend der UNIX-Terminologie Filter dar, und können so als Bausteine leicht (auf Kommandoebene) mit anderen verkettet werden. Zusammen mit geeigneten Kettenendelementen (Testgeneratoren, Testmonitoren)

kann auf diese Weise baukastenartig eine sehr wirkungsvolle Testumgebung aufgebaut werden, wobei der Aufwand zur Erstellung der Testwerkzeuge wegen deren vielseitiger Verwendbarkeit in Grenzen bleibt. Dieses Prinzip wird auch durch die Homogenität des Nachrichtenformats unterstützt. So ist es z.B. möglich, den üblichen Testnachrichtengenerator unabhängig vom globalen Kommunikationssystem zum Test des Porthandler -Prozesses PH zu verwenden. Durch weitere Filterelemente (Sendeadapter) kann derselbe Generator jedoch auch den realen Kommunikationsweg nutzen. Umgekehrt entsteht durch die Verwendung der gleichen Testhilfsmittel eine Testumgebung zum Testen des globalen Kommunikationssystems. Dessen Test wurde in mehreren Stufen vorgenommen: rein lokal, in Zweierbeziehungen zwischen Knoten (*back-to-back*) und im Broadcastbetrieb. Zur Erzeugung von Streßbelastung wurden noch besondere parametrierbare Testgeneratoren (Message- Repetierer) und Nachrichtensenken (Message-Receiver) zugleich an allen Endpunkten des Kommunikationssystems aufgesetzt. Diese Testwerkzeuge selbst konnten zuvor ohne das Kommunikationssystem mithilfe der zuvor beschriebenen Testumgebung getestet werden.

Zum Test des lokalen FTI-Prozesses wurde zunächst ein Rahmen geschaffen durch die Verifikation der Systemsteuerfunktionen des Post-Office (Kapitel 5.3.5), das den Nachrichtentransport und die Steuerung des Leichtgewichtsprozeß-Systems vornimmt. Das Postoffice stellt lokal eine zentrale Schnittstelle zur Beobachtung des gesamten Meldungsverkehrs zu Verfügung (Message-Logger) und spielt daher für den späteren Intergrationstest eine bedeutende Rolle. Diesselbe Schnittstelle leistet jedoch auch schon beim Test der Clerk-Module gute Dienste. In den Rahmen eingepaßt konnten mithilfe des Testmeldungsgenerators deren Modulfunktionen anhand des Schnittstellenverhaltens verifiziert werden.

Die Integration der Prozeßmodule wurde zweistufig durchgeführt. Es wurden getestet: 1. alle Schwergewichtsprozesse zusammengenommen als einzelne lokale Fehlertoleranzinstanz und 2. alle lokalen Instanzen im Multiprozessorsystem durch echten Fehlertoleranzbetrieb.

Neben der funktionellen Überprüfung auf Korrektheit der erbrachten Modulfunktionen spielt deren Verhalten bei Einwirkung von Fehlern sowohl an den externen Schnittstellen (Unzulässige Eingabewerte etc.) wie auch intern eine Rolle. Wegen der Unmöglichkeit des vollständigen Testens sind hier Testfälle zu klassifizieren und einzelne repräsentative Fehlerfälle zu simulieren.

Zusammengefaßt wurden folgende Fehlerfälle simuliert: (verwendet zu Modul- und Integrationstests)

- **Im globalen Kommunikationssystem:**

  Busfehler , Checksummenverfälschungen und andere, Plausibilitätstests unterlaufende Veränderungen, Botschaftsverfälschungen i.a. (Format und Inhalt von Protokollsteuerungselementen) zum Check auf Laufzeitfehler; Unterlassungen, Sreß (hohe Verkehrsbelastung): Test auf Verklemmung, eingehaltene Reihenfolge, Nachrichtenverlust.
- **Bei FTL- Interaktionen** (durch internen Moduleingriff simuliert):

  Unterlassungsfehler, veränderte Botschaftsequenzen während der Bearbeitung von Funktionsabläufen.
- **Für den Modultest:**

  Falsche Nachrichtenformate und -typen. Längenmodifikationen.

Daneben wurde auch begrenzt eine Fehler-Injektion vorgenommen (verwendet zum Test kern-

residenter Software wie modifizierte Exception-Handler, Kommunikationstreiber, und während des laufenden Fehlertoleranzbetriebs). Sie bestand nur in der Außerbetriebnahme von ganzen Rechnereinheiten (Reset) und von Teilkomponenten, wie z.B. den Mailbox-Adreß-Mapping, und dem Stop lokaler Systemprozesse.

Abschließend kann die **Testbarkeit** des Modula-Programmsystems aus den vorliegenden Erfahrungen folgendermaßen zusammenfassend bewertet werden:

- Die Modulabgrenzung und -zuordnung zu separaten Prozessen erlaubt eine gute Spezifikation der Modulfunktion und gestattet die leichte Überprüfbarkeit anhand des Ein/Ausgabeverhaltens.
- Bei der gewählten Struktur kann eine leistungsfähige Testumgebung einfach hergestellt werden.
- Eine Propagierung von Implementationsfehlern über Modulgrenzen hinaus ist sehr erschwert, sodaß der Programmverifikation ein einfaches Fehlermodell zugrundegelegt werden kann.
- Die Kausalität der Prozeßinteraktion ist leicht erkennbar durch Examinierung von Nachrichten-Traces. Die Struktur des Softwaresystems bietet hierzu sehr gute Unterstützung.
- Integrationstests von modulübergreifenden Funktionen sind nicht immer leicht durchzuführen aufgrund des hohen Grades an Asynchronität in diesem parallelverarbeitenden System. Dadurch gestaltet sich die Reproduktion von Testsituationen schwieriger. Dem muß durch repetatives Testen entgegnet werden. Die indeterministische Durchmischung von lokalen und globalen Botschaften führt zu Schwierigkeiten in der Realisierung von modulinternen Kausalitäten. So können z.B. globale Bewerbungsbotschaften Jobs betreffen, für die lokal noch keine Referenz existiert[1]. In solchen Fällen sind Vorbehaltsaktionen zeitüberwacht auszuführen. Sorgfältige Testfallauswahl für Reihenfolge-, Zeit- und Unterlassungsfehler im Kommunikationssystem ist in diesem Zusammenhang unerläßlich.
- Für Betriebssystemprogrammierungsaufgaben der LOS-Adaption und des Kommunikations- und Testtreibers unter UNIX sind die Testbarkeitsrandbedingung der Entwicklungsumgebung unter UNIX vorgegeben. Ihre Bewertung ist nicht Gegenstand der vorangegangenen Betrachtungen. Zur Verbesserung der Entwicklungs- wie auch Testumgebung wurden zusätzliche Werkzeuge geschaffen. Vor allem ist hier die Erweiterung des Betriebssystems für das dynamische - das heißt während des Betriebs mögliche - Laden von kompletten Treibern und Treiberfunktionen [Gü88a], und eine komfortable Werkzeug-Unterstützung (TOAST Tracer Output-Analysis-System [Hamm91]) für ein Echtzeit-Debugging, basierend auf einem Hardwaremonitor (Tracer) zu erwähnen. Dasselbe Werkzeug wird auch zur Erfassung von Leistungsdaten benutzt (siehe Kapitel 6.5).

6.2.3 Konfigurierbarkeit

Die Modulkonfigurierbarkeit ist von Bedeutung für die leichte Anpaßbarkeit von Systembesonderheiten und zur Veränderung von Systemeigenschaften, z.B. auch solche, die die Fehler-

1. Die fehlende Berücksichtigung dieser Möglichkeit in der Implementation wurde während der Testphase erkannt. Eine formale Programmverifikation hätte dies - weil auch in der Spezifikation nicht definiert - schwerlich aufgedeckt.

toleranz und damit Zuverlässigkeit selbst beeinflussen. Diese Aspekt ist besonders bei Experimentalsystemen interessant, wie z.B. beim ATTEMPTO-System.

Als ***austauschbare Grundmechanismen für Fehlertoleranz*** *kommen in Betracht:*

- Das Diagnoseverfahren: Hier bieten sich Variationen in den Diagnosemodellen und den verteilten Algorithmen zur Systemdiagnose an. Interessant sind nicht nur geringkomplexe Diagnosealgorithmen, sondern auch solche, die im hohen Maße robust bzw. fehlertolerant sind. Experimente sind in diesem Zusammenhang geplant.
- Zusatzmodule zur Fehlerbehandlung. Das ATTEMPTO-System ist in erster Linie ein reinrassig fehlermaskierendes System. Aufgrund der bei der Maskierung anfallenden Diagnoseinformation könnte leicht eine Rekonfiguration (z.B. im Sinne einer selbstreinigenden Redundanz [Lala85]), zumindest aber eine Komponentenisolation nach erkanntem permanenten Knotenausfall, bzw. eine Reintegration von durch transiente Fehler am Betrieb gehinderten Komponenten für weitere Jobs durchgeführt werden. Weiterhin ist eine Unterstützung zur Reparatur des Systems interessant, wie sie auch für das Teilprojekt ATTEMPTO-2 durch die Anbindung and das Diagnoseexpertensystem DIAMONT verwirklicht wurde. An der Schnittstelle können vielfältige Diagnosedaten übermittelt werden, dazu gehören z.B. zweifelsfrei erkannte Systemdefekte, Verdachtsmomente oder feinkörnigere Diagnoseresultate als Ergebnis von on-line eigenständig durchgeführten Selbsttests oder Fremdtests von Teilfunktionen der einzelnen Knotenrechner.
- Grundsätzlich ist eine Variation der hardwareabhängigen Selbsttestfunktionen interessant, nicht nur wegen der Anpassung an die konkrete Hardware, sondern auch für veränderte Testtiefe, die Zuverlässigkeit und Leistungsverhalten beeinflußt.
- Implizite Fehlererkennungsmaßnahmen der Hardware und des lokalen Wirts-Betriebssystems können in die Diagnose auf Systemebene miteinwirken. Dies kann dazu benutzt werden, im Fehlerfall die Stetigkeit der Diensterbringung zu verbessern, indem die Defektkomponenten - falls sie dazu im Rahmen der Ausnahmebehandlung in der Lage sind - ihren Ausfall ankündigen. Die restlichen Knoten im System wären dann nicht mehr darauf angewiesen, den Unterlassungsausfall durch eine Zeitüberwachung zu detektieren.
- Erweiterte Funktionalität als zusätzliche Mechanismen,

  - weitere Fehlertoleranzdienste, z.B. fehlertolerantes Filesystem,
  - Verwaltung von Redundanz im Kommunikationssystem (Ersatzwege)

Zur Berücksichtigung von ***Besonderheiten des Wirts-Betriebssystems*** muß konfigurierbar sein:

- Low-Level-Module zur systemseitigen Anpassung der Laufzeitumgebung für das MODULA-System,
- Systemcall-Handler, d.h. die eigentliche Kernmodifikation in gekapselter systemangepaßter Form,
- Kernel-Handler, als fehlertoleranzseitige Anpassung an das lokale Kommunikationssystem,
- lokale Interprozeßkommunikationsmechanismen zur Interaktion von Schwergewichtsprozessen,
- Ausnahmebehandlungen im Kern,
- Handler als Leichtgewichtsprozesse, falls das Betriebssystem dies unterstützt. Dies wird sich positiv auf das Laufzeitverhalten auswirken.

Letztlich muß auch eine ***Anpassung an die jeweilige System-Hardware*** vorgenommen werden können:

- Kommunikationstreiber zur Anpassung an unterschiedliche Kommunikationsmedien,
- globale Schnittstellen, z.B. redundante Plattenlaufwerke, Terminalankopplung, weitere Ressourcen des gleichen oder anderen Typs,
- Hardwareerweiterungen für erhöhten Maximalfehlertoleranzgrad (mehr Knoten).

Interessant ist auch ein ***Freiheitsgrad in der Systemrepräsentation*** nach außen. Dies betrifft allgemein Anzeige- und Bedieneigenschaften.

6.3 Nutzbarkeit

Das ATTEMPTO-System hat entsprechend seiner Aufgabe als fehlertolerantes und parallelverarbeitendes System zwei Aspekte der Nutzbarkeit. ***Als allgemeines verteiltes System*** treten Probleme mit der Manipulation mehrerer Programmobjekte auf und solche, die den Betriebskontext betreffen. Obwohl der Ausführungsort für ein Programm nicht festliegt, erwartet der Benutzer eine entsprechende einheitliche Laufzeitumgebung, die er nicht erst zuvor explizit für jeden Knoten festlegen möchte. Dazu gehört ein automatischer Abgleich von *Environment*-Daten und eine einheitliche Objekt-Referenzierung. Damit tauchen ganz allgemeine Probleme eines verteilten Dateisystems auf. Daneben wird es erwünscht sein, globale Systemressourcen, die nicht fehlertolerant begrenzt im System vorhanden sind, wirtschaftlich zu nutzen und in diesem Sinne zu verwalten. Nicht zuletzt wird auch der Einbenutzer-Charakter und die fehlende Unterstützung für verteilte Applikationen als Beschränkung empfunden werden. Die Lösung dieser Probleme liegt jedoch außerhalb des Blickwinkels der vorliegenden Arbeit. Bei der Realisierung werden praktische Probleme im Zusammenhang mit der Integration der Fehlertoleranzeigenschaften auftreten.

Die Nutzbarkeit eines parallel verarbeitenden interaktiven Systems wird stark geprägt durch die Ausgestaltung der jeweiligen Benutzeröberflache. In ATTEMPTO ist Parallelität auf Jobebene möglich. Jobs sind eigenständige Prozeßgruppen, die geschlossen auf Rechnerknoten laufen. Sie können jederzeit Eingaben vom Benutzer verlangen oder Ausgaben an die Systemumwelt tätigen. Im Experimentalsystem steuert der Benutzer das System über eine asynchrone Terminal-Schnittstelle. Hier, in einem vom Fehlertoleranzbetrieb unabhängigen Fehlerbereich, ist ein -im einfachsten Fall unintelligentes - Terminal angeschlossen. Dies führt naturgemäß zu Schwierigkeiten bei der Kontrolle von mehr als einem Job, falls nicht die Fehlertoleranzschicht selbst Funktionen einer Fenstersteuerung beinhaltet, die die Ein/Ausgaben den jeweiligen Jobs in gesonderten Fenstern zuordnet. Aus Gründen der Robustheit sind diese Funktionen verteilt auszuführen, da sie sich im Fehlermaskierungsbereich befinden. Die Verhältnisse vereinfachen sich, wenn diese Funktionen dem Endgerät selbst angelastet werden. Selbstverständlich wird dies die Komplexität und damit die Fehleranfälligkeit des Terminals beeinflussen.

Die Nutzbarkeit ***als fehlertolerantes System*** ist zum Teil durch die gleichen Probleme eingeschränkt. Eine einheitliche Laufzeitumgebung für die verteilten Prozeßreplikate muß immer hergestellt sein, um den deterministischen Programm-Gleichlauf nicht zu gefährden. Die Probleme eines verteilten Dateisystems werden noch zusätzlich durch Fehlertoleranzanforderungen erschwert. Gleichzeitig sind jedoch durch das Fehlertoleranzkonzept Mechanismen geschaffen worden, die gute Ansätze zur Lösung dieser Probleme bieten. Es existieren bereits Nachrichtenwege und Systemverwaltungsinstanzen, die für zusätzliche Aufgaben

herangezogen werden könnten, außerdem gestattet die modulare Systemarchitektur leicht Ergänzungen. Die Fehlertoleranz eines verteilten Dateisystems kann sich auf vorhandene Maskierungsmechanismen abstützen.

Die Einschränkungen der Nutzbarkeit sind letztendlich auch Einschränkungen für die konzeptionell geforderte Transparenz der Fehlertoleranz für Anwendungen. Programme können nicht den vollen Satz der System-Funktionen verwenden. Primär zielt die Prototyp-Implementation des ATTEMPTO-Systems nur auf die Realisierung der Grundfunktionen ab (Replikationsverwaltung, Eingabe- und Ausgabeüberwachung); nur für diese ist auch eine Portierbarkeit angestrebt und allgemein möglich. Für betriebssystemangepaßte Erweiterungen ist allerdings durch den sehr flexiblen Eingriffsmechanismus auf Betriebssystemfunktionen an der Systemaufrufschnittstelle Vorsorge getroffen worden.

Aus Aufwandsabschätzungen und praktischen Versuchen zur Verbesserung der Transparenzeigenschaften wurde deutlich:

- Die Transparenz der Grundfunktionen kann überschaubar realisiert werden. Diese Funktionen sind leicht nachzuvollziehen und können so zur Vertrauensbildung in die Wirksamkeit der Mechanismen beitragen.
- Es muß für jeden einzelnen Systemcall sorgfältig geprüft werden, inwieweit 1) besondere Maßnahmen zur Aufrechterhaltung der Fehlertoleranzeigenschaft erforderlich sind, und 2) der allgemeine verteilte Systemcharakter schon Sonderbehandlung erfordert.
- Die erforderlichen Sonderbehandlungen können leicht sehr aufwendig werden. Insbesondere kann es notwendig werden, Aktionen des lokalen Betriebssystems auf der FTL-Ebene zur globalen Synchronisation nachzuempfinden (d.h. doppelt auszuführen), wenn nicht gar nachträglich zu korrigieren. Den vollen Satz von Systemaufrufen für den Fehlertoleranzbetrieb zuzulassen kostet so womöglich mehr Aufwand als das lokale Betriebssystem selbst benötigt, d.h. die Betriebssystemerweiterung übertrifft den Umfang des Basisbetriebssystems.
- Es treten Probleme mit indeterministischem Verhalten der Prozeßreplikate an den Systemaufruf-Schnittstellen auf. Konzeptionell war eine Votierung über Eingabedaten nicht vorgesehen, da diese physikalisch parallel zu den Replikaten geführt werden. Indeterminismus, hauptsächlich die Anzahl der gelesenen Zeichen betreffend, wird unter bestimmten Betriebsbedingungen erst durch das lokale Betriebssystem, z.B. in dessen asynchronem Terminaltreiber, eingebracht. Das Grundkonzept hat ein Szenario für den Aktionsgleichlauf definiert, dessen Randbedingungen im praktischen Gebrauch des Systems (als interaktives, potentiell parallelverarbeitendes) unrealistisch sind; z.B. wurde Raw-Mode-Terminal-Input und Non-Blocking-I/O für interaktive Programme nicht berücksichtigt. In solchen Fällen ist ein Übereinstimmungsprotokoll zum Abgleich von Eingabedaten und -frequenz unumgänglich. Obwohl machbar, ist dies wegen der erheblichen Kosten unerwünscht.
- Signalbearbeitung macht erhebliche Probleme. Asynchrone Signale sind immanent indeterministische Ereignisse. Dürfen sie den Programmfluß der Prozeßreplikate beeinflussen, wie z.B. in UNIX möglich durch anwendungsspezifische Signalhandler, ist Synchronisationsverlust auch im fehlerfreien Fall zu befürchten. Der Programmfluß fehlertoleranter Systeme mit aktiver statischer Redundanz, die lose synchronisiert und dazu auf hohem Implementationsniveau software-realisiert sind, kann so gut wie nicht synchronisert werden. Dazu ist der Einsatz zusätzlicher dynamischer Redundanztechniken erforderlich (z.B. Checkpointing, verbunden mit einem Mechanismus zur interaktiven Konsi-

stenz), oder die (geregelte) lose Synchronisation muß ersetzt werden durch eine starre Taktsynchronisation oder eine globale Zeitscheibensynchronisation, damit das Signal zustandsgleich zugeführt (eingephast) werden kann.

Zur Zeit ist nur eingeschränkte Transparenz gegeben.

Die variablen Fehlertoleranzeigenschaften beeinflussen im besonderen die Benutzeroberfläche, für die besondere Mittel zur Handhabung bereitgestellt werden müssen. Zuerst muß die Zuverlässigkeitsanforderung selbst ausgedrückt werden können. Die Erweiterung der Shell-Syntax dafür in der Form "*meinprogramm #Fehlertoleranzgrad#*" wurde bereits im Kapitel 5.6.1 vorgestellt. Im Zusammenhang mit einer fenstergesteuerten Graphik-Benutzeroberfläche sollten dafür allerdings auch die erweiterten darstellerischen Mittel genutzt werden. Eine solche Benutzeroberfläche wurde im Rahmen einer Studienarbeit für sun-Workstations unter sunos4.1.2 unter Zuhilfenahme des *Openlook*-Toolkits *xview* erstellt [Klau92]. Abb. 46 zeigt die optische Ausgestaltung der ATTEMPTO-Benutzeroberfläche.

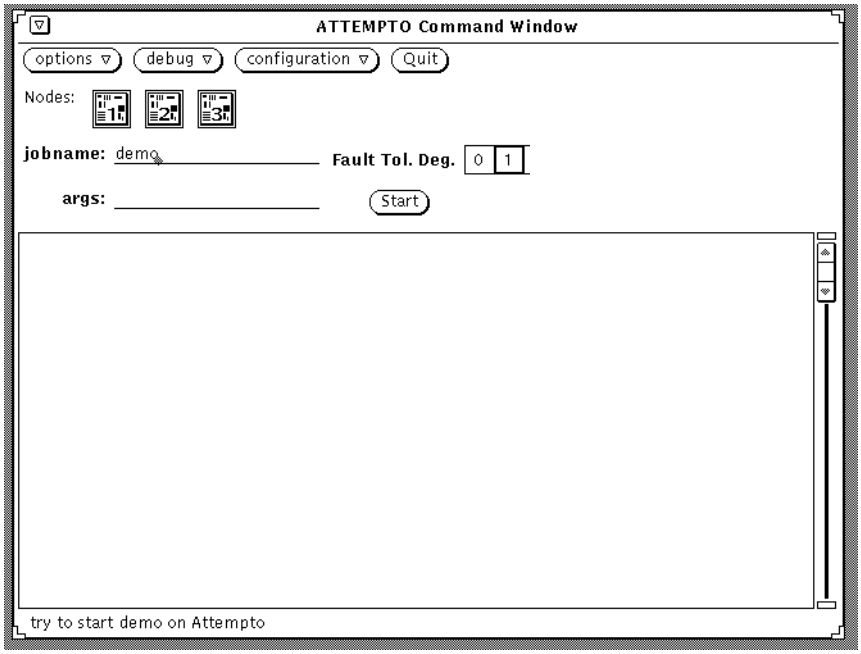


Abb. 46 ZentralesKommandofenster

Die Aufgabe dieses Kommandofensters ist die Hilfe beim Starten von Jobs sowie die Beeinflussung von Umgebungsvariablen der Kommandoschell auf den ATTEMPTO-Knoten. Dazu steht in der unteren Hälfte ein dem Hauptfenster untergeordnetes Textfensterobjekt zur Verfügung, das allgemein für Shell-Eingaben gedacht ist. Hier können Jobs in der oben beschriebenen Weise genau wie bei einem unintelligenten Terminal aufgerufen werden. Darüberhinaus gibt es noch die Möglichkeit, fehlertoleranzbezogene Anzeige- und Eingabefelder in der oberen Hälfte zu benutzen in Verbindung mit der Tatstatur und einem Mauszeiger, z.B. zur Auswahl des Fehlertoleranzgrades. Vorteilhaft an diesem Verfahren ist, daß unzulässige Eingabewerte bereits vor der Übersendung an die Fehlertoleranzschicht unterdrückt werden können. Das Fenster enthält u.a. auch eine Statusanzeige für die aktuell zur Verfügung stehenden Knoten. Ganz unten im äußeren Rahmen meldet das lokale Programmsystem die Einleitung von Aktionen, hauptsächlich um etwaige Wartezeiten, z.B. beim entfernten Jobstart, zu begründen.

Nach dem erfolgreichen Start, den die Fehlertoleranzschicht zurückmeldet, wird ein Job-Fen-

ster geöffnet. Für den Fall, daß bereits ein freies existiert, wird jedoch dieses verwendet. Jobfenster werden nach Beendigung eines Jobs nicht geschlossen, um die Ausgabe zu konservieren. Stattdessen wird im Fenster das Jobende kenntlich gemacht, sodaß es für einen neuen Job genutzt werden kann. Ein- und Ausgaben werden wie üblich im Textfenster vorgenommen. Alle Textfenster-Objekte speichern eine begrenzte Menge von Zeichen. Der Rahmen enthält noch zusätzliche jobbezogene Informationen: Fehlertoleranzgrad, Job-Identifizierungsnummer, Ausführungsorte (Jobkollegenknoten). Hier gibt es auch die Möglichkeit, dem Job Signale zuzuführen.

Das System zur Fensterverwaltung steht in Nachrichtenverbindung mit den Clerks der Fehlertoleranzschicht. Dazu wurde eine Kommunikationsprotokoll festgelegt. Alle Ein- und Ausgaben sowie auch Kontroll- und Statusmeldungen sind in Nachrichtenpakete verpackt und durch Typ und Jobnummer kenntlich gemacht. Kommandos sind der Shell, und damit dem Job 0 zugeordnet.

Die Benutzeroberfläche bietet noch weitere Möglichkeiten, die jedoch nicht direkt im Zusammenhang mit dem Fehlertoleranzbetrieb stehen. So können Betriebsparameter (z.B. der seriellen Leitung zum ATTEMPTO-System) und Schnittstellenkonfigurationen menüunterstützt (*Pull-Down-Menü*s) eingestellt werden. Eine Option bezieht sich auf den Zugriff auf die Systemkonsolen der einzelnen ATTEMPTO-Knoten. Hier kann ein Systemstart vorgenommen werden, Debug-Information übertragen, oder aber zusätzliche Status- und Diagnosedaten abgerufen werden. Alternativ kann das Diagnosesystem DIAMONT diese Zugänge zur Testbeauftragung verwenden. Gleichzeitig ist damit auch eine einfache (nicht fehlertolerante) Einzelnutzung der ATTEMPTO-Knoten möglich.

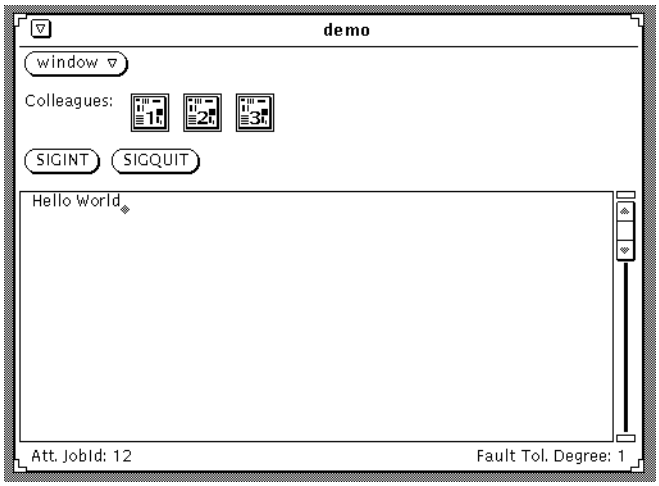


Abb. 47 Ein Jobfenster

6.4 Zuverlässigkeitsbetrachtungen

Redundanzaufwand und dessen Nutzung durch das Fehlertoleranzverfahren führen zu einem synergetischen Effekt auf die Systemzuverlässigkeit: ein fehlertolerantes Systems ist zuverlässiger als die Zuverlässigkeit seiner Komponenten. Ohne dies wäre es unmöglich, mit Standard-Hardware einen Zuverlässigkeitsgewinn zu erzielen. Dieser Effekt ist allerdings zeitabhängig. Bei einem maskierenden System kommt es beim Betrieb über eine bestimmte Nutzungsdauer hinaus sogar zu dem gegenteiligen Effekt, daß sich der Redundanzeinsatz schädlich auswirkt.

Die homogene Grundstruktur des ATTEMPTO-Systems vereinfacht die Ermittlung der Gesamtzuverlässigkeit als k-aus-n-Struktur mit Komponenten gleicher Überlebenswahrscheinlichkeit $R(t) = e^{-\lambda t}$ (λ = const. ist die Ausfallrate). Die Basis des Maskierungsverfahrens ist die Fehlersymptomverschiedenheit im Vergleichstestmodell. Aus diesem Grund ähnelt das System für die Ausführungsdauer eines fehlertoleranten Jobs einem 2-aus-n-System, wobei n=f+2[1]. Vorausgesetzt, der Vergleichstest entdeckt alle Fehler, und die restlichen Systemkomponenten außer den Rechnereinheiten selbst - hier ist vor allem der Kommunikationsbus zu nennen - sind ideal zuverlässig, ergibt sich aus kombinatorischen Zuverlässigkeitsmodellen die Überlebenswahrscheinlichkeit des fehlertoleranten Systems *Rges(t)* in Abhängigkeit vom Fehlertoleranzgrad f (Binomial- bzw. Bernoulli-Verteilung) ([Triv82]) zu:

$$Rges(f,t) = \sum_{j=2}^{f+2} \binom{f+2}{j} (R(t))^{j} (1-R(t))^{f+2-j}$$

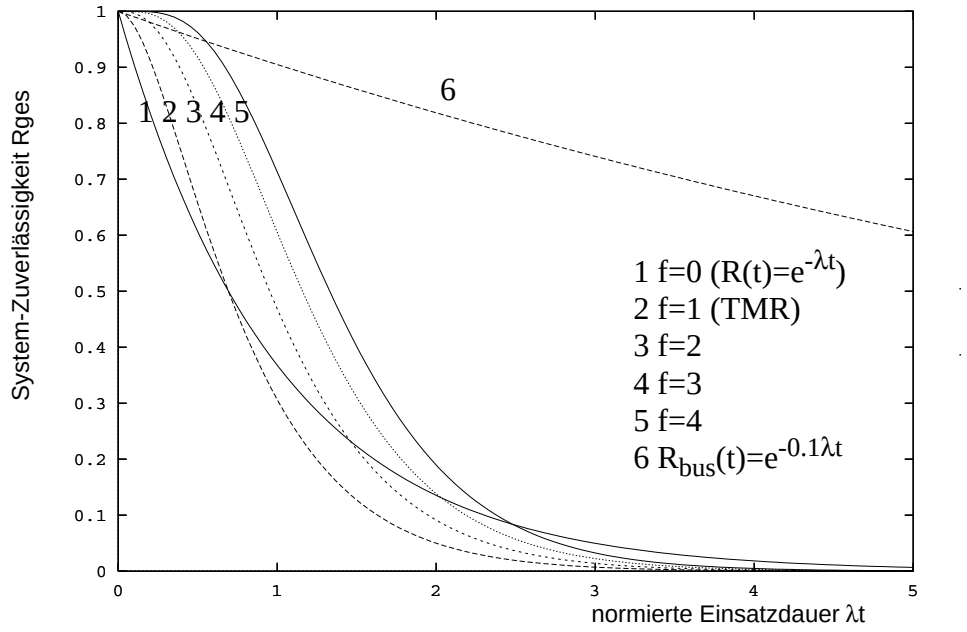


Abb. 48: Fehlertoleranzgradabhängiger Zuverlässigkeitsgewinn

Die Wirkung des Fehlertoleranzverfahrens auf die System-Überlebenswahrscheinlichkeit läßt sich aus Abb. 48: ersehen. Man erkennt die zuvor schon herausgestellte Zeitabhängigkeit des Zuverlässigkeitsgewinns. Der anfänglich hohe Gewinn schwindet mit steigender Einsatzdauer. So bewirkt der Aufwand eines TMR-Systems (f=1) nur bis $\lambda t \sim 0.7$ ($-\ln(0.5)$) eine Zuverlässigkeitssteigerung.

Der Erwartungswert der Lebensdauer:

$$E[Lges] = \int_0^{\infty} Rges(t)\,dt$$

1. Der Fehlertoleranzgrad (sonst t genannt) wird in diesem Kapitel mit f bezeichnet, um einer Verwechselung mit der Zeitvariablen t vorzubeugen.

ist mit E[Lges]=5/6λ sogar geringer als für eine Einzelkomponente. Verbesserungen lassen sich aber mit einem weiter erhöhten Fehlertoleranzgrad erreichen, sowohl hinsichtlich der Lebensdauer wie auch der Überlebenswahrscheinlichkeit bei geringer Einsatzdauer. Die erwartete Lebensdauer übertrifft bei t=2 bereits schon die der Einzelkomponente (E[Lges]=13/12λ) und kann noch weiter verbessert werden. Sie steigt jedoch für große *n* nur noch sehr langsam; der erhöhte Redundanzeinsatz wird zunehmend ineffizienter.

Das ATTEMPTO-Experimentalsystem soll für interaktive Anwendungen als Arbeitsplatzrechner dienen. Typische Joblaufzeiten sind meist kurz ($t<<1/\lambda$). Der Zuverlässigkeitsgewinn ist hier schon bei f=1 hoch, sodaß das Fehlertoleranzverfahren als gut geeignet für das Anwendungsgebiet gelten kann.

Gerade im Bereich sehr kurzer Einsatzdauern macht sich jedoch der Einfluß nichtredundanter Funktionskomponenten im System besonders bemerkbar. Die Gesamtzuverlässigkeit kann nicht größer sein, als deren minimale Einzelzuverlässigkeit. Wie in Abb. 48: mit angedeutet, beschneidet diese ($R_{bus}=e^{-0.1\lambda t}$) deshalb direkt die Überhöhung im Anfangsbereich.

Im Experimentalsystem stellt der einfach ausgelegte Kommunikationsbus einen Zuverlässigkeitsengpaß dar. Im vorliegenden Fall - VMEbus in einem System kleiner bis mittlerer Größe - darf davon ausgegangen werden, daß der Bus erheblich zuverlässiger als eine Knoten-Komponente ist. Dies ist hauptsächlich durch die vergleichsweise geringe Komplexität gerechtfertigt. Zur Bus-Komponente gehören neben den zentralen Teilkomponenten, z.B. dem Arbiter, die lokalen Busadapter, jedoch nur insoweit, als sie die Funktionsfähigkeit des Busses beeinträchtigen können. Die Gesamtkomplexität hängt also von der Knotenzahl im System ab, und nur bei ausreichend geringer Knotenzahl wird der Einfluß der Buskomponente auf die Gesamtzuverlässigkeit vernachlässigbar sein.

6.5 Leistungsverhalten im fehlertoleranten System

Dieses Kapitel stellt durch Messungen gewonnene Leistungsdaten vor und gewinnt daraus Optimierungsansätze. Allgemein stehen beim ATTEMPTO-Systemkonzept die Modularitätseigenschaften im Vordergrund, da sie den Allzweck-Charakter wesentlich begründen. Demgegenüber war hohe Systemleistung keine konzeptionelle Grundforderung. Vielfältiger Datentransport, das Abarbeiten von Kommunikationsprotokollen und häufige Prozeßwechsel lassen eine nicht unerhebliche Leistungseinbuße erwarten, die jedoch zunächst in Kauf genommen wurde. Absolute Leistungsdaten sind zudem wesentlich bestimmt durch die Leistungsfähigkeit der Standard-Grundkomponenten. Obwohl zum Beschaffungszeitpunkt Stand der Technik für ein kommerzielles Produkt dieser Art, müssen die Knotenrechner aus heutiger Sicht als veraltet gelten, sodaß die absoluten Daten unter diesem Blickwinkel relativiert werden müssen.

Zur Meßdatenerfassung stand ein Hardwaremonitor (VMEbus-Tracer) zur Verfügung, der die selektive Aufzeichnung von Ereignissen (globale Speicher-Zyklen) in Echtzeit ermöglicht. Damit ist auch eine globale Zeitreferenz gegeben. Die Aufbereitung der Meßdaten übernahm ein selbstentwickeltes Werkzeug TOAST (Tracer Output Analysis System), das das nachträgliche Herausfiltern von Meßdaten erleichtert, beim Auffinden von Ereigniszyklen behilflich ist und Graphiken in Form von Zeitdiagrammen erzeugen kann.

Abb. 49 stellt die Meßumgebung vor. Alle Komponenten können von einer zentralen Graphik-Workstation bedient werden. Für das Starten von Testprogrammen auf dem ATTEMPTO-Sy-

stem stehen die Systemkonsolen eines jeden Knotens in gesonderten Fenstern zur Verfügung. Ein weiteres Fenster emuliert das Fehlertoleranzterminal; alternativ kann (unter XWindow) die graphische ATTEMPTO-Benutzeroberfläche verwendet werden.

Die Meßdaten werden auf dem globalen Systembus erfaßt. Der VMEbus-Tracer zeichnet Buszyklen mit einer Speichertiefe von 2K x 120Bit auf und versieht diese mit einer Zeitangabe relativ zu einem Triggerereignis. Die Zeitauflösung beträgt ca. 60ns und ist damit gemessen an der maximalen CPU-Buszyklusfrequenz von 2,5MHz (68010 bei 10MHz Haupttakt) ausreichend genau. Die Meßdaten müssen explizit in dem zu testenden Programm erzeugt werden. Für die MODULA-Programmierumgebung wurde dazu eine Laufzeitbibliothek erstellt, bei der besonders auf geringen Meßoverhead geachtet wurde: die Meßdaten werden ohne Betriebssystemunterstützung direkt durch eine Wertzuweisung auf eine Speichervariable in einem globalen Speicherbereich (Trace Memory) erzeugt. Die Trace-Daten werden nach der Aufzeichnung zur weiteren Auswertung dem Werkzeug TOAST zugeführt; dieses besteht aus einem Baukastensystem von Einzelmodulen (als UNIX-Filter), die variabel hintereinandergekettet werden können und so sukzessive den Tracerdatenstrom aufbereiten. Die Daten werden eingelesen (*readtrace*), in eine lesbare Listenform übersetzt (translate), in einer oder mehreren Stufen ausgefiltert (*filter*) und letzendlich in Listenform oder als Graphik ausgedruckt (*mkpic)*. Die Analyse stützt sich auf Konfigurationsdateien, die die Kettenelemente heranziehen, um Aktoren und Ereignisse zu bezeichnen oder Filteraufgaben durchzuführen. Diese Aufgaben können in einer leicht verständlichen Syntax formuliert werden.

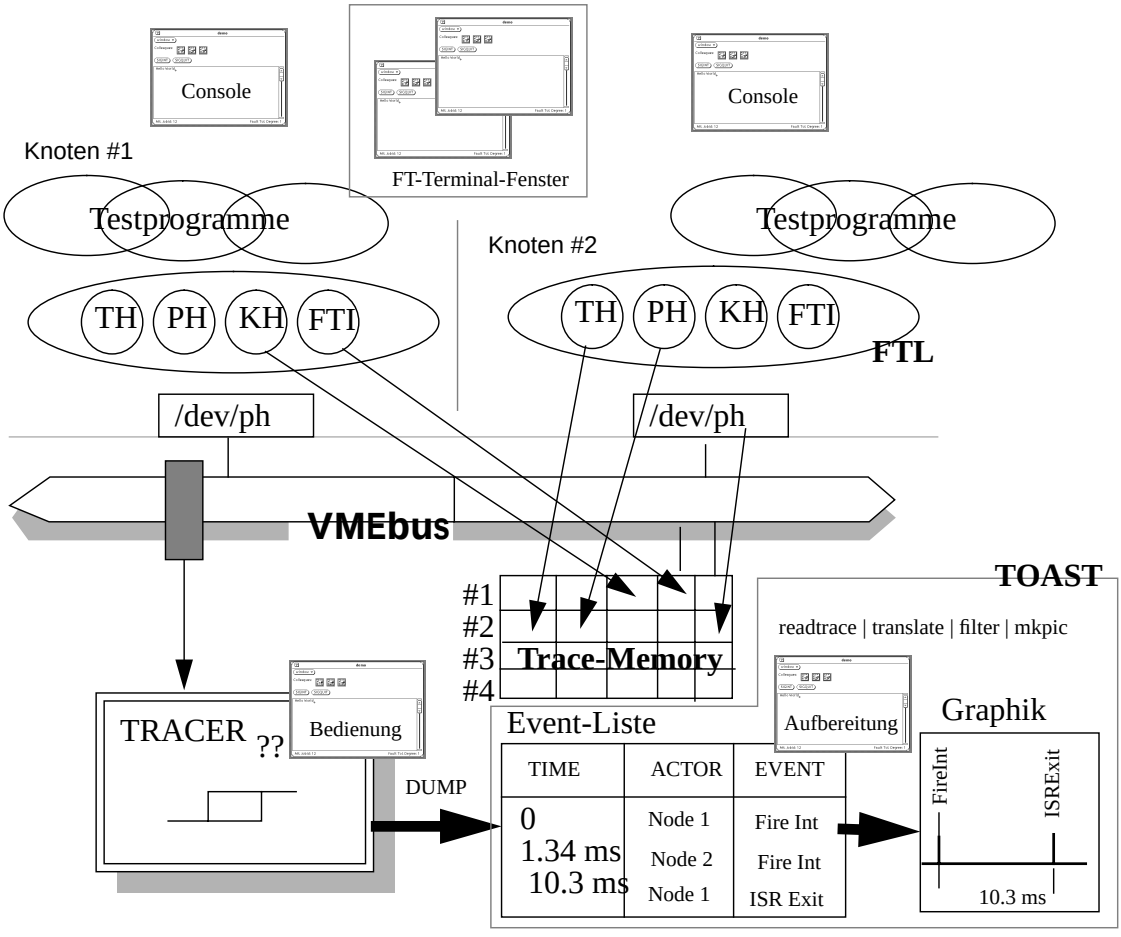


Abb. 49 Meßumgebung zur Echtzeit-Aufzeichnung und Datenauswertung mit Werkzeug TOAST

Zunächst wurden umfangreiche Messungen zur Bestandsaufnahme durchgeführt, sowohl isoliert am globalen Kommunikationssystem wie auch im laufenden Fehlertoleranzbetrieb, um lohnende Ansätze für eine Optimierung des Leistungsverhaltens zu ergründen. Grundsätzlich wird zwischen zwei Meßmethoden unterschieden: 1) die mikroskopische, die aus einem einmaligen Testlauf bestimmte Ereignisfolgen auswertet, z.B. die Laufzeit einer Nachricht aus der Differenz zwischen Sende- und Empfangszeitpunkt, und 2) die makroskopische, die mehrmalige Ereignisfolgen summarisch zusammenfaßt und durch Mittelwertbildung auf den Einzelfall schließt.

6.5.1 Leistungsmessungen und -optimierungen im Kommunikationssystem

Das Nachrichtenübermittlungsprinzip des ATTEMPTO-Broadcast-Kommunikationssystems weist besondere Eigenschaften auf (Kapitel 5.4). Der Bus wird in Rasterintervallen (Zeitschlitzen) konstanter Länge belegt, die allerdings von allen N Knoten gemeinsam für einen Rundspruch jeder an jeden benutzt werden können. Genau für diesen Fall ist die Zeitschlitzbreite vordefiniert, sodaß sich auch nur hierfür eine gute Ausnutzung des Übertragungskanals ergibt. Gleichzeitig ist damit die Maximallänge für eine Nachricht festgelegt. Für eine Punkt-zu-Punkt-Übertragung großer Datenmengen über diese Maximallänge hinaus sind deshalb keine hohen Datentransferraten zu erwarten. Für eine praktische Anwendung wird man bemüht sein, einen anwendungsangepaßten Kompromiß zwischen Gesamtdurchsatz im System und Prozeß-Prozeß-Durchsatz zu finden. Die Dimensionierung der wichtigen Systemparameter *Karenzzeit* und *Sperrzeit* muß wegen der Abhängigkeit von der Knotenzahl und den Randbedingungen des Betriebssystems für die Gewährleistung der Reihenfolgekonsistenz ohnehin die besonderen Systembedingungen berücksichtigen.

Eine Überschlagsrechnung soll zunächst die Grenzen für die Übertragungsbandbreite des Kommunikationssystems aufzeigen.

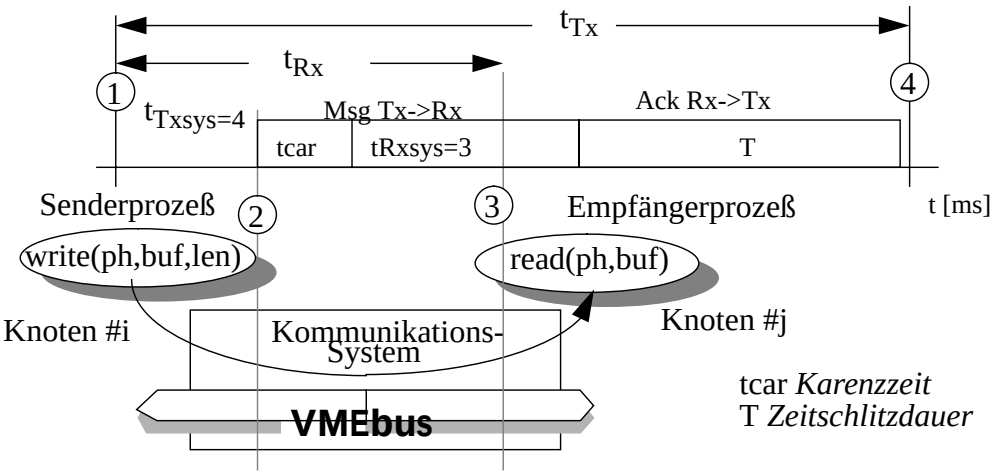


Abb. 50 Zur Bestimmung der maximalen Prozeß-Prozeß-Datenübertragungsrate

Ausgangspunkt ist eine mikroskopische Messung im oben abgebildeten Szenario zur Bestimmung der Systemcharakteristik beim Aufsetzen (t_{Txsys}) bzw. Entnehmen (t_{Rxsys}) einer Nachricht. Ein Senderprozeß übermittelt einem Empfänger auf einem anderen Knoten eine kurze (len~0) Botschaft. Außer den abgebildeten Prozessen sind keine weiteren Anwenderprozesse aktiv. Vier Meßpunkte, 3 im Anwender-, 1 im Systembereich (2), stecken folgende Beziehun-

gen ab:

$$t_{Txsys} = 4 \text{ ms}; t_{Rxsys} = 3 \text{ ms}; t_{Rx} = t_{Txsys} + t_{car} + t_{Rxsys}; t_{Tx} = 2T + t_{Txsys},$$

dabei ist t_{Rx} die eigentliche Übertragungszeit gemessen beim Empfänger, t_{Tx} die Nachrichtenübertragungszeit aus der Sicht des Senders; eine interessante Schlußfolgerung ist, daß “Nachrichten schneller empfangen werden, als sie versendet werden”. Dies liegt daran, daß der Sender auf die Bestätigung durch das Lokale Betriebssystem des Empfängers warten muß, diese aber bei der dargestellten Zeitschlitz-Dimensionierung erst nach der Ankunft der Nachricht im Prozeß erfolgt. Diese Dimensionierung läßt sich allerdings nicht grundlegend verbessern, da sie auch N^2 möglicherweise auch längere Nachrichten (LENMAX >> len) berücksichtigen muß. Der Sendecharakter wird damit synchron, obwohl der Empfängerprozeß den Fluß nicht explizit steuert[1]. Aus zwei Gründen wird der Sendeprozeß solange am Fortgang gehindert, bis die Bestätigung erfolgt ist: wegen der Übermittlungsverzögerung durch ausstehende ACKs (bis zu 3 Zeitschlitze, da die Bestätigung den direkt auf die Botschaft folgenden Zeitschlitz bei konkurrierendem Betrieb verpassen kann) und insbesondere im Fehlerfall ist ein schnelles Vollaufen des sendeseitigen Puffervorrates (OutQueue) zu befürchten, und außerdem ist eine direkte Fehlermeldung an den Sendeprozeß gewünscht.

Nimmt man einmal sehr optimistische Werte T/Tcar = 10ms/1.5ms (als zu knapp in einem anderen Kontext ermittelt) und LENMAX = 1KByte und berücksichtigt die Verwaltung der größeren Datenmenge mit $t_{blockmove}$= 3µs * LENMAX = 1.5ms (entspricht den tatsächlichen Verhältnissen) gelangt man zu

$$t_{Rx} = 4 \text{ ms} + 3 \text{ ms} + Tcar + 1.5 \text{ ms} = 10 \text{ ms},$$

und damit einer maximalen Prozeß-Prozeß-Übertragungsrate von 100KByte/s. Aus der Sicht des Senders betrachtet, leistet man nur weniger als die Hälfte, denn

$$t_{Tx} = 2T+t_{Txsys} = 24 \text{ ms}.$$

Ein praktischer Versuch (mit einem laufzeitoptimierten Kommunikationstreiber), der 100 Nachrichten à 1KByte versendet, mißt makroskopisch einen Durchsatz von $f_{P\text{-}P}$ = 45KByte/s und zeigt so, daß sich dieser Grenzwert auch bei Nachrichtenstückelung gut annähern läßt.

Nun lassen sich die Kosten für das Übermittlungsprotokoll relativieren. Der verwendete Prozessor (genauso der unterstützende DMA-Controller) kann Daten mit bis zu f_{CPU} = 5MByte/s transportieren. Da der VMEbus-Arbiter ein langsamer Single-Level-Arbiter ist (*Daisy Chain*) und ein gewisser Verwaltungsaufwand durch den Speicherblocktransfer berücksichtigt werden muß, reduziert sich dies auf nur noch f_{BUS} = 660KByte/s bei Zugriffen über den Bus in die empfangsseitigen Mailboxen, sodaß als Überschlag

$$f_{CPU} : f_{BUS} : f_{P\text{-}P} = 100 : 10 : 1$$

gelten kann. Die tatsächliche Busbelegung beträgt 7% (45/660).

1. Die Definition für synchrones Senden in Kapitel 5.4.7 hat vorausgesetzt, daß Bestätigen durch das System anstelle des Empfangsprozesses immer schnell vonstatten geht.

Die Ausgangssituation im Kommunikationssystem für den Fehlertoleranzbetrieb hat einen hohen Sicherheitsabstand einkalkuliert und weist T/Tcar = 26 ms/7 ms auf. Damit reduziert sich die maximale Übertragungsrate auf ca. $f_{P\text{-}P}$ = 20KByte/s.

Diese errechneten Werte gelten nur für die Punkt-zu-Punkt-Übertragung eines einzelnen Pakets (*Burst*-Übertragungsrate). Da dasselbe Transferintervall von allen N Knoten für einen Rundspruch verwendet werden kann, ist die theoretische Bustransferkapazität des Kommunikationsprotokolls mit $f_{Komm} = N^2 f_{P\text{-}P}$ aber wesentlich höher. Dabei ist konstanter Grundoverhead (t_{Txsys}, t_{Rxsys}) auch bei Rundspruch und ausreichend breite Dimensionierung des Zeitschlitzes vorausgesetzt. Die praktischen Verhältnisse liegen jedoch anders. Prinzipiell ist auch die Stückelung bei großer zu transportierender Datenmenge noch nicht berücksichtigt. Die Bilder Abb. 51 bis Abb. 53 zeigen die Abhängigkeit der Übertragungsgeschwindigkeit bei Rundspruch von den Parametern Nachrichtenanzahl, Paketgröße und Zahl der Rundspruch-Empfänger. Für die absoluten Zeitwerte der Ordinate ist die besondere Testumgebung wesentlich verantwortlich. Interessant sind daher nur Relationen.

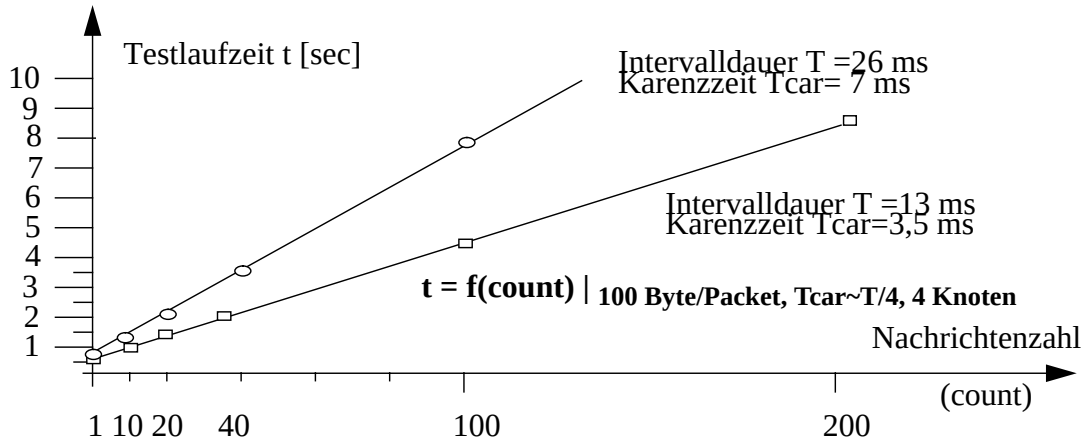


Abb. 51 Prozeßbezogene Broadcast-Übertragungszeit abhängig von der Nachrichtenzahl

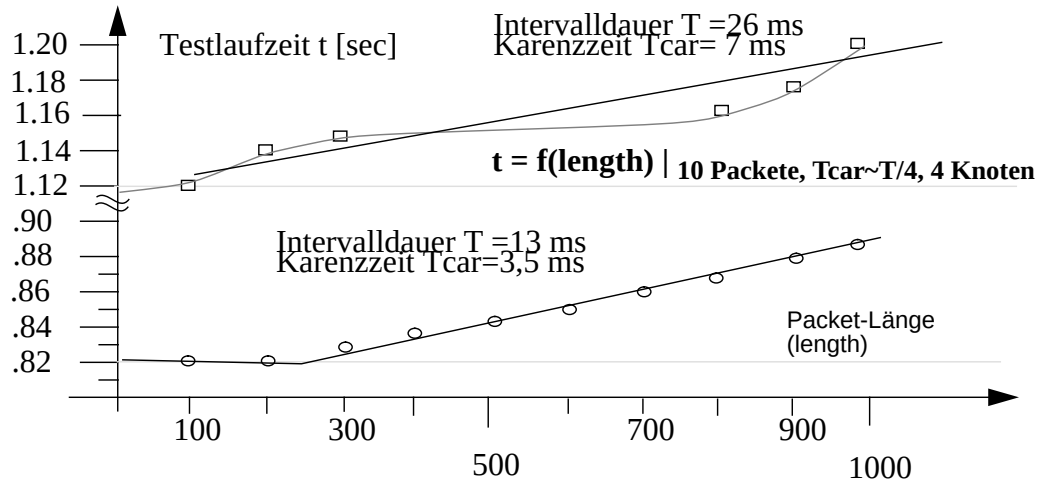


Abb. 52 Prozeßbezogene Broadcast-Übertragungszeit abhängig von der Nachrichtenpacketlänge

Aus Abb. 51 entnimmt man, daß die durchschnittliche Übertragungsgeschwindigkeit eines mit 100 Byte relativ kurzen Rundspruchs im Falle einer Zeitschlitzbreite T = 26ms ca. 70ms pro

Rundspruchpaket und bei halb so großen Zeitschlitzen ca. 40ms/Paket beträgt. Die Abweichungen von der idealen Gesetzmäßigkeit $t_{Tx} = 2T+t_{Txsys}$ erklärt sich dadurch, daß nicht alle Bestätigungen von den Rundspruchempfängern im selben Zeitschlitz liegen; gelegentlich, beim kürzeren Intervall durch die komprimierten Verhältnisse erkennbar häufiger, ist auch $t_{Tx} = 3T+t_{Txsys}$ möglich.

Die Verhältnisse ändern sich bei längeren Datenpaketen nicht wesentlich; die Abhängigkeit von der Länge beträgt etwa 80 µs/Byte mit einer für kleinere Zeitschlitze besonders ausgeprägten Schwellwertcharakteristik (Abb. 52). Die Nichtlinearität bei T=26ms resultiert aus der niedrigen Nachrichtenzahl von 10.

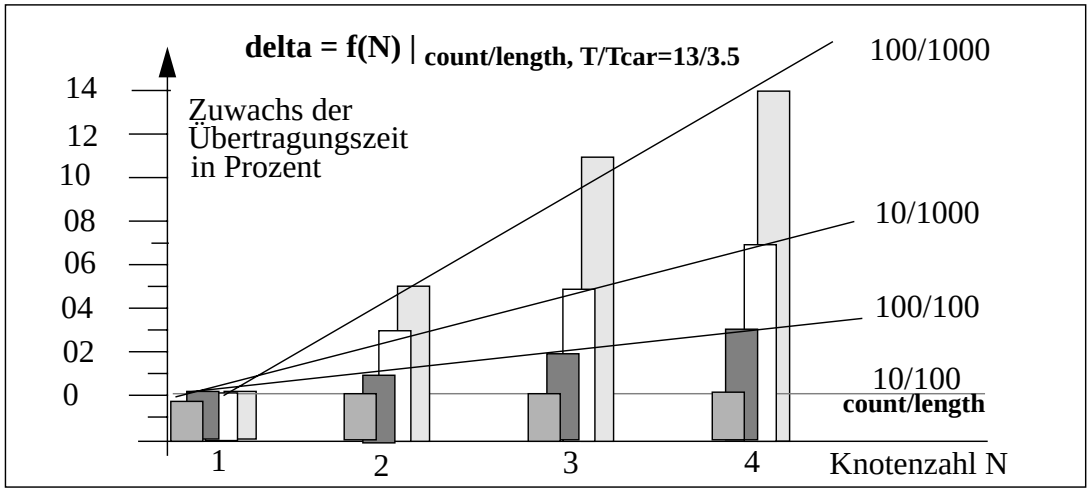


Abb. 53 Steigerung der Übertragungszeit bei wachsender Knotenzahl im Rundspruch; Parameter: Nachrichtenzahl und Paketlänge

Die Nachrichtenportionierung spielt auch eine wesentliche Rolle bei der Abhängigkeit von der Knotenzahl. Aus Abb. 53 gewinnt man:

- bei wenigen kurzen Nachrichten (10 à 100 Byte) ist innerhalb der Meßauflösung keine Abhängigkeit zu erkennen,
- viele kurze Nachrichten (100/100) weisen eine Zunahme von 1% pro Knoten auf (0.4ms/Knoten*count),
- bei langen Paketen ist die Zunahme stärker: 2% pro Knoten bei wenigen bzw. 4% pro Knoten bei vielen (2ms/Knoten*count).

Die Interprozeßkommunikation in ATTEMPTO findet noch auf weiteren Kanälen statt. Neben der soeben bewerteten Interprozessor-Kommunikation gibt es noch knotenlokale Kommunikation zwischen Schwergewichtsprozessen über UNIX-Pipes und zwischen Leichtgewichtsprozessen über Mailboxen. Die realen Kommunikationsverzögerungen sind von den Lastverhältnissen im Betrieb abhängig. Abb. 54 zeigt die minimalen Transferverzögerungen, die sich dann ergeben, wenn die kommunizierenden Schwergewichtsprozesse laufbereit ohne Konkurrenz durch andere im Speicher liegen. Handlerprozesse erhalten ihre Daten von globalen Schnittstellen, hier dargestellt ist der Empfang einer Interprozessor-Botschaft (t_{Rx} = 15 - 40 ms je nach Verkehr auf dem Bus). Die Kosten für eine Pipe-Kommunikation (8ms) sind betriebssystembestimmt. Die Interprozeßkommunikationskosten zwischen Leichtgewichtsprozessen betragen etwa die Hälfte, sind aber mit 4ms immer noch hoch (bedingt durch zeitaufwendiges Spei-

cherallokieren und Datenkopieren). Insbesondere, wenn Fehlertoleranz-Clerks miteinander kommunizieren, wozu sie das Post-Office benutzen müssen, ist der Zeitaufwand genauso groß wie zwischen UNIX-Prozessen. Den höchsten Aufwand kostet das Verschicken von globalen Nachrichten.

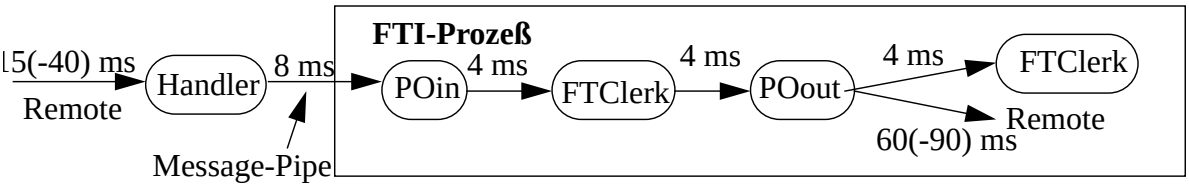


Abb. 54 Optimale Nachrichtentransferzeiten der Interprozeß-Kommunikation (Tcar/T=26ms/7ms)

Während des Versendens von Nachrichten ist innerhalb des FTI-Prozesses kein Empfangen möglich (zwischen Clerks zeigt die obige Darstellung die Kommunikation als kombinierte Sende/Empfangs-Aktion). Dies bedeutet, daß sich während des Versendens einer Interprozessor-Nachricht in der Message-Pipe Nachrichten der Handlerprozesse aufsammeln können. Tatsächlich wirkt sich die Sendeverzögerung sehr nachteilig auf die Bedienrate der Fehlertoleranzinstanz für ankommende Ereignisse aus. Abb. 55 zeigt die Verteilung der Wartezeiten der in der Message-Pipe gepufferten Nachrichten anhand eines repräsentativen Jobs. Die mittlere Wartezeit ist mit fast 90 ms sehr hoch; dabei kann die Pufferhöhe bis zu 8 Nachrichten betragen. Da sehr häufig am Ende eines Bearbeitungszyklus für eine eingehende Botschaft das Aussenden einer Nachricht an Jobkollegen steht, also das Verhältnis eingehender zu ausgehender Nachrichten ziemlich ausgewogen ist, führt dies zu gelegentlich sehr langen Wartezeiten, die sich direkt im Antwortzeitverhalten des Systems bemerkbar machen. Der Betrieb ist blockiert, obwohl außer der Entgegennahme weiterer externer Ereignisse keinerlei Bearbeitung während des Wartens auf das Ende des Sendens stattfindet. Hier ist eine Optimierung unbedingt angezeigt

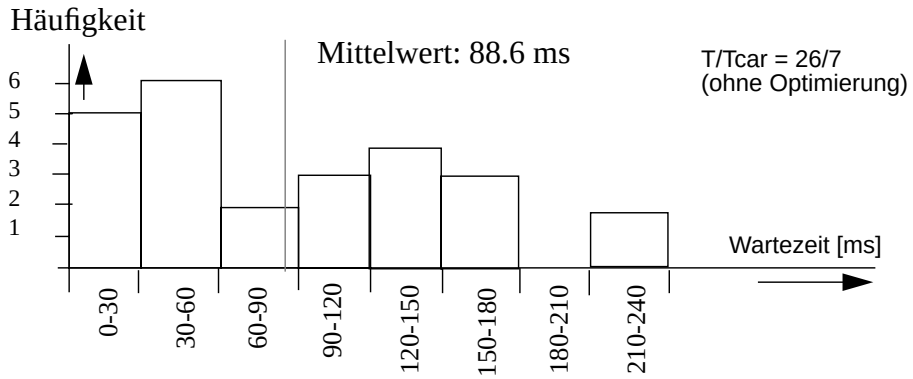


Abb. 55 Verteilung der Pufferwartezeiten in der Message-Pipe zum FTI-Prozeß während eines Jobs

Sie wurde in einem verbesserten Kommunikationstreiber vorgenommen. Bei eingehender Betrachtung des Sendeablaufs erkennt man, daß die Sendezeit t_{Tx} eigentlich schon vor Ablauf des gesamten ACK-Zeitschlitzes beendet werden könnte, nämlich dann, wenn die letzte Bestätigung korrekt eingetroffen ist. Diese liegt bereits nach der Karenzzeit vor. Da die Bearbeitung sehr kurz ist, kann schon aus der Portinterruptroutine heraus (siehe dazu Kapitel 5.4.7) der Sendeprozeß aufgeweckt werden, ohne den Ablauf des Cleartimers abzuwarten. Die Verbesserung beträgt annähernd T_{clear}. Dieser Wert ist wegen Tclear=3/4 T nicht unbeträchtlich. Diese Optimierung bringt insgesamt eineVerbesserung um etwa ein Drittel in der Ausgaberate, wie in einem späteren Zusammenhang die Abb. 59 illustrieren wird.

Ein weiterer Optimierungsbedarf wird aus Abb. 54 noch deutlich: die Einsparung des POout-Clerks kann für die interne Kommunikation der Fehlertoleranz-Clerks die Nachrichtentransferzeit etwa halbieren. Dies geht zwar auf Kosten der funktionalen Modularisierung; die Vorteile der zentralen Adreßindirektion für die Konfigurierbarkeit können jedoch durch prozedurale Abkapselung in einem Bibliotheksmodul (der 2. Ebene im ATTEMPTO-Schichtenmodell) erhalten werden. Diese Modifikation bringt ebenfalls deutlichen Gewinn.

Es wurde bereits deutlich, daß ein sorgfältiges Dimensionieren der Zeitschlitzverhältnisse erheblichen Gewinn erwarten läßt. Die Anwendung und die Eigenschaften des lokalen Wirtsbetriebssystems und der Hardware geben hier Randbedingungen vor. Grundsätzlich ist von Vorteil, daß die Nachrichtenpakete in der Länge sehr begrenzt und die Maximallänge bekannt ist, denn im Fehlertoleranzbetrieb kommunizieren nur Systemprozesse über Prozessorgrenzen hinweg. In Kapitel 5.4.6 wurde abgeleitet, daß $T = T_{car} + T_{clear} = 2\ T_{car} + T_{ISRmax}$ + delta gilt. Die Einbeziehung der Betriebssystemeinflüsse auf die Karenzzeit (Laufzeit höherpriorer Ausnahmebehandlungen) und den Sicherheitszuschlag delta (gleichpriore Ausnahmen und höhere sowie Interruptsperren) macht jedoch Schwierigkeiten bei der Dimensionierung, da wegen der fehlenden Kenntnis des Kern-Quellcodes insbesondere die Evaluierung der Dauer der kritischen Abschnitte schwerfällt. Dazu wurden längere Beobachtungen durchgeführt, die zeigen, daß die Grenze von 1,5 ms offenbar ausreicht, dabei ist schon ein Zuschlag berücksichtigt. Die Priorität des Portinterrupts liegt glücklicherweise so hoch, daß nur ganz wenige Ausnahmen zu berücksichtigen sind, zumal die meisten Ausnahmebehandlungen zu einem Knotenstillstand führen. Die restlichen Zeitanteile liegen im eigenen Zuständigkeitsbereich des Kommunikationstreibers, können also direkt gemessen werden. Eine optimale Dimensionierung wurde bei ähnlicher Vorgehensweise in [Grill93] berechnet, und resultiert in einer ausreichend sicheren Festlegung auf T=13 ms, T_{car}=3,5 ms; diese Werte wurden auch in den bereits vorgestellten Messungen benutzt. Die erzielten Verbesserungen für den Fehlertoleranzbetrieb sind ebenfalls aus Abb. 59 ersichtlich.

6.5.2 Charakteristik der Jobvergabe

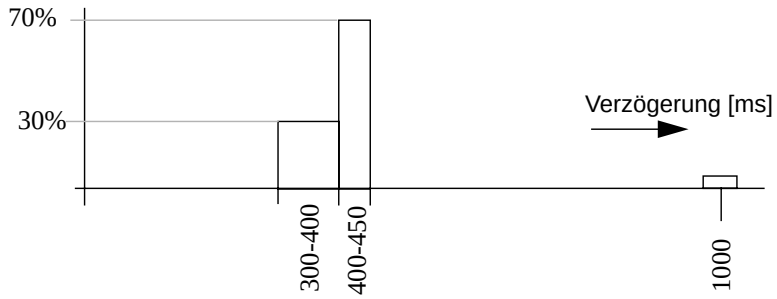


Abb. 56 Differenz im Startzeitpunkt von Jobreplikaten

Grundsätzlich ist die Verzögerung des Jobstarts durch das Verfahren der Selbstanziehung kein wesentliches Thema für die Leistungsbewertung, da die Jobvergabe in der ohnehin längeren Phase des Startens von Jobs untergeht, die durch das Laden des Programmcodes und das Installieren als Verwaltungsobjekt des Betriebssystems dominiert wird. Selbst erhebliche Verlängerungen würden durch den Benutzer geduldet, da sie nur einmalig auftreten. Dennoch soll die Abb. 56 einen ungefähren Überblick über die Abweichungen im Startzeitpunkt von Kollegen-Jobs unter angeglichenen lokalen Betriebsverhältnissen liefern. Ein Schwerpunkt

scheint bei etwa 400 ms zu liegen; davon fordert die Entgegennahme der Jobnachricht im TH-Prozeß einen großen Anteil. Es werden jedoch auch wesentlich längere Unterschiede beobachtet. Diese Effekte können nicht zweifelsfrei begründet werden, da der Codeumfang bzw. insgesamt der Speicherbedarf der Fehlertoleranzsoftware an Grenzen stößt, die gelegentliches Auslagen von Teilprozessen erforderlich machen. Das UNIX-Basissystem verwendet nur Gesamtprozeßauslagerung (*swapping*).

6.5.3 Maskierungsleistung

Softwareimplementierte Fehlermaskierung kostet im Gegensatz zu hardwareimplementierter immer Systemleistung. Der entstehende Overhead ist - abgesehen von Job-Verwaltungsaufgaben z.B. zur Gewährleistung konsistenter Eingabedaten für die Jobreplikate - immer auf eine Ausgabeoperation zu beziehen und zwar unabhängig von der damit an die Systemumwelt transportierten Datenmenge. In einem lose gekoppelten System verteilter Maskierungsinstanzen ist der Grundaufwand für den Maskierungsmechanismus relativ hoch, da in der Regel eine Vielzahl von Nachrichten ausgetauscht werden müssen, um die Korrektheit des Jobergebnisses abzusichern und die Ausgabe selbst vorzunehmen. Die Ausgabeaktivität eines Jobs kann bei entsprechender Prozessorleistung sehr hoch sein; ein Engpaß wird i.a. erst an der Schnittstelle zur Ausgabeperipherie auftreten. So ist auch in einem (nicht fehlertoleranten) ATTEMPTO-Knotenrechner die Ausgabefrequenz durch die Baudrate der seriellen Leitung zum Terminal begrenzt, bei einer üblichen Baudrate von 9600 Baud also auf etwa 1000 Zeichen/s. Nur bei ausreichend geringem Overhead der Fehlerunterdrückung im fehlertoleranten System wird an der Schnittstelle keine Leistungseinbuße meßbar sein. Die tatsächlichen Verhältnisse im Experimentalsystem liegen jedoch viel schlechter. Die Anwendung prägt allerdings das Erscheinungsbild. Während bei größeren Datenmengen pro Ausgabeoperation eine verringerte Ausgabeleistung kaum bemerkt wird, wird dies bei schneller zeichenweiser Ausgabe durch den Benutzer als sehr störend empfunden.

Ein handliches Maß zur Beschreibung der Systemleistung des fehlertoleranten Systems ist die ***Maskierungsrate***. Sie kennzeichnet die Anzahl der Ausgabeoperationen pro Zeiteinheit im Fehlertoleranzbetrieb. Sie läßt sich leicht mit einem Testprogram messen, das nur aus einer Ansammlung von *write*-Systemaufrufen besteht. Eine Bestandsaufnahme am ursprünglichen System ergab 1,6 Maskierungen/s bei einem Fehlertoleranzgrad von t=1. Durch die Optimierungen im Kommunikationssystem konnte dies bereits auf 2,6 Maskierungen/s verbessert werden. Der Overhead ist aber immer noch ganz erheblich.

Die Maskierungsrate wurde als Mittelwert bei 108 Ausgaben gemessen (plus einer nicht zu maskierenden Ausgabe der Benutzer-Shell auf jedem Knoten). Jede Ausgabeoperation trägt nur ein Zeichen. Der Ausgaberhythmus war dabei regelmäßig, allerdings wurden periodisch maximal 2 Zeichen aufgesammelt und dann geschlossen vom jeweiligen Ressourcen-Manager ausgegeben. Interessant ist auch, daß die Ausgabe nur von zwei bestimmten Knoten vorgenommen wurde, obwohl der dritte sich (sogar am häufigsten) um diese mitbeworben hat.

Dies ist aus der Abb. 57 zu erkennen, die das gesamte Meldungsaufkommen im Postoffice auflistet. Man erkennt eine Unsymmetrie im Diagnosealgorithmus anhand der unterschiedlichen Anzahl der Nachrichten vom Typ *Signature, DemandKey* und *Key,* die wie im Kapitel 5.6.4 beschrieben verwendet werden. Er ähnelt damit dem optimierten Algorithmus in Kapitel 4.2, enthält aber noch zusätzliche Elemente. Nur vom mittleren Knoten fordern alle einen Schlüssel an. Die Schlüsselanforderungen erlauben es, knotenprivate und damit verwechselungsfreie

Schlüssel zu verwenden.

Die Signaturen werden per Rundspruch an alle Kollegen verschickt, und nicht nur an die Nachbarn im Diagnosegraphen. Damit ist jederzeit eine vollständige Diagnose aller Kollegen möglich, die Kosten bleiben jedoch in Grenzen, da die Transferzeit für Multicastnachrichten annähernd unabhängig von der Knotenzahl ist. Auch der Empfang von überzähligen Signatur-Nachrichten ist relativ billig, da keine zusätzlichen Ausgabenachrichten kausal davon abhängen.

	Knoten #1		Knoten #2		Knoten #3	
MsgType	Received	Sent	Recvd	Sent	Recvd	Sent
NewJob	1	0	1	0	1	0
StartRequest	4	1	4	1	4	1
Signature	216	108	216	108	216	108
DemandKey	0	108	216	108	108	108
Key	107	0	108	213	106	108
OutptRequest	257	105	257	106	257	46
OutStatus	0	108	0	108	0	108
UserJobStart	0	1	0	1	0	1
UserJobWrite	109	0	109	0	109	0
UserJobExit	3	1	3	1	3	1
ReturnPid	1	0	1	0	1	0
clearDIB	0	1	0	1	0	1
clearDOB	0	1	0	1	0	1
clearSAB	0	1	0	1	0	1
ResRequest	114	66	114	3	114	45
ResRelease	111	64	111	2	111	45
OutputData	0	64	0	2	0	45
watchData	0	45	0	107	0	64
Summe	923	674	1140	763	1030	683
Gesamt-Summe Sent:		2120				

Abb. 57 Job-Meldungsaufkommen im PostOffice bei der Fehlermaskierung von 108 Ausgabeportionen (Fehlertoleranzgrad t=1)

Bei genauer Betrachtung zeigt sich, daß die Diagnosebeziehungen etwas unglücklich gewählt sind: durch die unsymmetrische Belastung werden die Knoten #2 und #3 benachteiligt; der langsamste bestimmt aber die Maskierungsrate, da die Runde zum Austausch der Signaturen synchronisiert ist. Erst werden alle Signaturen eingesammelt, bevor weiter verfahren wird. Der langsamste ist offensichtlich der mittlere Knoten, denn er hat das höchste Meldungsaufkommen zu bearbeiten und gewinnt deshalb auch nie den Wettlauf um die Ausgabe. Ähnliches gilt auch für den Knoten #3, der ebenfalls weniger erfolgreich ist als der Knoten #1. Das Beispiel des unvorteilhaften Diagnosealgorithmus ist absichtlich gewählt, um im Vergleich mit anderen bestimmte Einflußmechanismen auf das Leitungsverhalten zu verdeutlichen. Man sollte natürlich den Diagnosalgorithmus für diesen Fehlertoleranzgrad modifizieren. Daß jedoch der in Kapitel 4.2 vorgestellte optimierte Diagnosealgorithmus der beste ist, muß bezweifelt werden, da er zwar geringes Gesamt-Meldungsaufkommen realisiert, dabei aber unsymmetrisch bleibt. Ein Vergleich mit einem symmetrischen Diagnosealgorithmus bei höherem Fehlertoleranzgrad t=2 zeigt das auf den ersten Blick überraschende Ergebnis, daß die Maskierungsrate unverändert ist. Das im Verhältnis 4:3 angestiegene Gesamt-Meldungsaufkommen hat unter den vorliegenden Voraussetzungen nicht dominant die Maskierungsleistung beeinflußt. Dafür sind nur die kritischen Sendebotschaften gerechnet, für eingehende Botschaften ist der zahlenmäßige Unterschied sogar noch größer.

	Knoten #1		Knoten #2		Knoten #3		Knoten #4	
MsgType	Received	Sent	Recvd	Sent	Recvd	Sent	Recvd	Sent
NewJob	1	0	1	0	1	0	1	0
StartRequest	4	1	4	1	4	1	4	1
Signature	324	108	324	108	324	108	324	108
DemandKey	108	108	108	108	108	108	108	108
Key	103	108	106	103	103	106	108	103
OutptRequest	321	82	321	80	321	81	321	78
OutStatus	0	108	0	108	0	108	0	108
UserJobStart	0	1	0	1	0	1	0	1
UserJobWrite	109	0	109	0	109	0	109	0
UserJobExit	4	1	4	1	4	1	4	1
ReturnPid	1	0	1	0	1	0	1	0
clearDIB	0	1	0	1	0	1	0	1
clearDOB	0	1	0	1	0	1	0	1
clearSAB	0	1	0	1	0	1	0	1
ResRequest	118	29	118	26	118	31	118	31
ResRelease	112	29	112	22	112	29	112	28
OutputData	0	29	0	22	0	29	0	28
watchData	0	83	0	90	0	83	0	84
Summe	1205	687	1208	675	1205	685	1210	679
Gesamt-Summe Sent:		2762						

Abb. 58 Job-Meldungsaufkommen im PostOffice bei der Fehlermaskierung von 108 Ausgabeportionen (Fehlertoleranzgrad t=2)

Es scheint also wichtiger zu sein, ein knotenbezogenes Minimum des Botschaftsaufkommens zu finden. Eine gleichmäßigere Verteilung kann trotz etwa angestiegener Gesamtzahl die Parallelität im System besser auszunutzen.

Ein einfaches Aufsummieren der Nachrichten läßt die Kausalitäten der Interaktionen außer Betracht. Diese lassen sich leicht charakterisieren anhand von funktionalen Phasenzugehörigkeiten. Solche Phasen sind z.B. die Runden des Diagnosealgorithmus (Signaturverschicken, Key erhalten). Sie finden sich auch wieder in der Beschreibung des Maskierungsmechanismus im Kapitel 5.6.4 als Signaturaustausch und Ergebnisbestätigung und werden dort fortgesetzt durch die Ermittlung des Ausgabeberechtigten und der Phase des Ressourcen-Zuteilens. Ob der Selbstattraktionsmechanismus der beiden letztgenannten Phasen für das Leistungsvermögen eine günstige Wahl ist, könnte auch in Frage gestellt werden. Da er jedoch sehr einfach ist und gute Robustheitseigenschaften aufweist, wird darauf verzichtet, nach einer Alternative zu suchen. Generell sollte man versuchen, die Phasen in der Anzahl so gering wie nötig und so kurz wie möglich zu halten. Unter diesem Gesichtspunkt muß die Existenz der *DemandKey*-Nachricht unter Kosten/Nutzen-Betrachtungen kritisch hinterfragt werden. Der optimierte Diagnosealgorithmus aus Kapitel 4.2 ist hier günstiger. Noch interessanter ist allerdings in dieser Beziehung ein Vergleich mit einem Paarentscheid (2-von-N), der ein einfaches einphasiges Maskierungsprotokoll (Verteilungsprotokoll) erlaubt. Der Diagnosealgorithmus wurde in diesem Sinne modifiziert. Das Ergebnis wies mit einer Verbesserung um 50% den insgesamt größten prozentualen Gewinn aus im Vergleich mit den anderen durchgeführten Optimierungsansätzen; die Maskierungsrate steigt auf 3,9 Maskierungen/s. Das einfache Protokoll kann besonders von den geringen Broadcast-Übertragungskosten profitieren. Ein klassischer Mehrheitsentscheid hätte denselben Vorteil, würde aber ein anderes Redundanzkonzept verlangen.

Darüberhinaus wurde noch versucht, den Einfluß weiterer Phasen im funktionalen Ablauf auf das Leistungsvermögen einzuschätzen. Für die Jobvergabe ist dies bereits vorgestellt worden. Das Antwortzeitverhalten auf Programm-Eingaben wäre ebenfalls interessant, ist jedoch für die Leistungsfähigkeit von geringer Bedeutung. Vielversprechend ist dagegen eine Optimierung der Ressourcenverwaltung, da sie das Ausgabeverhalten entscheidend mitbestimmt, worauf die sehr begrenzte maximale Ausgabefrequenz eines nichtfehlertoleranten Test-Jobs hinweist. Ein Ansatz wurde in [Grill92] untersucht. Für unseren Zweck soll es genügen, den Einfluß des bestehenden Ressource-Locking-Mechanismus zu quantifizieren. Dazu wird er ersatzlos gestrichen. Wegen der ausgeprägten Synchronisation beim Signaturenaustausch tritt bei kleinen Ausgabeportionen keine Durchmischung der Ausgaben auf; Jobparallelität ist aber natürlich nicht möglich. Die Verbesserung ist beim nichtfehlertoleranten Betrieb am augenfälligsten; hier steigt die maximale Ausgabefrequenz auf fast das Doppelte an. Der absolute Gewinn beträgt unabhängig vom Fehlertoleranzgrad 40ms pro Ausgabeoperation. Die Überwachung des fehlertolerant ablaufenden Benutzerjobs kostet als einzig verbleibende Phase also ebensoviel.

Die erreichten Verbesserungen durch die bisher beschriebenen Optimierungsansätze stellt Abb. 59 noch einmal zusammenfassend in ihrer Wirkung auf die Ausgabe- bzw. Maskierungsrate gegenüber. Bei der Darstellung ist zu beachten, daß sich die jeweiligen Zuwächse akkumulieren, d.h. eine Maßnahme baut auf den Erfolgen der vorherigen (weiter links im Balkenpaket stehenden) auf. Erst mit den Maßnahmen zur prozeduralen Verbesserung der Ausgabebehandlung bzw. der Maskierungsmechanismen tritt eine Abhängigkeit vom Fehlertoleranzgrad auf. Da für diese Fälle das Maskierungsverfahren bei allen Fehlertoleranzgraden das gleiche ist (entsprechend einem vollvermaschten Diagnosegraphen), wird hier eine direkte Abhängigkeit vom Meldungsaufkommen vermutet. Die Nachrichtenzahl für Fehlertoleranzgrade 0<=t<=2 vergleicht Abb. 60. Nimmt man als Ausgangswerte für die Bewertung das Gesamtbotschaftsaufkommen und vergleicht das Vermehrungsverhältnis mit der Verringerung der Maskierungsrate, wie dies Abb. 61 vornimmt, wird die Vermutung in etwa bestätigt.

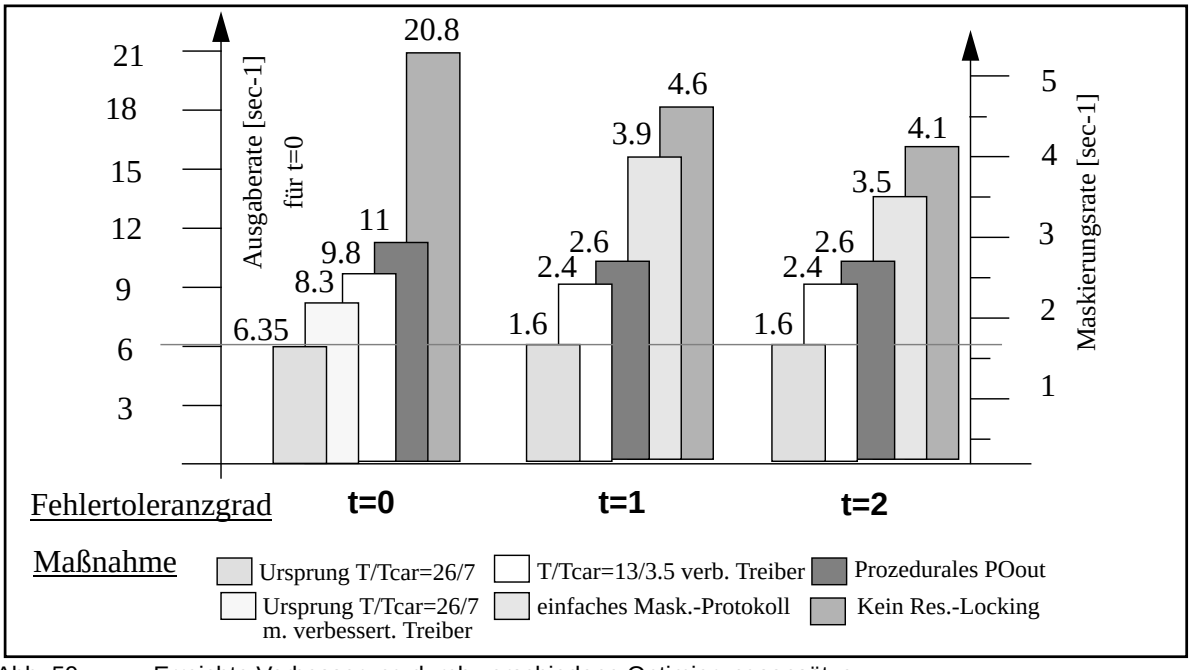


Abb. 59 Erreichte Verbesserung durch verschiedene Optimierungsansätze

MsgType	t=0	t=1	t=2	
NewJob	1	1	1	
StartRequest	4/1	4/1	4/1	
Signature	-	216/216	324/216	
OutptRequest	-	109/109	432/109	**global**
UserJobWrite	109	109	109	
UserJobExit	1/1	3/1	4/1	
ReturnPID	1	1	1	
Summe global	116/2	446/327	875/327	
OutputData	109	45	64	
watchData	-	64	45	**lokal**
Signature	-	108	108	
ClearDIB/DOB/SAB	3	3		
OutStatus	-	108	108	
UserJobStart	1	1		
Summe lokal	113	329	329	
Summe global+lokal:	231	1102	1531	

Abb. 60 Globales und lokales Meldungsaufkommen eines Knotens bei unterschiedlichem Fehlertoleranzgrad (ohne Ressource-Locking und bei vereinfachtem Maskierungsprotokoll); 108 Ausgaben

Fehlertoleranzgrad t	0		1		2
Ausgaberate [sec^{-1}] (Maskierungsrate)	20.8		4.6		4.1
Verringerungsverh.	1	:	4.5	:	5.1
Nachrichtenzahl	231		1102		1531
Vermehrungverhältnis	1	:	4.77	:	6.63

Abb. 61 Fehlertoleranzgradabhängigkeit im Nachrichtenaufkommen und der Ausgabe-/Maskierungsrate bei reduziertem Meldungsaufkommen (ohne Ressource-Locking, vereinfachtes Mask.-Protokoll)

6.5.4 Weitere Optimierungsmöglichkeiten

Die bisher vorgestellten Optimierungsansätze sind die als besonders wirkungsvoll eingeschätzten. Darüberhinaus bestehen noch weitere Möglichkeiten, deren Realisierung jedoch z.T. im Widerspruch zu den vorgegebenen Konzeptprämissen stehen. Einige sollen zum Abschluß nur aufgelistet werden:

- Reduktion gesteigerter Systemaktivitäten (Anzahl von zusätzlichen Systemcalls)
- prozed. Verbesserung im Kommunikationssystem: negative statt positive Bestätigung

- Minimierung von Zwischenpuffer-Instanzen im Kommunikationssystem
- Optimierung im Botschaftsformat
- Vermeidung von Kopieraktionen zwischen Mailboxen
- Wegeoptimierung für Botschaften bzw. Verringerung des Botschaftsaufkommens durch Zusammenlegung von Clerks
- Verzicht auf Leichtgewichtsprozesse
- Speicher- statt Nachrichtenkopplung
- Direktzugriffe auf Daten fremder Clerks ohne abstrakte Datentypen = Zentralisierung der Datenlagerung bei dezentraler Verwaltung
- Zentralisierung von Systemfunktionen (z.B. zentraler Resource-Manager) unter Verletzung ursprünglicher Design-Prinzipien
- Verändertes Synchronisations-Schema (geregelte Synchronisation mit Puffertiefe > 1)
- Gezielte Laufzeitverbesserungen für kritische (= häufig benutzte und/oder langlaufende) Prozeduren.
- Komplette oder Teilverlagerung der Fehlertoleranzmechanismen in den lokalen Betriebssystemkern.

Ein wichtiger Aspekt verlangt in Zukunft noch nach ausführlicher Bewertung und Optimierung. Er betrifft die Leistungsdegradierung im Fehlerfall. Zur Zeit zeigt sich wegen zu langer Zeitschranken ein unbefriedigendes Übergangsverhalten (Stolperbetrieb).

7. Zusammenfassung

ATTEMPTO ist eine exemplarische Realisierung einer fehlertoleranten UNIX-Workstation auf der Basis eines lose gekoppelten verteilten Mehrrechnersystems. Der Verteilungscharakter bezieht sich auf die Parallelverarbeitung von Jobs, automatische Replikatverwaltung, Kontrolle globaler Systemressourcen und die Funktion der Fehlermaskierung.

Die grundlegenden **Systemanforderungen und -merkmale** können zusammengefaßt werden zu:

- modulare Software-implementierte Fehlertoleranz als Betriebssystemerweiterung
- leichte Portabilität auf unterschiedliche Basisbetriebssysteme
- preisgünstige Systemrealisierung durch Standardkomponenten
- Anwendungs-Transparenz der Fehlertoleranzmechanismen
- Fehlerunterdrückung durch unmittelbare verteilte Systemdiagnose
- weitreichendes Fehlermodell
- variabler Redundanzeinsatz entsprechend der Benutzeranforderung an die Zuverlässigkeit
- leichte Konfigurierbarkeit und Testbarkeit.

Die erwähnten Anforderungen werden durch die Softwarearchitektur besonders gut unterstützt. Eine Standard-Hardware kann als gut geeignete Basis gelten. Die zusätzlichen Betriebssystemfunktionen sind vom Basisbetriebssystem isoliert gehalten in einer eigenen Schicht (Fehlertoleranzschicht FTL), mit einem sehr begrenzten Zugang zum lokalen Standardbetriebssystem (UNIX). Die lokalen Fehlertoleranzinstanzen (FTI) dieser Schicht bestehen aus einzelnen Schwergewichtsprozesse (UNIX-Prozesse), die ihrerseits in mehrere Leichtgewichtsprozesse (Clerks) aufgeteilt sind. All diese Prozesse kapseln verschiedene abgeschlossene Funktionen in Module im Sinne von abstrakten Datentypen und bauen auf hierarchisch gegliederten Dienstleistungsmodulen auf. Alle Prozesse kommunizieren ausschließlich durch Botschaftsaustausch in einem einheitlichen Nachrichtentransportsystem. Die Verteilung von Botschaften übernimmt in jeder lokalen FTI ein zentraler Prozeß (Post Office), der sich deshalb besonders gut eignet, das Modul-Ein/Ausgabe-Verhalten sowie die Modul-Interaktionen zu überwachen. Hierdurch wird die Testbarkeit sehr begünstigt.

Zur Konzeptvalidierung wurden zunächst nur die Grundfunktionen realisiert und durch funktionelles Testen verifiziert. Sich daran anschließende Erweiterungen betrafen hauptsächlich die Behandlung zusätzlicher Systemcalls von fehlertoleranten Benutzerjobs über die reinen Ein/Ausgabe-Operationen an den Maskierungsschnittstellen hinaus, um die geforderte Anwendungs-Transparenz zu verbessern. Hierdurch wurden entsprechende Erfahrungen gewonnen, die eine **Konzeptvalidierung hinsichtlich** dieser **Nutzbarkeitseigenschaft** Transparenz erlauben. Wenngleich das Grundkonzept sich auf die elementaren Operationen *read* (zurVerteilung von Eingabedaten auf die Replikate) und *write* (zur Votierung über Jobresultate für die Fehlermaskierung) zunächst beschränkte - auch um von den Besonderheiten des jeweiligen Basisbetriebssystems in Hinblick auf die Eigenschaft Portierbarkeit zu abstrahieren - soll natürlich die Möglichkeit zur Erweiterung auch unter Berücksichtigung solcher Spezifika gegeben sein. Dazu muß an der Schnittstelle zwischen Standardkern und Betriebssystem-Erweiterung leicht Einfluß auf bestehende Mechanismen des lokalen Betriebssystems genommen werden können, was durch den gewählten Kopplungsmechanismus besonders gut gewährleistet ist.

Im einzelnen wurde deutlich:

- Die Transparenz der Grundfunktionen kann überschaubar realisiert werden. Bei jeder Erweiterung muß sorgfältig geprüft werden, inwieweit die Fehlertoleranzeigenschaft davon berührt wird, und/oder ob der allgemeine verteilte Systemcharakter allein schon Sonderbehandlung erfordert.
- Die erforderlichen Sonderbehandlungen können leicht sehr aufwendig werden. Insbesondere kann es notwendig werden, Aktionen des lokalen BS auf der FTL-Ebene zur globalen Synchronisation nachzuempfinden (d.h. doppelt auszuführen), wenn nicht gar nachträglich zu korrigieren.
- Es treten Probleme mit indeterministischem Verhalten der Prozeßreplikate an den Systemcall-Schnittstellen auf. Indeterminismus wird unter bestimmten Betriebsbedingungen erst durch das lokale Betriebssystem, z.B. durch dessen asynchrone E/A-Bearbeitung, eingebracht. In solchen Fällen ist ein Übereinstimmungsprotokoll zum Abgleich von Eingabedaten und -frequenz unumgänglich. Wegen der hohen Kosten ist dies unerwünscht.
- Aus ähnlichen Gründen macht die Signalbearbeitung erhebliche Probleme. Asynchrone Signale sind inhärent indeterministische Ereignisse. Dürfen sie den Programmfluß der Prozeßreplikate beeinflussen, ist Synchronisationsverlust auch im fehlerfreien Fall zu befürchten. Der Programmfluß fehlertoleranter Systeme mit statischer Redundanz, die lose synchronisiert und dazu auf hohem Implementationsniveau softwarerealisiert sind, kann so gut wie nicht synchronisiert werden. Dazu ist der Einsatz zusätzlicher dynamischer Redundanztechniken erforderlich (z.B. Checkpointing, verbunden mit einem Mechanismus zur interaktiven Konsistenz), oder eine Hardwareunterstützung.

Wie bei allen Systemen, die dem Benutzer mehrere nebenläufige Jobs ermöglichen, ist die leichte **Handhabbarkeit** derselben als ein Aspekt der **Nutzbarkeit** ein wichtiger Punkt, der besondere Beachtung verdient. Dieses in verteilten Systemen allgemeine Problem wird noch durch die Fehlertoleranzeigenschaft erschwert, da hier jobbezogene Ein-/Ausgabekanäle von mehreren Jobreplikaten zu koordinieren sind, und die Fehlertoleranz zusätzliche Bedienfunktionen erfordert. In diesem Zusammenhang wurde eine fenstergesteuerte Benutzeroberfläche entwickelt. Im ATTEMPTO-System ist Parallelität auf Jobebene alternativ zu oder gemeinsam mit fehlertoleranten Jobs möglich. Der Benutzer spezifiziert den Fehlertoleranzgrad seiner Jobs bei deren Start, kann also Parallel-Verarbeitungsleistung zugunsten von Fehlertoleranz opfern. Ohne eine entsprechende Benutzeroberfläche - im Sinne einer jobbezogenen Multi-Window-Terminal-Emulation - ist eine übersichtliche Kontrolle parallel arbeitender Jobs gar nicht möglich. Neben einem Kommandofenster zum Jobstart gibt es für jeden Job im System ein separates Fenster, dem verschiedene Statusattribute wie Ausführungsort(e), Fehlertoleranzgrad, etc., zugeordnet sind.

Hinsichtlich der **Testbarkeit** kann folgende Bewertung gelten:

- Wegen der Modulkapselung in separaten Prozessen sind die Modulfunktionen gut spezifizierbar und an ihren Schnittstellen leicht überprüfbar.
- Die gewählte Systemarchitektur gestattet die einfache Einbettung einzelner oder auch mehrerer unveränderter Module in eine leistungsfähige baukastenartige Testumgebung, die ohne großen Aufwand erstellbar ist.

- Die Nachrichtenkopplung erschwert eine Propagierung von Implementationsfehlern über Modulgrenzen hinaus, sodaß der Programmverifikation ein einfaches Fehlermodell zugrundegelegt werden kann.
- Die Kausalität der Prozeßinteraktion ist leicht erkennbar durch Examinierung von Nachrichten-Traces. Diese können an zentraler Stelle eines jeden Knotens erfaßt werden.
- Integrationstests von modulübergreifenden Funktionen sind erschwert durch die indeterministische Durchmischung von lokalen und globalen Botschaften im Post-Office. Sorgfältige Testfallklassifizierung ist in diesem Zusammenhang unerläßlich.

Eine wichtige Bedeutung für die vorliegende Arbeit nimmt die **Bewertung des Leistungsverhaltens** des fehlertoleranten Systems ein. Dies dient hauptsächlich zur Beurteilung der Kosten für die Fehlertoleranz und ist gleichzeitig Grundlage für eine **Leistungsoptimierung** im System ATTEMPTO. Vielfältiger Datentransport, das Abarbeiten von Kommunikationsprotokollen und häufige Prozeßwechsel führten zu einer nicht unerheblichen Leistungsverminderung in der ersten Prototyp-Realisierung, sodaß eine Optimierungsphase angebracht war. Meßergebnisse zeigen - wie übrigens bei fast allen vergleichbaren Systemen auch, die in ähnlicher Weise Erweiterungen aufsetzen auf bestehende Verhältnisse -, daß der Aufwand für Modularität und Flexibilität zu Lasten des Leistungsvermögens geht. Die Maximalfrequenz von Maskierungsvorgängen, als ein handliches Maß zur Beschreibung des Leistungsvermögens im fehlertoleranten System, liegt gegenwärtig bei weniger als 10 Maskierungen/s. Während sich dies bei der Ausgabe von längeren Zeichenketten als atomare Schreiboperation nicht besonders nachteilig bemerkbar macht, führt eine zeichenweise Ausgabe in schneller Folge zu einer stark reduzierten effektiven Übertragungsrate auf der Terminalleitung.

Für den Leistungsabfall sind verschiedene Gründe verantwortlich:

- Die Kopplung zwischen Standardkern und FTL ist lose.
- Zusätzliche System-Prozesse sind an der User-Job-Aktion beteiligt.
- Prozeß-Kernel-Interaktionen sind vervielfacht.
- Prozeß-Scheduling ist nicht explizit global synchronisiert. De facto ergibt sich allerdings eine Synchronisation an Entscheidungspunkten, an denen die Fehlertoleranzschicht Ergebnis- bzw. Zustandsmeldungen von beteiligten Kollegen erwartet. In der Regel schläft der Benutzerprozess während dieser Zeit im überwachten Systemaufruf.
- Die langsamen Prozeßreplikate bestimmen die Geschwindigkeit. Die Anwendungsprozeßexemplare unterliegen einer ereignisgesteuerten geregelten Synchronisation, wobei die Ausgabewerte des Anwenderprozesses für den dezentralen Maskierungsentscheid in Puffern aufgesammelt werden. Zur Zeit beträgt die Puffertiefe 1 Jobergebnis, was eine relativ starre Synchronisation bedeutet, ein Protokoll zur Flußkontrolle aber unnötig macht und somit die Implementation erleichtert. Der wesentliche Grund jedoch liegt in der Zeitschrankenüberwachung begründet, die wesentlich einfacher wird, wenn eine Akkumulation von Laufzeitdifferenzen nicht in Betracht gezogen werden muß.
- Multicast- und Zuverlässigkeitseigenschaften des Kommunikationssystems sind ausschließlich software-implementiert. Der Zeit-Overhead zur Übertragung einer Nachricht ist nahezu konstant und richtet sich nach der maximalen Paketgröße unabhängig von der Menge der tatsächlich transportierten Daten, sowie auch nach der Maximalzahl der Knoten im System, gleich wieviele davon augenblicklich in der Adressatenmenge des

Multicasts enthalten sind. Das Senden erhält durch den starren Zeitrahmen einen synchronen Charakter, der die Arbeit der lokalen Fehlertoleranzinstanz unerwartet stark behindert.

- Wegen der Modulkapselung sollen Datenobjekte nicht referenziert werden, sondern müssen kopiert werden. Dies gilt selbst für Leichtgewichtsprozesse. Als Folge davon ist auch die Interprozeßkommunikation zwischen Fehlertoleranz-Clerks recht zeitaufwendig, obwohl eine Optimierung schon die Anzahl der Kopiervorgänge durch Abkürzung von Nachrichtenwegen verringern konnte.
- Der verteilte Diagnosealgorithmus, der die Grundlage für die Fehlermaskierung darstellt, optimiert zwar die Nachrichtenkomplexität, nicht aber die Zeitkomplexität. Für hohe Leistung ist die letztere allein entscheidend. Die Anzahl der Nachrichtenaustauschphasen im Maskierungsmechanismus muß in erster Linie gering gehalten werden, wie ein Ansatz zur Leistungsoptimierung gezeigt hat. In dieser Hinsicht ist ein einfaches Verteilungsprotokoll zur Fehlermaskierung vorteilhafter, obwohl es mehr Nachrichten benötigt.
- Für weitreichende Transparenz sind verhältnismäßig viele Systemcalls der überwachten Prozesse sonderzubehandeln. Der Durchsatz wird also nicht nur durch die Maskierung von Ausgabedaten begrenzt, sondern auch durch verlorene Rechenleistung für den notwendigen Abgleich zur Wahrung der Konsistenz von Systemdaten, um einen deterministischen Programmlauf der Prozeßreplikate zu garantieren.

Viele dieser Gründe resultieren direkt aus der modularen Struktur und der losen Kopplung der Replikate. Eine Forderung nach hoher Maskierungs-Leistung könnte sich also nicht homogen in die übrigen Systemanforderungen einfügen.

Abgesehen von diesen Leistungsbeschränkungen, denen durch entsprechend leistungsfähige Grundkomponenten begegnet werden muß, besitzt das ATTEMPTO-Konzept attraktive Eigenschaften, die es vor anderen Fehlertoleranzkonzepten auszeichnet. Die leichte Anpassungsfähigkeit auf vielfältige Systemplattformen, die prinzipiell einfachen Grundmechanismen, die in erster Linie auf seinem Charakter als maskierendes System beruhen, aber auch umgesetzt in der konkreten Systemimplementation durch die modulare Softwarearchitektur noch einfach durchschaubar sind, sowie die Flexibilität bei unterschiedlichen Zuverlässigkeitsanforderungen begründen die gute Eignung für eine Mehrzweckfehlertoleranz. Besonders vorteilhaft ist, daß die Fehlertoleranz als Betriebssystemdienst in der Anwendung keine Berücksichtigung finden muß, sondern daß sogar unveränderte, unter dem lokalen Monoprozessor-Betriebssystem ausführbare Binärprogramme unter dem erweiterten Multiprozessor-Betriebssystem fehlertolerant ablaufen können.

8. Literaturverzeichnis

[ABCLR83] E.Ammann, R.Brause, M.Dal Cin, E.Dilger, J.Lutz, T.Risse, *ATTEMPTO: A Fault-Tolerant Multiprocessor Working Station*, Design and Concepts. Proc. FTCS-13 (1983) 10-13

[ACD82] J-M.Ayache, J-P.Courtiat, M.Diaz, *REBUS, A Fault-Tolerant Distributed System For Industrial Real-Time Control*. IEEE Trans. Comp. Vol. C-31 (1982) 637-647

[ADC81] E.Ammann, M.Dal Cin, *Efficient algorithms for comparison based self-diagnosis*, in Dal Cin,M. , Dilger,E. (eds): Self-Diagnosis and Fault-Tolerance, Attempto Verlag Tuebingen (1981) 1-18

[AK88] H.R.Aschmann, H.Kirrmann, *ALPHORN, A Toolbox for Programming Fault-Tolerant Distributed Process Control Systems.* Asea Brown Boveri Research Report CRB 88-027 C, 1988

[Alt93] J.Altmann, *Diagnoseprotokolle in Multiprozessoren.* Dipl.Arb. Univ. Erlangen/ Nürnberg 1993

[Amm82] E.Ammann, *Vergleichstestmodelle für selbstdiagnostizierbare Systeme.* Informatik Fachberichte 54 Springer (1982) 74-87

[AnLe81] T.Anderson, P.A.Lee, *Fault Tolerance - Principles and Practice.* Prentice Hall, London, 1981

[Av71] A.Avizienis et al., *The STAR Computer: An Investigation into the Theory and Practice of Fault-Tolerant Computing.* IEEE Trans. Comp., Vol. C-20 (1971) 1312-1321

[AvCh78] A.Aviziensis, L.Chen, *N-Version Programming: A Fault-Tolerant Approach to Reliability of Software Operation.* Proc FTCS-8 (1978) 3-9

[Bab90] Ö.Babaoglu, *Fault-Tolerant Computing Based on Mach.* ACM Operating Systems Rev., Vol. 24 (1990) 27-39

[Bach86] M.Bach, *Design of the UNIX Operating System.* Prentice Hall 1986

[BaKeWi90] H.R.Baader,H.Kersten, E.vanWickeren: *Werkzeuge zur formalen Spezifikation und Verifikation vertrauenswürdiger Systeme.* In H.Kersten (Hrsg.): Sichere Software. Hüthig 1990

[BS84] T. Basil Smith, *Fault-Tolerant Processor Concepts and Operation.* Proc. FTCS-14 (1984) 158-163

[Bart81] J.Bartlett, *A Nonstop Kernel.* Proc. 8th Symp. Oper. Sys. Principles. ACM (1981) 22-29

[BBG83] A.Borg, J.Baumbach, S.Glazer, *A Message System Supporting Fault Tolerance*.

Proc. 9th Symp.Oper. Sys. Principles. ACM (1983) 90-99

[Ben90] M.Ben Or, *Randomized Agreement Protocols.* In: B.Simons(eds), *Fault-Tolerant Distributed Computing.* Lecture Notes in Comp. Science 448, Springer (1990) 72-83

[Bern88] P.A.Bernstein, *Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing*. IEEE Computer 21(2) (1988) 37-45

[BGM76] F.Barsi, F.Grandoni, P.Maestrini, *A theory of diagnosability of digital systems.* IEEE Trans.Comp., vol C-25 (1976) 585-593

[BJ85] K.Birman, T.Joseph, *Comminication Support For Reliable Distributed Computing*. In: B.Simons(eds), *Fault-Tolerant Distributed Computing.* Lecture Notes in Comp. Science 448, Springer 1990, 124-137

[Blount77] M.Blount, *Probabilistic treatment of diagnosis in digital systems.* Proc. FTCS-7 (1977) 72-77

[BM91] M.Banatre, G.Muller et al., *Design Decisions for the FTM: A General Purpose Fault Tolerant Machine.* IRISA, Universität Rennes, Interne Veröffentlichung Nr. 570, Januar 1991

[Borg89] A.Borg, W.Blau et al, *Fault Tolerance Under UNIX*. ACM Trans. Comp. Sys.-Vol. 7 (1989) 1-24

[Brau87] R.Brause, *Simulation eines fehlertoleranten Multi-Prozessorsystems unter UNIX*. Proc. GI-Workshop, ASIM Müchen 1987

[BMR82] D.Brownbridge,L.Marshal,B.Randell, *The Newcastle Connection or UNIXes of the World Unite!* Software Practice and Experience 12 (1982) 1147-1162

[CGR88] R.F.Cmelik, N.H.Gehani, W.D.Roome, *Fault-Tolerant Concurrent C: A Tool For Writing Fault-Tolerant Distributed Programs*. IEEE Trans. Comp. (1988) 56-61

[CAS85] F.Cristian, H.Aghili, R.Strong, *Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement.* IEEE Proc FTCS-15 (1985) 200-206

[CDSA90] F.Cristian, D.Dolev, R.Strong, H.Aghili, *Atomic broadcast in a real-time environment*. In: B.Simons(eds), *Fault-Tolerant Distributed Computing.* Lecture Notes in Comp. Science 448, Springer 1990, 51-71

[CHak81] S.L.Hakimi, K.Y.Chwa, *Schemes for fault-tolerant computing: a comparison of modularly redundant and t-diagnosable systems.* Information and Control, vol.49 (1981) 212-238

[D83] D.Dolev, et al., *Reaching Approximate Agreement in the Presence of Faults.* Proc. IEEE Symp. Distrib. Software a. Databases (1983)

[DaSK85] A.T.Dahbura, K.K.Sabnani, L.L.King, *The comparison approach to multiprocessor fault diagnosis.* Proc FTCS-15 (1985) 260-265

[DaWa78] D.Davies, J.F.Wakerly, *Synchronization and Matching in Redundant Systems.* IEEE Trans. Comp. Vol. C-27 (1978) 531-539

[DBLDR87] M.Dal Cin, R.Brause, J.Lutz, E.Dilger, T.Risse, *ATTEMPTO-An Experimental Fault-Tolerant Multiprocessor System.* Microprocessing a. Microprogr. 20 (1987) 301-308

[DC79] M.Dal Cin, *Fehlertolerante Systeme.* Teubner Studienbücher Informatik, Stuttgart 1979

[DC81] M.Dal Cin, *Self-diagnosis for parallel computers.* Informatik Fachberichte (IFB) 50 Springer (1981) 285-292

[DC82] M.Dal Cin, *A diagnostic device for large multiprocessor systems.* Proc. FTCS-12 (1982) 357-360

[DC84] M.Dal Cin, *Distributed Diagnosis for Computing Networks.* Micropocessing a. Microprogramming 14 (1984) 139-144

[DC87] M.Dal Cin, *Ein Diagnoseverfahren für Systeme mit mehreren Verarbeitungseinheiten.* IFB 147, Bremerhaven, Springer (1987)191-198

[DC88] M.Dal Cin, *On Explicit Fault-Tolerant Parallel Programing.* Interner Bericht 4/88 J.W.Goethe-Universität Frankfurt/M 1988

[DC89] M.Dal Cin, et al., *Mechanismen zur Fehlerdiagnose und -behebung für die MEMSY-Hochleistungsstruktur.* Arbeitsber. des IMMD, Band 22, Nr.13, Erlangen (1989) 113-130

[DC90] M.Dal Cin, *On Distributed System-Level Self-Diagnosis.* IFB 214 (1990) 187-196

[DCD81] M.Dal Cin, E.Dilger (Eds), *Self-Diagnosis and Fault-Tolerance.* Werkhefte der Univ. Tübingen. ATTEMPTO-Verl. Tübingen 1981

[DCF85] M.Dal Cin, F-H.Florian, *Analysis of a Fault-Tolerant Distributed Diagnosis Algorithm.* Proc. FTCS-15 (1985) 159-164

[Dent68] J.Dent, *Diagnostic Engineering Requirements.* Spring Joint Computer Conference (1986) 503-507

[DeRi82] F.Demmelmeier, W.Ries, *Implementierung von anwendungsspezifischer Fehlertoleranz für Prozeßautomatisierungssysteme.* IFB 54, Springer (1982) 299-314

[DiAm84] E.Dilger, E.Ammann, *System-Level Self-Diagnosis in n-Cube-Connected Multiprocessor Networks.* FTCS-14 (1984) 184-189

[DM83] A.Dahbura, G.Masson, *Greedy diagnosis of hybrid fault situation.* IEEE

Trans.Comp. Vol. C-32 (1983) 777-782

[DolStro82] D.Dolev, H.Strong, *Polynomial Algorithms For Multiple Processor Agreement.* JACM (1982) 401-407

[DolStro85] D.Dolev, H.Strong: *A simple model for agreement in distributed systems.* In: B.Simons(eds), *Fault-Tolerant Distributed Computing.* Lecture Notes in Comp. Science 448, Springer (1990) 42-50

[Dol82] D.Dolev, *The Byzantine Generals Strike Again.* JACM 3 (1982) 14-30

[DR85] D.Dolev, R.Reischuk, *Bounds on Information Exchnge for Byzantine Agreement*. JACM 32 (1985) 191-204

[DRS82] D.Dolev, R.Reischuk, R.Strong, *Early Stopping in Byzantine Agreement.* JACM (1987).

[Echt90] K.Echtle, *Fehlertoleranzverfahren.* Studienreihe Informatik, Springer 1990

[Egan88] J.Egan, T.J.Teixeira, *Writing a UNIX Device Driver.* Wiley, New York 1988

[Fär81] G.Färber, *Task-Specific Implementation of Fault Tolerance in Process Automation*. In Dal Cin,M. , Dilger,E. (eds): S*elf-Diagnosis and Fault-Tolerance,* Attempto Verlag Tuebingen (1981) 84-102

[FLM85] M.Fisher,N.Lynch,M.Merritt: E*asy Impossibility Proofs for Distributed Consensus Problems.* Distributed Computing vol.1. (1985). Auch in: B.Simons(eds), *Fault-Tolerant Distributed Computing.* Lecture Notes in Comp. Science 448, Springer 1990, 146-170

[FLP85] M.Fisher, N.Lynch, M.Paterson, *Impossibility of Distributed Consensus with One Faulty Process*. JACM 32, (1985) 374-382

[Frei82] R.Freiburghouse, *Making Processing Fail-Save.* Mini-Micro Systems (1982) 228-233

[FrWe82] S.G.Frison, J.H.Wensley, *Interactive Consistency and its Impact on the Design of TMR Systems.* Proc FTCS-12 (1982) 228-233

[FuRa88] Fussel, S.Rangarajan, *A probabilistic method for fault diagnosis of multiprocessor systems.* Proc. FTCS-18 (1988) 278-283

[Gad92] K-H.Gadow, *Automatische Generierung von Softwarediversität auf prozeduraler Hochsprachenebene als Mittel der Hardwarefehlertoleranz.* Diplomarbeit Univ. Erlangen/Nürnberg 1992

[Glaz84] S.D.Glazer, *Fault-Tolerant Mini Needs Enhanced Operating System.* Computer Design (1984) 239-245

[Gunn83] P.Gunningberg, Voting and Redundancy Managemant Implemented by Protocols in Distributed Systems. Proc. FTCS-13 (1983) 182-185

[Gü88] W.Günter, *Fehlertolerante Interprozessorkommunikation in ATTEMPTO.* Im Arbeitsbericht 1988 für das DFG-Projekt Da141/5

[Gü88a] W.Günter, *Einrichten einer Entwicklungsumgebung zur Systemprogrammierung.* Im Arbeitsbericht 1988 für das DFG-Projekt Da141/5

[Gü92] W.Günter, *CPUs nutzen gemeinsam LAN-Controller.* VMEbus, 2 (1992) 36-40

[Grill93] P.Grill, *Leistungsoptimierung im fehlertoleranten Multiprozesor ATTEMPTO.* wird erscheinen als Stud.Arbeit Univ. Erlangen/Nürnberg 1993

[Hamm91] D.Hamm, *TOAST Tracer Output Analysis System. Manual.* Interne Dokumentation J.W.Goethe-Universität 1990

[HM84] R.Horst, S.Metz, *New system manages hundreds of transactions per second.* Electronics (1884) 223-227

[HSL78] A.L.Hopkins, T.B.Smith, J.H.Lala, *FTMP - A Highly Reliable Fault-Tolerant Miltiprocessor for Aircraft*. Proc.IEEE, Vol.66 (1978) 1221-1239

[Jew91] D.Jewett, *Integrity S2: A Fault-Tolerant UNIX Platform*. Proc. FTCS-21 (1991)

[Klau92] G.Klausen, *Fenstergesteuerte Benutzeroberfläche für den fehlertoleranten Arbeitsplatzrechner ATTEMPTO.* Stud.Arbeit Univ. Erlangen/Nürnberg 1992

[Kam89] K.Kammers, *Parallelisierung prozeduraler, blockorientierter Programmiersprachen.* Dipl.Arb. Fachb. Informatik, J.W.Goethe-Univ. Frankfurt/M 1989

[Kn90] H.Kneilmann, *Multihost-Ethernet-Konzentrator am VMEbus*. Dipl.Arb. Fachb. Informatik, J.W.Goethe-Univ. Frankfurt/M 1990

[Kram91] M.Kramer, *Software-implementierte Fehlertoleranzmechanismen für objektorientierte, verteilte Systeme*. Dissertation Universität Erlangen/Nürnberg 1991

[KuRe80] J.Kuhl, S.Reddy, *Distributed fault-tolerance for large multiprocessor systems.* Proc. 7th Symp.Comp.Arch. (1980) 23-30

[KuRe81] J.Kuhl, S.Reddy, *Fault-diagnosis in fully distributed systems.* Proc. FTCS-11 (1981) 100-105

[La81] B.Lampson, *Atomic Transactions. In: Distributed Systems - Architecture and Implementation,* B.Lampson, Springer (1981) 246-265

[Lala85] P.K.Lala, *Fault Tolerant and Fault Testable Hardware Design.* Prentice Hall 1985

[Lap90] J.C.Laprie (Ed.), *Dependability: Basic Concepts and Terminology*. IFIP WG 10.4 Dependable Computing ans Fault Tolerance. (In Deutsch erschienen im Springer Verlag)

[Lam82] L.Lamport, R.Shostak, M.Pease, *The Byzantine Generals Problem.* ACM Trans. Progr. Languages a. Systems, Vol. 4 (1982) 382-401

[Lam83] L.Lamport, *Weak Byzantine Generals Problem*. JACM 30 (1983) 668-676

[Lee90] S.Lee, *Probabilistic multiprocessor and multicomputer diagnosis.* Ph.D.Dissertation, Univ. of Michigan 1990

[LShin90] S.Lee, K.Shin, *Optimal multiple syndrome probabilistic diagnosis.* Proc FTCS-20 (1990) 324-331

[LSM82] C.Liaw, S.Su,Y.Malaiya, *Self-Diagnosis of Non-Homogenous Distributed Systems.* Proc. FTCS-12 (1982) 349-352

[Lu89a] J.Lutz, R.Brause, M.Dal Cin, T.Philipp, *Parallelisierungskonzept für ATTEMPTO-2.* Interner Bericht 1/89, Fachb. Informatik, J.W.Goethe-Univ. Frankfurt/M 1989

[Lu89b] J.Lutz, R.Brause, M.Dal Cin, K.Kammers,T.Philipp, *Die Erweiterungen der ATTEMPTO-2 Laufzeitbibliothek.* Interner Bericht 2/89, Fachb. Informatik, J.W.Goethe-Univ. Frankfurt/M 1989

[Mae82] E.Maehle, *Fehlertolerantes Verhalten in Multiprozessoren. Untersuchungen zur Diagnose und Rekonfiguration.* Dissertation Univ. Erlangen/Nürnberg 1982

[Män86] H.Mäncher, *Ein fehlertolerantes Microrechnersystem für die dezentrale Prozeßautomatisierung.* Dissertation D17 der TH-Darmstadt 1986

[Malek80] M.Malek, *A comparison connection assignment for diagnosis of multiprocessor systems.* Proc. 7. Symp. on Comp. Arch., (1980) 31-35

[Malek82] M.Malek, J.Maeng, *Partitioning of large multicomputer systems for efficient fault diagnosis*. Proc. FTCS-12 (1982) 341-346

[MAN90] D.Manske, *Fehlertolerante UNIX-Systeme*. Fachbericht IAB Nr.32 Universität-GH-Paderborn 1990

[MSS82] P.M.Melliar-Swmith, R.L.Schwartz, *Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerant Flight Control System.* IEEE Trans.Comp. Vol. C-31 (1982) 616-630

[Ne 88] J.Nehmer, *Entwurfskonzepte für Verteilte Systeme, eine kritische Bestands-aufnahme*. IFB, GI-Jahrestagung 88 (1987) 70 - 96

[MM78] S.Mallela, G.Masson, *Diagnosable Systems for Intermittent Faults.* IEEE Trans.Comp. Vol. c-27 (1978) 560-566

[Mo83] K.Moritzen, *Verteilte Selbstdiagnose in Rechnersystemen.* Diplomarbeit Univ. Erlangen/Nürnberg 1983

[MPR87] Broschüre: *MPR 1300-SD, Mehrprozessor-Synchron-Dup;ex-Rechner. Kurzbeschreibung.* Krupp Atlas Elektronik GmbH 1987

[Oppm93] H.Oppmann, *Selbsttests und Diagnose im fehlertoleranten Multiprozessor ATTEMPTO*. Wird erscheinen als Diplom-Arbeit Univ. Erlangen/Nürnberg 1993

[Pet67] W.Peterson, *Prüfbare und korrigierbare Codes.* R.Oldenburg Verlag München-Wien 1967

[Phil89] T.Philipp, Beiträge zum Arbeitsbericht 1989 für das DFG-Projekt Da141/5

[Phil91] T.Philipp, *An Expert System Shell for the Diagnosis of Parallel Computers.* Proc. 5th Int. GI/ITG/GMA Conf. Fault Tolerant Computing Systems (1991). IFB 283 (1991) 77-87

[PMC67] F.P.Preparata, G.Metze,R.T.Chien, *On the connection assignment problem of diagnosable systems.* IEEE Trans.Electron. Comp., vol EC-16 (1967) 848-854

[Popek81] Popek,J. et al: *LOCUS, a network transparent, high reliability distributed system*. Proc 8th SOSP. Monterey 1981

[RBDDL84] T.Risse, R.Brause, M.Dal Cin, E.Dilger, J.Lutz, *Entwurf und Struktur einer Betriebssystemschicht zur Implementierung von Fehlertoleranz,* IFB 84, Springer (1984) 66-76

[Reh90] A.Rehor, *Profiling von Modula-2-Programmen für verteilte Systeme*. Dipl.Arb. Fachb. Informatik, J.W.Goethe-Univ. Frankfurt/M 1990

[Rei87] R.Reischuk, *Konsistenz und Fehlertoleranz in Verteilten Systemen - Das Problem der Byzantinischen Generäle.* IFB 156, Springer (1987) 65-81

[Ris87] T.Risse, *Modelling Interrupt Based Interprocessor Communication by Time-Petri-Nets.* Interner Bericht 1/87, Fachb. Informatik, J.W.Goethe-Univ. Frankfurt/M 1987

[SBB92] M.Stahl, R.Buskens, R.Bianchini, *On-line diagnosis in general topology networks.* In: Workshop on Fault-Tol. Parallel a. Distr. Systems, IEEE Comp. Society, Massachusetts (1992) 114-121

[Schm85] J.Schmalzried, *Fehlertolerante Benutzer-Schnittstelle.* Dipl.Arb. im Fachb. Physik der Univ. Tübingen, Mai 1985

[Schn90] F.Schneider, *The State Machine Approach: A Tutorial.* In: B.Simons(eds), *Fault-Tolerant Distributed Computing.* Lecture Notes in Comp. Science 448, Springer 1990, 18-41

[Ser89] O.Serlin, *Commercial Fault-Tolerant Sytems: Issues and Prospects.* In T.Anderson (Eds), *Dependability of Resilient Computers.* BSP Professional Books, Chapter 9, Oxford, England 1989

[SGS84] F.Schneider, G.Gries, R.Schlichting, *Fault-Tolerant Broadcasts*. Science of Comp. Progr. 4 (1984) 1-15

[Siew82] D.P.Siewiorek, R.S.Swarz, *The Theory and Practice of Reliable System Design*. Digital Press, Bedford MA, 1982. Auch in: D.P.Siewiorek, *Faults and Their Manifestation*. Lecture Notes in Computer Science 448, Springer 1990, 244-261

[SIM90] Dr.Daar, *Redundante Automatisierungssysteme SIMATIC S5*. Siemens AG Bereich Automatisierungstechnik. Erlangen 1990.

[Sim90] B.Simons, J. Lundelius Welch, N.Lynch, *An Overview of Clock Synchronization*. In: B.Simons(eds), *Fault-Tolerant Distributed Computing*. Lecture Notes in Comp. Science 448, Springer 1990, 84-96

[Strehl] V.Strehl, *Logik und Berechenbarkeit* I, Vorl.-Scriptum Univ. Erlangen/ Nürnberg, 1987

[Tou84] S.Toueg, *Randomized Byzantine Agreement*. Proc. 3th Princ. Distr. Computing (1984) 163-178

[Trib89] T.Tribius, *Ein paralleles kooperierendes Prolog-Prozeß-System*. Dipl.Arb. Fachb. Informatik, J.W.Goethe-Univ. Frankfurt/M 1989

[Trib92] T.Tribius, *Ein programmiersprachlicher Ansatz zur Entwicklung von explizit-fehlertoleranten Programmen auf Parallelrechnern*. Dissertation Universität Erlangen/Nürnberg 1992

[Triv82] S.Trivedi, *Probability and Statistics with Reliability, Queueing and Computer Science Applications*. Prentice Hall Inc., Eaglewood Cliffs 1982

[TW89] D.Taylor, G.Wilson, *Stratus*. In T.Anderson (Eds), *Dependability of Resilient Computers*. BSP Professional Books, Chapter 10, Oxford, England 1989

[VME87] Draft 2.2 of MSC Working Group P1014, *VMEbus, a standard specification for a versatile backplane bus*. IEEE Computer Society 1987

[Wen78] J.H.Wensley et al.: *SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control*. Proc. IEEE Vol. 66, (1978) 1240 - 1255

[Wen81] J.H.Wensley, *Fault-tolerant Computers ensure reliable industrial controls*. Electronic Design, Vol. 29 (1981) 198-204

[Wenz91] G.Wenzel, *Debugging-Tool für eine verteilte Systemumgebung*. Dipl.Arb. Fachb. Informatik, J.W.Goethe-Univ. Frankfurt/M 1991

[Wi88] N.Wirth, *Programming in MODULA-2*. Springer (1988)

[YaMa] C-L.Yang, G.Masson, *A Generalization of Hybrid Fault Diagnosis*. Proc. FTCS-15 (1985) 36-41

[YST85] T.Yoneda, T.Suzuoka, Y.Tohma, *Implementation of Interrupt Handler for Loosly-Synchronizes TMR Systems*. Proc. FTCS-15 (1985) 246-251

9. Anhang

9.1 Robustes Mailbox-Adreß-Mapping in ATTEMPTO

9.1.1 Ausgangssituation

Die Hardware der Single-Board-Computer (SBC) ermöglicht eine variable Abbildung von Teilbereichen des lokalen Speichers in den globalen VMEbus-Adreßbereich (als *Slave-Mode*-Adreßraum). Die Konversion wird in einem PROM (alternativ in einem PAL) vorgenommen, und besteht darin, die oberen 8 VMEbus-Adreßleitungen (A16-A23) für die Dauer des Buszugriffes auf entsprechende lokale Adreßleitungen umzusetzen.

Die Minimalgröße des Slave-Mode-Adreßraumes für einen SBC beträgt 64KByte; eine Beschränkung der Maximalgröße existiert nur insofern, als auf einem Board lokale und globale Adressen unterscheidbar sein müssen.

Die Umsetzung ist dynamisch variierbar über 4 programmierbare Steuerleitungen, wobei eine Kombination für die Abschaltung der Konversion reserviert ist.

9.1.2 Anwendung in ATTEMPTO

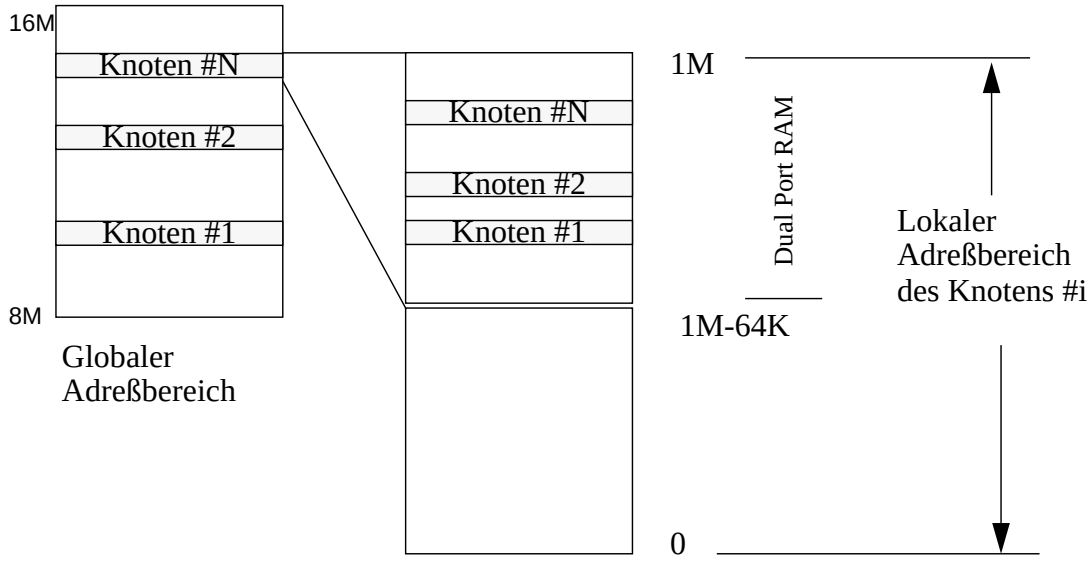


Abb. 9 - 1 Adreß-Mapping

In ATTEMPTO beitzt jeder der N SBCs jeweils N Mailboxen für den Botschaftsaustausch: 3<=N<= nmax. nmax ist bedingt durch weitere Hardwarevoraussetzungen auf gegenwärtig 7 begrenzt. Das Übermittlungsprotokoll ist fehlertolerant, reagiert jedoch sensibel auf Verfälschung der Nachrichten durch Fremdeinfluß (durch andere SBCs), z.B. als Folge transienter Störungen auf Adreßleitungen während eines Datentransferzyklus auf dem VMEbus. Da hier keine Möglichkeit zur Fehlerkorrektur besteht, wird versucht, durch geeignete Wahl der Mailbox-Basisadressen die Wahrscheinlichkeit für schädliche Störauswirkungen zu minimieren.

Hierbei ist zu unterscheiden zwischen einer fehlerhaften SBC-Dekodierung und einer Fehldekodierung von Mailboxen auf einem SBC, d.h. die Mailboxbasisadressen müssen so kodiert sein, daß sich sowohl im globalen wie im lokalen Memory-Map ausreichender Sicherheitsabstand ergibt.

Für alle N Mailboxen auf einem SBC wird ein Bereich von 64KByte (entsprechend der Minimalgröße s.o.) reserviert. Die grundsätzliche Struktur zeigt Abb. 9 - 1.

9.1.3 Globales Adreß-Mapping

Gesucht: (wenigstens) 7 binäre Vektoren eines Linearcodes (n,k), die bei n=8 einen maximal möglichen Hammingabstand besitzen.

Diese Vektoren sind die Basisadressen der Mailboxen im globalen Adreßbereich bzw. deren 8 höchstwertige Stellen. Angestrebt wird eine möglichst kompakte Ausnutzung des VMEbus-Adreßraumes.

Es sei mit

n = Anzahl der Codestellen im Codewort w;
k = Anzahl der Nutzbits; n-k = Anzahl der Prüfstellen (Redundanzanteil);

$I = (i_1,i_2,...i_k)$	ein k-Tupel von Binärelementen und
(**I**)	die Matrix des Vektorraumes gebildet aus allen k-Tupeln als Zeilenvektoren,
$P = (p_1,p_2,...,p_{n-k})$	ein Prüfvektor entstanden aus
P = I (**C**)	mit (**C**) *Codematrix*,

dann wird ein Codewort $W = (w_1,w_2,..,w_n)$ gebildet durch

W = (I,P)	Hinzufügen des Prüfvektors P zum Zeilenvektor I bzw.
W= I { (**E**) | (**C**) },	(**E**) Einheitsmatrix,

oder mit

(**G**) = { (**E**) | (**C**) },	der Generatormatrix,
W = I (**G**).	

Diese Intepretation entspricht der Vorgehensweise bei der Erstellung eines *systematischen Codes*: der Nutzinformation wird Redundanz in Form von Prüfstellen angefügt, die nach einer bestimmten (durch die Codematrix (**C**) vorgegebenen Kodierungsvorschrift aus der Nutzinformation gebildet wird. Der so entstehende Linearcode aus 2^k Elementen wird dadurch ein Unterraum im Vektorraum aller n-Tupel.

Das Hinzufügen ist äquivalent einer Multiplikation des Nutzvektors mit einer Generatormatrix (**G**). Sie besitzt k linear unabhängige Zeilen (durch die Einheitsmatrix liegt die kanonische Staffelform vor). Die Menge aller Linearkombinationen der Zeilenvektoren von (**G**) (der Zeilenraum von (**G**)) ist der Coderaum.

Mit

(**W**) = (**I**)(**G**)

können alle Codevektoren erzeugt werden.

Unter der Annahme eines symmetrischen Binärkanals (ein Fehler bedeutet das "Umkippen" eines Binärelementes in seinen dualen Wert) wird die Wahrscheinlichkeit eines nicht erkennbaren Fehlers - d.h. die Verfälschung eines Codewortes derart, daß ein anderer Codevktor entsteht - um so geringer, je größer die Hammingdistanz des Codes ist.

Die *Hammingdistanz* eines Linearcodes entspricht dem kleinsten Gewicht seiner von Null verschiedenen Vektoren. Das *Hamminggewicht* eines Codevektors ist dabei die Anzahl seiner von 0 verschiedenen Elemente.

Es existiert eine Obergrenze für den erreichbare Hammingdistanz d_{max} in einem (n,k)- Linearcode:

$d_{max} <= n\ 2^{k-1}/(2^k-1)$ obere Schranke nach Plotkin [Pet67].

In unserem Fall mit n=8, k=3

$d_{max} <= 8*4/7$,

d.h. die maximale Hammingdistanz wird bestenfalls 4 betragen, ein Ergebnis, das sich auch mit n=7 erreichen läßt. Wir gehen daher von einem (7,3)- Code aus und legen die 8. Stelle fest zur Beschränkung des globalen Mailboxadreßraumes (A23 = 1 trägt der Forderung nach Kompaktheit Rechnung).

Die Codematrix (**C**) muß nun so beschaffen sein, daß sich der maximale Hammingabstand ergibt.

Sie ist (k,n-k)-reihig; ihre Spaltenvektoren c_i legen die Vorschrift zur Bildung der Prüfstellen p_i fest, 1<=i<=n-k :

$$p_i = I * c_i = \sum_{j=1}^{k} I_j * c_{ij} .$$

Für die Auswahl geeigneter Spaltenvektoren von (**C**) sind folgende Betrachtungen nützlich:

- Ein Null-Spaltenvektor erzeugt eine konstante Prüfstelle, die die Hammingdistanz nicht erhöht,
- Ähnliches gilt auch für die Spalten einer (k x k)-Einheitsmatrix; sie führen zu einer Wiederholung von Nutzbits,
- zwei gleiche Spalten in (**C**) können die Hammingdistanz nicht erhöhen.

Es verbleibt (mit k=3)

$$(\mathbf{C}) = \begin{bmatrix} 0\,1\,1\,1 \\ 1\,0\,1\,1 \\ 1\,1\,0\,1 \end{bmatrix}$$, wie gewünscht mit n-k Spalten.

Die Generatormatrix enthält alle möglichen 3-Tupel als Spaltenvektoren außer dem Nullvektor:

$$(\mathbf{G}) = \{ (\mathbf{E}) \mid (\mathbf{C}) \} = \begin{bmatrix} 1\,0\,0\,0\,1\,1\,1 \\ 0\,1\,0\,1\,0\,1\,1 \\ 0\,0\,1\,1\,1\,0\,1 \end{bmatrix}$$.

Der Coderaum wird erzeugt durch

$$(\mathbf{W}) = (\mathbf{I})(\mathbf{G})\ .$$

Die Rechnung wird sehr einfach durch die Äquivalenz

$$(\mathbf{W}) = \{ (\mathbf{I}) \mid (\mathbf{I})\,(\mathbf{C}) \},$$

die eine sukzessive Erstellbarkeit der Spalten von (**W**) ausweist, wenn man noch berücksichtigt, daß die Multiplikation der Binärmatrizen (**I**) und (**C**) nichts weiter ist als die Paritätsbildung über die Nutzbitstellen des jeweiligen Vektors I, die im zugehörigen Spaltenvektor von (**C**) mit 1 ausgewiesen sind.

Es ergibt sich

$$(\mathbf{W}) = \begin{bmatrix} W1 \\ W2 \\ W3 \\ W4 \\ W5 \\ W6 \\ W7 \\ W8 \end{bmatrix} = \begin{bmatrix} 0\,0\,0\,0\,0\,0\,0 \\ 0\,0\,1\,1\,1\,0\,1 \\ 0\,1\,0\,0\,1\,1\,0 \\ 0\,1\,1\,0\,1\,1\,0 \\ 1\,0\,0\,0\,1\,1\,1 \\ 1\,0\,1\,1\,0\,1\,0 \\ 1\,1\,0\,1\,1\,0\,0 \\ 1\,1\,1\,0\,0\,0\,1 \end{bmatrix}$$.

Die Hammingdistanz des Codes beträgt 4 (alle Codeworte != Null haben gleiches Gewicht). Dieser Code ist kombinatorisch äquivalent einem **MACDONALD (7,3)** - Code [Pet67].

A23	A22	A21	A20	A19	A18	A17	A16	HEX	SBC
1	0	0	1	1	1	0	1	9D	1
1	0	1	0	1	0	1	1	AB	2
1	0	1	1	0	1	1	0	B6	3
1	1	0	0	0	1	1	1	C7	4
1	1	0	1	1	0	1	0	DA	5
1	1	1	0	1	1	0	0	EC	6
1	1	1	1	0	0	0	1	F1	7

Abb. 9 - 2: Globale Mailboxbasisadressen

9.1.4 Lokales Adreß-Mapping

Innerhalb des 64KByte- Globalmailbox-Bereiches sind wieder N (minimal 7) lokale Mailboxen unterzubringen.

Festlegung: Die Mailboxgröße beträgt 1 KByte (10Bit) => n=6;

Gesucht: (6,3) - Code mit maximalem Hammingabstand.

Ein (6,3) - Code mit optimalem Hammingabstand dmax = 3 geht aus dem obigen (7,3) - Code durch Streichung einer beliebigen Spalte (z.B. der ersten) hervor [Pet67]:

$$(\mathbf{W}_{64k}) = \begin{bmatrix} 0\,0\,0\,0\,0\,0 \\ 0\,1\,1\,1\,0\,1 \\ 1\,0\,0\,1\,1\,0 \\ 1\,1\,0\,1\,1\,0 \\ 0\,0\,0\,1\,1\,1 \\ 0\,1\,1\,0\,1\,0 \\ 1\,0\,1\,1\,0\,0 \\ 1\,1\,0\,0\,0\,1 \end{bmatrix}$$

A15	A14	A13	A12	A11	A10	HEX	SBC
0	0	0	1	1	1	1C	1
0	1	1	0	1	0	68	2
0	1	1	1	0	1	74	3
1	0	1	0	1	1	AC	4
1	0	1	1	0	0	B0	5
1	1	0	0		1	C4	6
1	1	0	1	1	0	D8	7

Abb. 9 - 3: Lokale Mailbox-Basisadressen