Verlässliche, adaptive Informationssysteme - Zuverlässigkeit und QoS

Markus Edinger

Matrikelnummer: 334520
Email: medinger@student.uni-kl.de

Betreuer: Boris Stumm

Inhalt. Die Verwendung verteilter Systeme hat in den letzten Jahren immer mehr an Bedeutung gewonnen, es soll nun aber trotzdem versucht werden, Systeme zu entwickeln, in deren Sicherheit Vertrauen gesetzt werden kann. Außerdem sollen auch Attribute der Zuverlässigkeit, wie Verfügbarkeit, oder Funktionsfähigkeit gewährleistet werden. Diese Ausarbeitung beschäftigt sich mit der Zuverlässigkeit von Informationssystemen und deren Klassifizierung. Darüber hinaus werden Möglichkeiten vorgestellt Fehler in Klassen zu unterteilen. Es wird ein Einblick in die Gewährleistung von Leistungsmerkmalen in einem verteilten System durch Quality of Service (QoS) gegeben. Die Beurteilung und der Vergleich der Zuverlässigkeit verschiedener Systeme wird im Abschnitt Dependability Benchmarking erläutert.

Inhaltsverzeichnis

1 - Einleitung

Bei der Entwicklung von Informationssystemen muss ein großes Augenmerk auf die Zuverlässigkeit gelegt werden. Neben der korrekten Funktion sind auch Eigenschaften wie Verfügbarkeit und Sicherheit von großer Bedeutung. Als Überbegriff für diese Attribute eines Systems wurde der Ausdruck Zuverlässigkeit geprägt. Um diese gewährleisten zu können muss man die Bedrohungen eines Systems kennen. Wobei es sich hierbei nicht nur um bösartige Angriffe wie Viren oder Würmer handeln kann, sondern Hardwarefehler und Benutzerfehler müssen auch schon während der Entwicklung eines Systems behandelbar sein. Aus diesem Grund erstellte man Fehlerklassen um spezielle Strategien zur Behandlung der verschiedenen Fehler entwickeln zu können.

Neben der Zuverlässigkeit stellt auch die Qualität der Leistung eine Anforderung an das Informationssystem. Trotz der Verteilung eines Systems versucht man eine gewisse Qualität der Leistungen zu garantieren. Diese Verfahren nennt man „Quality of Service“.

Um Abschätzen zu können wie gut ein System die geforderten Zuverlässigkeitsmerkmale unterstütz, wird der Begriff des Benchmarking auf den Bereich der Zuverlässigkeit erweitert. Hier werden Methoden entwickelt, Maßeinheiten für die Zuverlässigkeit eines Systems zu erhalten und es dadurch mit anderen Systemen vergleichbar zu machen.

2 - Zuverlässigkeit

Für den Begriff Zuverlässigkeit gibt es einige Definitionen Dies liegt an der Tatsache, dass für jeden Entwickler und Benutzer eines Informationssystems Zuverlässigkeit eine etwas andere Bedeutung hat. Es gibt verschiedene Attribute der Zuverlässigkeit, die für unterschiedliche System verschieden wichtig sind. Aus diesem Grund wird hier eine sehr allgemein Definition gegeben:

Zuverlässigkeit (dependability) ist die Vertrauenswürdigkeit eines Systems, so dass Vertrauen in den Dienst den es erbringt, gesetzt werden kann, wobei *Dienst* (service) das Verhalten des Systems darstellt, wie es vom Benutzer beobachtet wird. Der *Benutzer* (user) wiederum ist ein System, menschlich oder physikalisch, das mit einem vorangehenden System zusammenwirkt. Ein Benutzer nimmt also den Dienst eines anderen Systems in Anspruch. Dieses andere System kann auch die Leistung des ersten Benutzers in Anspruch nehmen, wodurch auch das zweite System zum Benutzer wird.

Wir unterscheiden bei einem System seine *Funktion* (function) und sein *Verhalten* (behavior). Die Funktion ist die Aufgabe des Systems, wie sie in der Spezifikation beschrieben wird. Das Verhalten beschreibt, wie das System implementiert ist und was es wirklich tut.

Der *Gesamtzustand* eines Systems umfasst die Hardware, Software, die gespeicherten Informationen und die Verbindungen innerhalb des kompletten Systems.

Der *externe Zustand* ist der Teil des Gesamtzustandes, der für einen Benutzer sichtbar ist, also auch die Leistungen, die von einem System erbracht werden. Alles andere bezeichnet man als *internen Zustand* [Lapr92, ALLR03].

Abbildung 1.1 gibt einen Überblick über die Begriffe, die in diesem Kapitel behandelt werden und ihren Zusammenhang mit dem Thema Zuverlässigkeit.

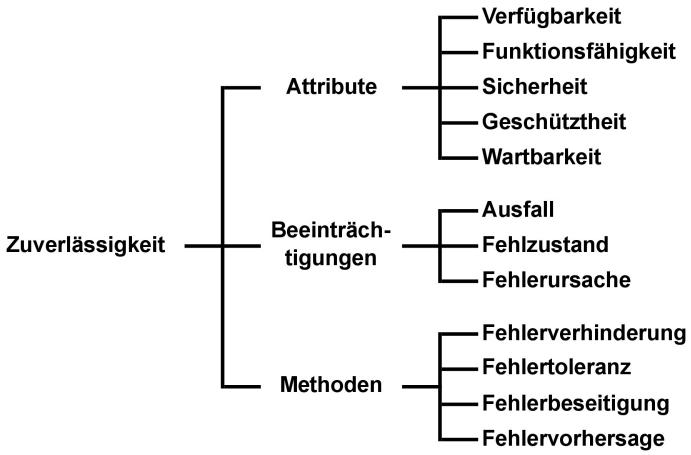

Abb. 2.1 Zuverlässigkeitsbaum

2.1 Attribute der Zuverlässigkeit

Um Zuverlässigkeit richtig beschreiben zu können, muss man sich überlegen, was man von einem zuverlässigen System erwartet. Nach [ALLR03] gibt es fünf verschiedene Anforderungen an ein System, die alle unter dem Oberbegriff Zuverlässigkeit zusammengefasst werden können.

- **Verfügbarkeit (Availability)** - *Bereitsein zum Gebrauch*
  Es wird von einem System erwartet, dass es zu einem großen Maße verfügbar ist. Die Ausfallzeiten müssen also sehr gering sein. Jim Gray hat diese wünschenswerte Eigenschaft eines Systems mit seinen „eight nines of availability“ (99,999999%) ausgedrückt. Das bedeutet, ein System darf in hundert Jahren nur eine Sekunde ausfallen. Eine solche Anforderung ist keineswegs reell, sie stellt eher ein Ideal dar, dem man sich so gut es geht annähern will [Härd05].

  Verfügbarkeit ist ein quantitatives Maß. Es beschreibt die Wahrscheinlichkeit, dass man das System in einem funktionierenden, korrekt arbeitenden Zustand vorfindet.
- **Funktionsfähigkeit (Reliability)** - *Kontinuität der Leistung*
  Die Funktionsfähigkeit ist definiert als die Wahrscheinlichkeit, dass ein fehlerfreies System bis zur Beendigung seines Dienstes auch fehlerfrei bleibt. Das heißt, dass man unter Voraussetzung eines fehlerfreien Anfangszustandes des Systems auch ein korrektes Ergebnis seines Dienstes erwarten kann.

- **Sicherheit (Safety)**[1] - *Vermeidung katastrophaler Folgen für die Umwelt*
  Die Sicherheit eines Systems ist die Eigenschaft, ungewollte, schwerwiegende Folgen für das System (Daten und Komponenten) und den Benutzer verhindern zu können. Beispielsweise muss verhindert werden, dass durch einen Benutzerfehler wichtige Daten permanent gelöscht werden.
- **Geschütztheit (Security)** - *Verhindern von nicht autorisiertem Zugriff/ nicht autorisierte Handhabung von Information*
  Die Geschütztheit beschreibt die Eigenschaft eines Systems, unberechtigten Zugriff auf Komponenten, Daten oder Information zu verwehren und bösartige Angriffe auf das System zu erkennen und zu unterbinden.
- **Wartbarkeit (maintainability) –** *Möglichkeit, das System Änderungen zu unterziehen*
  Unter Wartbarkeit versteht man, dass ein System an veränderte Spezifikationen angepasst, Fehler entfernt und Verbesserungen eingebaut werden können. Es gibt Wartungen, die während des Betriebs erfolgen, aber auch welche, für die das System (oder zumindest Komponenten davon) gestoppt werden müssen. Darunter leidet die Verfügbarkeit.

2.2 Beeinträchtigungen der Zuverlässigkeit

Man nennt ein System korrekt, wenn es die Leistung erbringt, die für das System spezifiziert wurde. Sobald das System diese Spezifikationen nicht mehr erfüllt geht es in einen fehlerhaften Zustand über.[2]

- **Ausfall (Failure)**

Ein Ausfall tritt auf, wenn die Leistung des Systems nicht mehr der korrekten, erwarteten Leistung entspricht. Es ist der Übergang von einem korrekten in einen inkorrekten Systemzustand. Die Zeit, in der das System inkorrekte Dienste liefert nennt man *Dienstausfall* (service outage). Den Übergang zu korrektem Service nennt man *Dienstwiederherstellung* (service restauration).

Ein Ausfall kann nach verschiedenen Gesichtspunkten charakterisiert werden.

– *Art des Ausfalls.* Wird ein falscher Inhalt geliefert, oder wird der Dienst zum falschen Zeitpunkt geliefert (zu früh oder zu spät)? Es gibt auch Ausfälle, die sowohl falschen Inhalt liefern und ein Timing-Problem haben. Man unterscheidet hier den *halt failure* (das System stoppt und geht in einen konstanten Zustand über), und ei-

[1] In der englischsprachige Literatur findet man die Begriffe Safety und Security. Das deutsche Wort Sicherheit würde für beide Begriffe eine gültige Übersetzung darstellen. Um die beiden Begriffe unterscheiden zu können wird das Wort Sicherheit für Safety und Geschütztheit für Security verwendet[Lapr92].

[2] In der englischsprachigen Diskussion werden die Begriffe Failure, Fault und Error verwendet. Im Deutschen gibt es keine Wörter, die die gleichen Bedeutungen besitzen. Im Bereich der Zuverlässigkeit haben sich die Begriffe Ausfall für Failure, Fehlerursache für Fault und Fehlzustand für Error eingebürgert [Lapr92]. Wenn aus dem Zusammenhang klar ist, ob es sich um einen Fault oder Error handelt, wird nur das Wort Fehler benutzt. Dies ist der Fall, wenn zusammengesetzte Wörter benutzt werden, wie z.B. Fehlertoleranz, wobei in diesem Fall Fehler eigentlich Fehlerursache bedeutet.

nen *sprunghaften Ausfall (erratic failure)*, in diesem Fall entsteht ein alternierender Systemzustand.
- *Erkennbarkeit*. Ist der Ausfall zu erkennen, so dass man darauf reagieren kann, oder wird ohne Fehlermeldung weitergearbeitet?
- *Konsistenz*. Wird von allen Benutzer und zu jeder Zeit der gleiche Ausfall erzeugt, oder werden immer unterschiedliche Dienste geliefert? Inkonsistente Ausfälle nennt man auch *byzantinische Ausfälle*. Dieser Name geht auf eine Schlacht der byzantinischen Armee zurück, in der einige Generäle Verräter waren und falsche Informationen übermittelten [LaPS82].
- *Konsequenzen*. Worauf wirkt sich der Fehler aus? Ist eine Beeinträchtigung der Verfügbarkeit zu erwarten, leidet die Sicherheit oder die Geschütztheit unter dem Fehler oder wird lediglich die Performance gemindert? Auf Grund der Konsequenzen ist es möglich einen Ausfall zu klassifizieren. Diese Klassifizierung nennt man die *Schwere* (severity) eines Fehlers. Die Skala der Schwere ist flexibel und geht von geringfügig (minor) bis katastrophal (catastrophic).

- **Fehlzustand (Error)**

Ein Fehlzustand ist der Teil des Systemzustandes, der für einen Ausfall verantwortlich ist. Ein Fehlzustand kann erkannt werden durch eine Fehlermeldung oder Fehlerlogging. Bleibt ein Fehlzustand unentdeckt so nennt man ihn *latent*. Ob ein Fehlzustand zu einem Ausfall führt hängt davon ab, ob er Teil des externen Zustandes ist, das heißt für einen Benutzer sichtbar oder nicht.

Trotz eines externen Fehlzustandes ist das Auftreten eines Ausfalls nicht zwingend notwendig. Zum Beispiel kann der Fehler zu dem Zeitpunkt an dem ein Dienst aufgerufen wird schon wieder überschrieben sein, oder er kann einen korrekten Dienst liefern, indem auf redundante Daten im System zurückgegriffen wird.

- **Fehlerursache (Fault)**

Hierbei handelt es sich um die verantwortliche oder hypothetische Ursache für einen Fehlzustand.

Die verschiedenen Arten der Fehlerursachen sind sehr weitreichend. Defekte Festplatten oder Speicherbausteine können Fehlerursachen sein, aber auch Programmfehler oder Sicherheitslücken, die den Befall von Viren oder Würmern ermöglichen. Auch Dienstausfälle einer anderen Komponente können Ursache für einen neuen Fehler sein. Eine detaillierte Aufstellung folgt in Abschnitt 3.

- **Pathologie von Ausfall, Fehlerursache und Fehlzustand**

Wie ein Fehler entsteht und wie er sich im System fortpflanzt und Quelle neuer Fehler sein kann wird in Abbildung 2.2 gezeigt und anschließend näher erläutert.

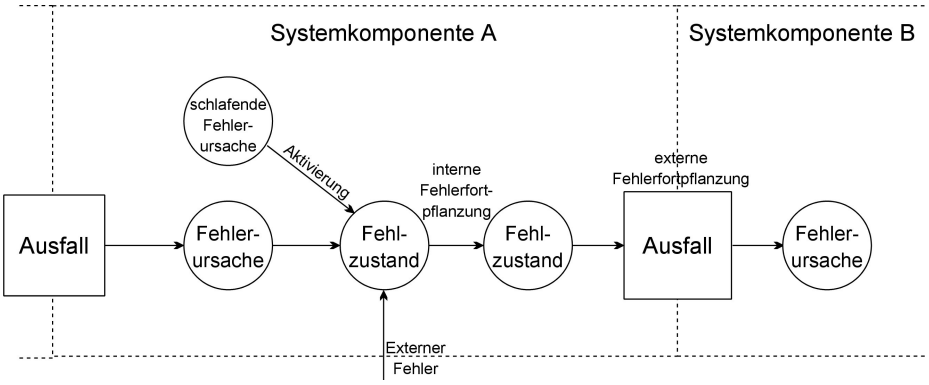

Abb. 2.2 Pathologie von Ausfall, Fehlerursache und Fehlzustand [ALLR03]

1. *Fehlerursache*: Eine Fehlerursache kann schlafend sein, d.h. inaktiv, solange sie keinen Fehlzustand im System erzeugt. Sobald eine Fehlerursache aktiviert wird, z.B. durch einen bestimmten Systemaufruf oder das Verwenden einer defekten Hardwarekomponente, geht das System in einen Fehlzustand über.
2. *Fehlerfortpflanzung*: Es gibt die *interne Fehlerfortpflanzung*, die innerhalb einer Systemkomponente geschieht und die *externe Fehlerfortpflanzung*. Sie tritt auf, wenn der Fehler die Schnittstelle zu einer anderen Komponente erreicht. Zu diesem Zeitpunkt wird der Dienst, der an Komponente B geliefert wird, falsch, es tritt also ein Ausfall auf. Außerdem entsteht eine externe Fehlerursache in Komponente B, diese kann weitere Fehlerzustände erzeugen.
3. *Dienstausfall*: Er tritt auf, wenn die Fehlerfortpflanzung bis zu einer Schnittstelle fortgeführt wird. Dieser Ausfall ist eine neue interne Fehlerursache für Komponente A und eine externe Fehlerursache für Komponente B.

2.3 Sicherstellung der Zuverlässigkeit

Als Methoden oder Mittel der Zuverlässigkeit bezeichnet man *Zuverlässigkeitsverfahren* (dependability procurement) und *Zuverlässigkeitsvalidation* (dependability validation).

An Zuverlässigkeitsverfahren gibt es zwei Arten, die Fehlerverhinderung und die Fehlertoleranz. Fehlerbeseitigung und Fehlervorhersage zählt man zur Gruppe der Zuverlässigkeitsvalidationen [ALLR03].

- **Fehlerverhinderung (Fault prevention)**

Unter Fehlerverhinderung versteht man Methoden, die das Auftreten eines Fehlzustandes oder das Einführen einer Fehlerursache ins System verhindern sollen. Diese Methoden treten bei der Entwicklung des Systems und zur Laufzeit des Systems in Kraft. Es gibt klassische Methoden der Fehlerverhinderung bei der Softwareentwicklung, wie Sicherheitsregeln oder intelligente Programmiersprachen, die Fehler schon bei der Eingabe erkennen können. Im Hardwarebereich gibt es vordefinierte Designregeln, die Fehler zu vermeiden helfen. Später werden Fehlerklassen eingeführt für die jeweils die passenden Fehlerverhinderungsstrategien entwickelt werden können.

- **Fehlertoleranz (Fault tolerance)**

Bei der Fehlertoleranz werden Methoden entwickelt, korrekte Leistungen zu erbringen, obwohl ein Fehler im System ist. Das heißt, es wird ein Ausfall vermieden, obwohl eine Fehlerursache vorhanden ist. Die Fehlertoleranz besteht aus zwei wichtigen Schritten:

– Erstens die *Fehlererkennung*. Sie kann *nebenläufig* erfolgen, dass heißt, während des normalen Betriebs untersucht das System die Korrektheit der Leistungen und leitet nach dem Erkennen eines Ausfalls die nötigen Maßnahmen ein. Fehlererkennung kann auch *präventiv* erfolgen, also während der normale Dienst ausgesetzt ist. Hier können auch latente Fehlzustände und schlafende Fehlerursachen erkannt werden.
– Der zweite Schritt ist die *Systemwiederherstellung*. Er gliedert sich in zwei Teile. Als erstes wird das System wieder in einen fehlerfreien Zustand gebracht, diesen Schritt nennt man *Fehlzustandsbehandlung*. Dies kann durch ein *Rollback* geschen. Dabei wird das System wieder in einen alten Systemzustand gebracht der als richtig bekannt war, bevor der Fehler aufgetreten ist. Die aktuellen Transaktionen, die vom Fehler betroffen sind werden abgebrochen. Für diese Methode müssen Rücksetzpunkte vorhanden sein, in denen der jeweils aktuelle Zustand gesichert wird, um ihn später gegebenenfalls wiederherstellen zu können. Die zweite Möglichkeit ist ein *Rollforward*, hier wird ein neuer Systemzustand ohne Fehler erzeugt, wobei der Dienst, bei dem der Fehler aufgetreten war nicht zu Ende geführt wird. Bei diesen beiden Möglichkeiten wird an den Benutzer eine Fehlermeldung gesendet, dass der Dienst fehlgeschlagen ist. Die dritte Möglichkeit ist die *Fehlermaskierung*, hierbei wird versucht einen korrekten Dienst zu liefern, obwohl ein Fehler erkannt wurde. Dies ist nur bei ausreichender Redundanz möglich. Der Benutzer wird von der fehlerhaften Situation nichts mitbekommen und nur der Administrator wird informiert, um weitergehende Schritte einzuleiten.

  Danach wird versucht, den Fehler aus dem System zu entfernen, diesen Schritt nennt man *Fehlerursachenbehandlung*. Dieser Prozess verläuft in vier Schritten:
  – *Diagnose*. Identifizierung und Aufzeichnung von Fehlerursachen. Man bestimmt die Art des Fehlers und die Komponente des Systems, in der der Fehler auftrat. Dies ist wichtig um den Fehler behandeln zu können.
  – *Isolation*. Hier wird die fehlerhafte Komponente (entweder eine physikalische Komponente wie eine Festplatte oder defekter Speicher im Hardwarefall oder eine Funktion etc. im Softwarefall) aus dem System genommen. Der Fehler wird in einen schlafenden Zustand versetzt.
  – *Rekonfiguration*. Es werden Ersatzkomponenten hinzugeschaltet oder Dienste und Programme werden auf andere, funktionierende, Systemkomponenten verteilt.
  – *Reinitialisierung*. Die neue Systemkonfiguration wird überprüft, eventuell werden Updates für hinzugeschaltete Komponenten ausgeführt und die neue Konfiguration wird gesichert.

- **Fehlerbeseitigung (Fault removal)**

Hierbei wird versucht, die Anzahl und die Schwere der im System vorhandenen Fehler zu reduzieren. Man unterscheidet die Fehlerbeseitigung während der Entwicklung und die Fehlerbeseitigung während des Betriebs.

Die Fehlerbeseitigung während der Entwicklung durchläuft drei Phasen:

- *Verifikation*. Es wird überprüft, ob das System die geforderten Eigenschaften erfüllt. Man unterscheidet zwei Arten der Verifikation, erstens die *statische Verifikation*, bei der eine theoretische Analyse des Systems vollzogen wird, anhand von statischen Analysen, wie Komplexitätsanalyse, Codeanalyse, oder modellbezogenen Analysen wie z.B. repräsentierenden Petrinetzen. Eine weitere Möglichkeit bietet die *dynamische Verifikation*, bei der das System Tests unterzogen wird um die gewünschten Systemeigenschaften zu überprüfen. Falls die Eigenschaften nicht erfüllt werden, müssen die folgenden beiden Schritte durchlaufen werden.
- *Diagnose und Korrektur*. Es ist nun nötig die Fehlerursachen ausfindig zu machen, die dafür verantwortlich sind, dass die Systemspezifikationen nicht erfüllt werden konnten. Diese müssen korrigiert werden. Nach der Korrektur ist eine erneute Verifikation notwendig, um ausschließen zu können, dass die Korrektur schädliche Nebeneffekte für das System hatte.

Während des Betriebs kann die Fehlerbeseitigung *korrigierend* sein, das heißt, ein Fehler ist aufgetreten und muss entfernt werden. Im schlechtesten Fall muss hierfür das System oder eines seiner Komponenten aus dem Betrieb genommen werden, was einen Verlust an Verfügbarkeit zu Folge hat.

Die zweite Art ist die *präventive* Fehlerbeseitigung. Eine Fehlerursache wird beseitigt, obwohl noch kein Ausfall zu verzeichnen war. Dies ist der Fall, wenn veraltete Hardware ausgetauscht wird, oder wenn bei einem anderen System mit der gleichen Software Ausfälle zu verzeichnen waren.

- **Fehlervorhersage (Fault forecast)**

Mit diesen Methoden versucht man die Zahl der aktuell im System vorhandene Fehler abzuschätzen und das künftige Auftreten von Fehlern und der Folgen vorherzusagen. Bei qualitativen Abschätzungen wird versucht, die Schwere der Fehler und deren Konsequenzen einzuordnen, so dass es möglich wird, Fehlerklassen zu erstellen um auf diese reagieren zu können. Die quantitative Abschätzung gibt Maßeinheiten für die Attribute der Zuverlässigkeit. Hier kann z.B. ein Maß für die Verfügbarkeit berechnet werden oder für die Fehleranfälligkeit eines Systems. Ein großes Augenmerk gilt hierbei auch der Performance des Systems, im fehlerfreien Fall und wenn Fehler aufgetreten sind. Ein Mittel für solche Abschätzungen sind z.B. Benchmarks, bei denen verschiedene Systeme durch ein standardisiertes Verfahren verglichen werden können und mit dem auch Zahlenwerte für die interessanten Maßeinheiten der Zuverlässigkeit bestimmt werden können.

2.4 Entwicklung zuverlässiger Systeme

Unser Ziel ist es ein zuverlässiges System zu entwickeln, das nach [Härd05] die folgenden Attribute besitzt:

- *Problemfrei* - Ein System soll potentiell von mehreren Millionen genutzt werden, der Wartungsaufwand soll aber von einem Administrator erledigt werden können.
- *Geschützt* - Das System darf nur autorisierten Personen zugänglich sein, seine Leistungen können nicht durch nicht autorisierte Personen beeinträchtigt werden und die Informationen müssen sicher sein.

– *Verfügbar* - Das System ist höchstens für eine Sekunde in hundert Jahren nicht erreichbar.

Diese Vorgaben sind als ein Ideal anzusehen und das Ziel ist es, diesem Ideal möglichst nahe zu kommen. In der Welt der modernen Informationssysteme muss ein solches System mit den folgenden Anforderungen zurechtkommen:

– Hohe Verteilung mit großer räumlicher Trennung.
– Offenheit der Systemstruktur und Autonomie der einzelnen Systemkomponenten.
– Heterogenität von Struktur und Daten der einzelnen Komponenten.
– Sehr lange laufende interaktive Prozesse.

Die Schwierigkeiten, die ein verteiltes System mit sich bringt sind vielfältig. Die einzelnen Komponenten sind sehr schwer zu kontrollieren. Die Fehlerquellen sind vielfältig, weit verteilt und sehr schwer ausfindig zu machen. Daher ist es notwendig, schon bei der Entwicklung ein großes Augenmerk auf die Sicherheit und die Wartbarkeit eines solchen Systems zu legen. Ein Ansatz zur Entwicklung zuverlässiger Systeme sieht drei Grundaspekte vor:

– Die Systemstruktur muss in granulare Komponenten aufgeteilt sein.
– Diese Komponenten sollen beschränkte und vergleichbar einfache Funktionen haben.
– Die Schnittstellen zwischen den Komponenten müssen kontrollierbar sein und die Kommunikation zwischen den Komponenten soll so gering wie möglich sein.

3 - Fehlermodelle

Man betrachtet Fehlerursachen[3] in der Regel aus acht unterschiedlichen Gesichtspunkten, die es ermöglichen einen Fehler einzuordnen und zu kategorisieren. In diesem Abschnitt werden diese Gesichtspunkte dargestellt und erläutert. Später werden Fehler auf Grundlage dieser Gesichtspunkte analysiert und Fehlerklassen werden vorgestellt. Diese Fehlerursachen gliedern sich in drei Hauptkriterien:

3.1 Fehlerursachen

- **Art der Fehlerursache (nature):**

– *Unvermögen (capability):*
  Hier unterscheidet man, ob eine Fehlerursache versehentlich (accidental fault) oder auf Grund von Inkompetenz (Incompetence Fault) in das System gelangt ist.
– *Vorsatz (intent):*
  Hierbei wird beobachtet, ob der Fehler durch den Entwickler unbewusst (non-deliberate fault) oder bewusst (deliberate fault) produziert wurde. Bewusste Fehler entstehen, wenn eine Fehlerursache in Kauf genommen wird, weil dadurch Vorteile für eine andere Systemeigenschaft entstehen.
– *Absicht (objective):*

[3] Wenn in diesem Kapitel von Fehler die Rede ist, so handelt es sich i.d.R. um Fehlerursachen

Diese Unterscheidung betrifft die Zielsetzung des Verursachers, ob er mutwillig einen Fehler in das System eingeschleust hat, um bösartigen Schaden anzurichten (malicious fault), oder ob es ein Fehler ohne bösartigen Vorsatz war (non-malicious falut). Die Verhinderung von bösartigen Fehlern gewinnt zunehmend an Bedeutung; es vergeht kaum ein Tag, an dem kein neuer Virus, Wurm oder Trojaner bekannt wird.

- **Ursprung der Fehlerursache (origin):**

– *Phänomenologische Gründe (Phenomenological causes):*
Hier wird unterschieden, ob der Fehler physikalischen bzw. natürlichen Ursprungs ist (natural fault), oder ob es sich um einen menschlichen Fehler handelt (human-made fault).

– *Systemgrenzen (System boundaries):*
Handelt es sich um einen internen Fehler (internal fault) oder um einen externen Fehler, also einen Fehler, der durch die physikalische oder menschliche Umgebung beeinflusst wird (external fault).

– *Phase der Entstehung (phase of creation):*
Man unterscheidet Entwurfsfehler (development faults), die entweder während der Entwicklung oder während der Modifikation (z.B. Wartung) des Systems entstehen und Benutzungsfehler (operational faults), der während der Benutzungsphase (use phase) entstehen können.

– *Dimension (dimension):*
Hier wird zwischen Hardwarefehlern (hardware faults) und Softwarefehlern (software faults) unterschieden.

- **Dauer (duration):**

– *Hartnäckigkeit (persistence):*
Die Unterscheidung zwischen andauernden Fehlern (permanent faults) und zeitlich begrenzten Fehlern (transient faults).

3.2 Fehlerklassen

Jede Fehlerursache kann jeweils nach diesen acht Kriterien kategorisiert werden. Aus der Kombination dieser acht Kriterien erhält man 256 Fehlerklassen, von denen nach [ALLR03] aber nur 31 relevant sind, da sich manche Kriterien gegenseitig ausschließen, z.B. kann ein bösartiger Fehler nicht unabsichtlich in das System gelangt sein. Ein natürlicher Fehler kann nicht nach der Absicht oder dem Vorsatz beurteilt werden. Eine Aufstellung der 31 relevanten Fehlerklassen wird in Abbildung 3.1 dargestellt.

Es gibt hierbei drei Hauptkategorien für Fehlerklassen:

– *Entwicklungsfehler:* Hier sind alle Fehlerklassen enthalten, bei denen die Fehler während der Entwicklungsphase entstanden sind.
– *Physikalische Fehler:* Beinhaltet alle Fehlerklassen, die die Hardware betreffen.
– *Eingabefehler:* Hierzu gehören alle externen Fehler.

Wir wollen uns nun einige dieser Klassen genauer anschauen. Die in Klammer angegebenen Nummern beziehen sich auf die jeweilige Fehlerklasse nach Abb.2.1.

- *Softwarefehler(1-4)*: Die Fehler dieser Klasse entstanden während der Entwicklungsphase und haben ihren Ursprung auf der Softwareebene.
- *Logische Bomben (5, 6)*: Logische Bomben bergen ein Sicherheitsrisiko, da sie die Sicherheit des Systems gefährden. Sie sind mit Vorsatz schon während der Entwicklungsphase entstanden.
- *Fehlerhafte Hardware (7-10)*: Diese Fehlerursachen entstehen während der Entwicklungsphase, da falsche Hardwarekomponenten oder Konfigurationen verwendet werden.
- *Produktionsfehler (7-11)*: Die benutzten Komponenten sind schon seit der Produktion fehlerhaft. Diese Fehler können auch natürliche Fehlerursachen haben.
- *Verschleiß (12,13)*: Hardwarekomponenten können mit der Zeit an Qualität verlieren und somit zur Fehlerursache werden.
- *Interferenz (14-21)*: Hardwarekomponenten, die ihre Leistungen gegenseitig beeinträchtigen verursachen Fehler, die in diese Klassen eingeordnet werden.
- *Eindringungsversuche (22-24)*: Nichtautorisierte Eingriffe in das System, gefährden die Integrität des Systems.
- *Viren und Würmer (25)*: Sicherheitslücken im System können das Eindringen von Viren und Würmern ermöglichen und gefährden das gesamte System.
- *Eingabefehler (26-31)*: Dies sind alle nicht bösartigen menschlichen Fehlerursachen, die während der Betriebsphase auf Softwareebene entstehen.

	Ursprung								Art						Dauer			
	Phase der Entstehung		System-grenzen		Phänomen-ologische Ursache		Dimension		Absicht		Voratz		Unvermögen					
Fehlerklasse	Entwicklungs-fehler	Benutzungs-fehler	Interne Fehler	Externe Fehler	Physikalische Fehler	Menschliche Fehler	Hardware-fehler	Software-fehler	Nicht bösartig	bösartig	unbeabsichtigt	beabsichtigt	Versehen	Inkompetenz	permanente Fehler	transiente Fehler		
1	X		X			X		X	X		X		X		X		Entwicklungsfehler	Physikalische Fehler
2	X		X			X		X	X		X			X	X			
3	X		X			X		X	X			X	X		X			
4	X		X			X		X	X			X		X	X			
5	X		X			X		X		X		X			X			
6	X		X			X	X			X		X			X			
7	X		X			X	X		X		X		X		X			
8	X		X			X	X		X		X			X	X			
9	X		X			X	X		X			X	X		X			
10	X		X			X	X		X			X		X	X			
11	X		X		X		X		X		X		X		X			
12		X	X		X		X		X		X		X		X			
13		X	X		X		X		X		X		X			X		
14		X		X	X		X		X		X		X		X			
15		X		X	X		X		X		X		X			X		
16		X		X		X	X		X		X		X			X	Eingabefehler	
17		X		X		X	X		X		X			X	X			
18		X		X		X	X		X		X			X		X		
19		X		X		X	X		X			X	X			X		
20		X		X		X	X		X			X		X	X			
21		X		X		X	X		X			X		X		X		
22		X		X		X	X			X		X			X			
23		X		X		X	X			X		X				X		
24		X		X		X		X		X		X				X		
25		X		X		X		X		X		X			X			
26		X		X		X		X	X		X		X			X		
27		X		X		X		X	X		X			X	X			
28		X		X		X		X	X		X			X		X		
29		X		X		X		X	X			X	X			X		
30		X		X		X		X	X			X		X	X			
31		X		X		X		X	X			X		X		X		

Abb. 3.1 Fehlerklassen nach[ALLR03]

4 - Quality of Service in verteilten Systemen

Wenn man verteilte Systeme entwickelt, stößt man schnell auf Probleme, die durch die Verteilung auftreten. Teilausfälle von Systemkomponenten, Schwankungen der Übertragungsrate, usw. müssen behandelt werden können. Durch diese Probleme wurde es nötig eine Architektur zu entwickeln, die mit den Problemen der verteilten Systeme umgehen kann. Dafür wurde der Begriff „*Quality of Service*“ (QoS) oder auf deutsch *Dienstgüte*[4] geprägt. QoS umfasst also nichtfunktionale Eigenschaften eines Dienstes wie z.B. Performance.

Es gibt unterschiedliche Definitionen von QoS; im Siemens Online Lexikon [Siem06] findet man die folgende Definition:

„Dienstgüte, quality of service (QoS):

Unter Dienstgüte (QoS) versteht man alle Verfahren, die den Datenfluss in LANs und WANs so beeinflussen, dass der Dienst mit einer festgelegten Qualität beim Empfänger ankommt. Es handelt sich also um die Charakterisierung eines Dienstes, der für den Nutzer unmittelbar sichtbar ist und dessen Qualität er messen kann. Technisch handelt es sich um eine Parametrisierung von Protokollen zur Bestimmung des Übertragungsverhaltens für bestimmte Dienste.“

Eine allgemeinere Definiton wurde von der International Telecommunication Union (ITU) [ITU97] gegeben:

„Quality of Service (QOS): A set of qualities related to the collective behaviour of one ore more objects.“

Die Qos-Entwicklung wurde notwendig, da Anwendungen, die einen hohen Grad an Systemressourcen in Anspruch nehmen immer mehr an Bedeutung gewinnen. Multimedia-Anwendungen brauchen eine hohe Bandbreite für die Video- und Audioübertragung. Ein weiteres Feld ist die Fehlertoleranz, hier werden redundante Datenquellen genutzt und es wird ein Mechanismus benötigt, die Steuerung dieser Dienste zu koordinieren.

Es werden einige Anforderungen, an ein verteiltes System gestellt [GTUW04]:

- Umgang mit der *Heterogenität*: In einem verteilten System treffen viele Komponente unterschiedlicher Art aufeinander. Sowohl Hardware als auch Software unterschiedlichster Art wird in einem verteilten System integriert. Ein QoS-System muss die Handhabung aller im System vorkommenden Komponenten gewährleisten können.
- Ein weiterer Schwerpunkt ist die *Adaptivität*: Das System sollte selbstständig in der Lage sein, die vorhandenen Ressourcen so einzusetzen, dass der Benutzer das optimale Ergebnis erzielt. Es ist also notwendig, dass sich das System zur Laufzeit an Veränderungen der Systemstruktur anpasst.
- Das System benötigt eine gewisse *Robustheit*, d.h. es muss starken Belastungen des Netzwerks vorbeugen und Ausfälle in ihrer Zahl und Schwere minimieren. Außerdem sollte es gegen nicht autorisierte Manipulationen gesichert sein.

[4] Der Begriff Dienstgüte hat sich in der Literatur nicht etabliert, darum wird hier der englische Begriff Quality of Service benutzt.

4.1 Begriffsbildung

QoS-Management ist der Überbegriff der Leistungen eines Systems, die nötig sind um QoS-Eigenschaften des Systems überwachen, kontrollieren und steuern zu können.

Alle Anforderungen, die ein Nutzer an das System stellt, werden quantifiziert und als *QoS-Anforderungen* zusammengefasst.

QoS-Charkateristik ist ein Aspekt von QoS eines Systems, Dienstes oder Ressource, der identifiziert und quantifiziert werden kann. Ein Beispiel für eine QoS-Charakteristik ist die zugesicherte Übertragungsrate eines Netzwerkes. QoS-Anforderungen werden spezifiziert und konkret messbaren Werten zugeordnet. Beispiele dafür sind Durchsatz, Delay oder Jitter (Delay variation).

QoS-Parameter sind Werte, die QoS-Charakteristiken beschreiben oder abgrenzen. Zum Beispiel eine maximale Zeitdauer, die die Übertragungsverzögerung nicht überschreiten darf oder eine minimale Bandbreite, die von der Anwendung erwartet wird.

Eine *QoS-Kategorie* ist die Klasse in die die Anforderungen des Nutzers oder der Applikation einzuordnen sind. Beispielsweise Echtzeit, Multimedia, Verfügbarkeit, Sicherheit usw. [OMG97].

4.2 Dimensionen von QoS

Ein QoS-System zu spezifizieren ist sind sehr komplex und nach [OMG97] ergeben sich die verschiedensten Arten von Problemstellungen. Drei dieser Dimension werden hier erläutert.

- **Statische bzw. dynamische QoS**

Beim *statischen QoS* werden die Leistungsansprüche zur Entwicklungszeit festgelegt. Hier wird versucht die für den Benutzer maximale Leistung schon während der Entwurfsphase zu erreichen. Es wird durch Softwarelösungen wie Scheduling-Strategien oder proritätsgesteuerte Tasks und durch die richtige Hardwarekonfiguration, wie ausreichendem Speicher, schnellen Netzwerken mit großer Bandbreite usw. versucht, die optimale Leistung zu erbringen. Diese Lösungen sind aber statisch während des Betriebs.

Beim *dynamischen QoS* wird versucht, während des laufenden Betriebes die geforderten Ansprüche für die jeweilige Anwendung hinsichtlich der aktuellen Systemlast zu erfüllen. Hierfür werden dynamisch die Dienste an die Anwendung angepasst. Es werden also QoS-Level eingeführt, für die das System entsprechende Leistungen erbringt. Beispielsweise schaltet die Anwendung dynamisch auf eine niedrigere Auflösung bei einem Videostream, wenn die Netzlast zu groß wird.

Bei den meisten Systemen liegt die Lösung irgendwo dazwischen. Da ein rein statisches System sehr unflexibel ist, wird meistens ein Mittelweg gewählt.

- **Ressourcen-Management und Anwendungsanpassung**

Wenn ein Systems ausgelastet ist und die Ansprüche der Benutzer nicht mehr erfüllt werden können gibt es zwei Arten der Lösung dieses Problems.

Beim *Ressourcen-Management* wird versucht, den erwünschten QoS wieder zu erlangen, indem man versucht, die Ressourcen so zu verwenden, dass der alte Status wieder erreicht wird, dies geschieht z.B. durch Re-Routing oder anpassen der Prioritäten der einzelnen Anwendungen.

Bei der *Anwendungsanpassung* wird dem Benutzer ein etwas schlechterer aber immer noch tolerierbarer Dienst erbracht als erwünscht. Beispiele sind die Qualität eines Videoübertragung z.B. bezüglich ihrer Framerate oder der Bildgröße.

- **Verhandlung und Abrechnung**

Es hat sich als notwendig erwiesen, dass eine Bepreisung von Ressourcen notwendig ist. Ansonsten würde jeder Benutzer die höchste Qualität für seine Anwendung anfordern und das verteilte System wäre sehr schnell überlastet [GTUW04].

Ein Anbieter stellt somit eine breite Palette an Übertragungsqualität an, die unterschiedlich teuer je nach Qualität sind. Somit wird ein Benutzer nur die Qualität anfordern, die er auch wirklich benötigt. Durch Einführung eines solchen Systems könnte diese Qualität dann aber auch garantiert werden. Diese Maßnahme setzt allerdings einen Verhandlungsalgorithmus voraus, der vor der Übertragung durchgeführt werden muss. Weiter ist es nötig, die wirklich gelieferte Leistung exakt messen zu können, um eine Abrechnung durchführen zu können.

4.3 Elemente einer QoS-Architektur am Beispiel CORBA

Nach [BeGe99] werden die folgenden Elemente für eine QoS-Architektur benötigt:

- **Middleware**

Die vorhandenen Middleware-Architekturen für verteilte System (z.B. CORBA oder .NET) unterstützen QoS nicht ausreichend, so dass eine Integration der QoS-Mechanismen notwendig ist. In Abbildung 4.1 wird die Struktur einer Middleware gezeigt.

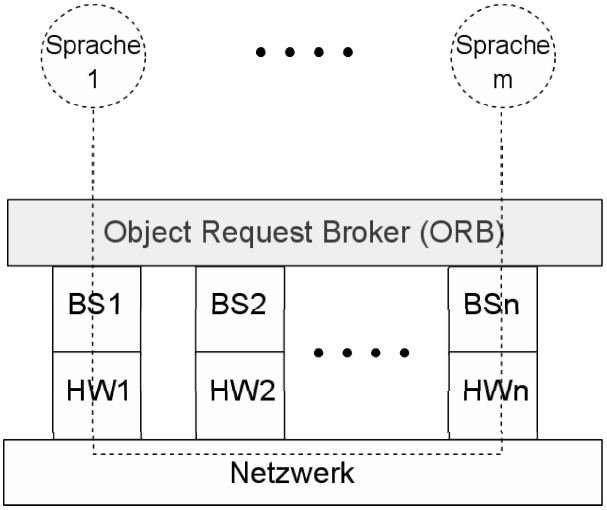

Abb. 4.1 Der Aufbau einer CORBA-Middleware [BeGe99]

Der Object Request Broker (ORB) ermöglicht es den Objekten, die in unterschiedlichen Sprachen geschrieben sind miteinander zu kommunizieren trotz der unterschiedlichen Betriebssysteme, Hardwarekonfigurationen oder Netzwerkstrukturen. Hierfür werden die Schnittstellen durch eine „Interface Definition Language“ (IDL)

realisiert, die die Kommunikation zwischen den verschiedenen Komponenten ermöglicht.

Die QoS-Integration muss sich nun durch das gesamte System ziehen, da alle Systemkomponenten unterschiedliches Augenmerk auf die geforderten QoS-Eigenschaften legen müssen.

- **QoS-Integration während der Entwicklung**

Wenn ein System mehrere QoS-Kategorien unterstützen soll, oder wenn weitere QoS-Parameter integrierbar sein sollen, ist es sinnvoll, eine Schnittstelle für die QoS-Parameter und QoS-Operationen einzuführen. Diese wird in der Middleware realisiert Es gibt zwei Ansatzpunkte:

- *Erweitern der IDL*: Die vorhandene Schnittstelle in der Middleware wird um QoS-Definitionen und QoS-Operationen erweitert.
- *Einführen einer QoS-Definition-Language*: Es wird eine eigenständige Sprache eingeführt, die parallel zu der IDL die QoS-Definitionen und QoS-Operationen unterstützt.

- **QoS-Integration während der Laufzeit [AuCH96]**

Die QoS-Integration während der Bereitstellung durchläuft zwei Phasen:

- *QoS-Bereitstellung* (QoS Provision): Diese Phase durchläuft drei Schritte:
  - *QoS-Mapping*: Die Aufgabe von QoS ist es, die Anforderungen der Nutzer zu erfüllen. Beim Mapping werden diese Ansprüche auf die einzelnen Ebenen des Systems übertragen. Die verschiedenen Komponenten eines Systems wie z.B. Betriebssystem, Netzwerk oder Speicher haben unterschiedliche Schwachstellen und ein QoS-Kriterium hat unterschiedliche Ausprägungen in diesen Ebenen. Deshalb ist es nötig, diese QoS-Klassifikationen in für die einzelnen Komponenten verständliche Parameter zu „übersetzen“
  - *Verfügbarkeitsüberprüfung* (admission testing): Es wird nun auf Grundlage der beim QoS-Mapping ermittelten Werte überprüft, ob es möglich ist den Dienst mit den vorgegebenen Eigenschaften zu erfüllen und welcher Dienstanbieter über welchen Pfad in Frage kommt.
  - *Ressourcenverwaltung* (resource reservation): In diesem Schritt werden die ermittelten Ressourcen, wie CPU, E/A-Geräte, Speicher und Netzwerkpfade für den angefragten Prozess reserviert. Der Netzwerkpfad wird durch den Router auf Grundlage aller zur Verfügung stehenden QoS-Spezifikationen festgelegt.
- *QoS-Management*: Um einen vereinbarten QoS-Level zu halten, müssen oft korrigierende Schritte einzuleiten. Im Folgenden werden nun die Operationen aufgezeigt, die notwendig sind um einen QoS-Level zu halten:
  - *QoS-Überwachung* (QoS monitoring): Bei der Überwachung werden die geforderten Parameter gemessen, wie den Durchsatz, den Datenverlust oder die Übertragungszeit.
  - *QoS-Anpassung* (QoS maintenence): Hier werden die gemessenen Werte mit den geforderten Parametern verglichen und falls nötig korrigierende Maßnahmen ergriffen. Es gibt zwei Möglichkeiten auf eine reduzierte Performance zu reagieren: Erstens die QoS-Degradierung, dabei wird dem Benutzer mitgeteilt, dass der vereinbarte QoS-Status nicht aufrecht gehalten werden kann. Der Benutzer hat nun die Möglichkeit zu entscheiden, ob ihm die verminderten Leistungen für seine Anwendung genügen oder nicht. Die zweite Möglichkeit ist

es, nach Möglichkeit mehr Ressourcen für den Prozess zu reservieren, oder dem Prozess eine höhere Priorität zu geben, um den gewünschten QoS-Level aufrecht zu halten.

5 - Dependability Benchmarking[5]

Dependability Benchmarks wurden eingeführt, um Informationssysteme hinsichtlich ihrer Zuverlässigkeitsattribute zu untersuchen und einschätzen zu können. Der Vergleich mehrerer Systeme wird durch Dependability Benchmarks möglich. Es ist dazu nötig standardisierte Maßeinheiten für die zu untersuchenden Attribute des Systems festzulegen.

Herkömmliche Benchmarks widmen sich weitestgehend dem Gebiet der Leistung. Dependability Benchmarks gehen weiter, hier wird das System auf alle Attribute der Zuverlässigkeit, wie Verfügbarkeit, Funktionsfähigkeit, Sicherheit, Geschütztheit und Wartbarkeit untersucht. Es ist nicht selten der Fall, dass ein Ausfall im System mehrere Attribute der Zuverlässigkeit verletzt. An einem einfachen Beispiel soll dies verdeutlicht werden [AsBD04]:

Für die Performance wird ein Grenzwert für die Antwortzeit des Systems vorausgesetzt. Wird dieser Grenzwert überschritten wird sicher diese Forderung verletzt. Es ist nun aber möglich, dass dieser Ausfall auch Auswirkungen auf andere Attribute wie z.B. der Verfügbarkeit hat. Wenn bei einem Online-Warenhaus wie z.B. amazon die Antwortzeit für den Benutzer zu groß wird, so wird er die Seite schnell wieder verlassen, ohne seinen Einkauf getätigt zu haben, somit kann die zu große Antwortzeit auch als ein Problem der Verfügbarkeit angesehen werden. Oder wenn die zu große Antwortzeit beim Versenden einer Fehlermeldung auftritt, wird der Administrator nicht rechtzeitig auf eine gefährliche Situation im System reagieren können, was somit eine Gefährdung der Sicherheit zu Folge hat.

Dieses einfache Beispiel verdeutlicht, wie differenziert die Ausfälle im Bezug auf die Zuverlässigkeit eines Systems betrachtet werden müssen. Es ist daher nötig, dem Benchmarkingsystem sehr detaillierte Spezifikationen vorzugeben.

5.1 Benchmarkspezifikationen

Da eine Benchmark mehrere Systeme miteinander vergleichbar machen soll, ist es notwendig, dass die jeweiligen Systeme genau bekannt sind und dass die Umgebung in der die Tests ablaufen sehr genau kategorisiert sind [LAAS04]. Als erstes ist es muss das System genau beschreiben werden können. Sowohl die Hardwarekomponenten, als auch die verwandte Software, die Vernetzung und alle Kommunkationssysteme müssen genau deklariert sein. Des weiteren ist es wichtig, die Benchmarkumgebung des Systems genau beschreiben zu können, in welchem Lebenszyklus sich das System befindet, wie viele und welche Arten von Benutzer arbeiten auf dem System und welche Ergebnisse man mithilfe der Benchmark für dieses System erlan-

[5] Der Begriff Zuverlässigkeitsbenchmark ist in der Literatur nicht etabliert, darum wird hier der englische Begriff Dependability Benchmarking verwendet.

gen will. Weiter ist es notwendig, die Maßeinheiten der Ergebnisse zu spezifizieren und zu erläutern wie sie für das Ergebnis der Untersuchungen zu interpretieren sind. Eine dritte Sparte der Spezifikationen ist die Testdefinition, so ist es wichtig, nicht nur die zu untersuchende Software zu betrachten, sondern die gesamte Umgebung in der das System läuft. Die klare Definition des Workloads[6], also der Eingabe in das zu untersuchende System, die eine Arbeitsumgebung simulieren soll, sowie des Faultloads, also der Simulierung von Fehlersituationen, um zu untersuchen, wie das System mit Ausnahmesituationen fertig wird.

- **Kategorisierungsklassifikation**

  Als Erstes betrachten wir uns die *Kategorisierungsklassifikation* (Categorization dimension). Hier ist es wichtig, den *Anwendungsbereich* des zu untersuchenden Systems zu beschreiben. Unterschiedliche Anwendungsbereiche erfordern unterschiedliche Benchmarks [MuSW02]. Eine Datenbankanwendung hat sehr spezielle Anforderungen mit sehr strikten Vorschriften, wie sie andere Programme nicht haben. Außerdem muss die *Ausführungsumgebung* in Betracht gezogen werden. Eingebettete Systeme sind z.B. anfälliger gegen physikalische Fehler, während für Datenbankanwendungen die Benutzerfehler eine sehr große Rolle spielen. Aufgrund diesen Klassifikationen werden die Maßeinheit der Benchmark gewählt (benchmark measures).

  Weiter muss der Bezug zwischen dem untersuchten System und der Benchmark dargestellt werden. Es wird die Lebensphase des Systems festgelegt, in der die Benchmarkuntersuchung stattfindet, wer der Benchmarkbenutzer (Benchmark user) ist, also wer die Ergebnisse verwendet und wer Benchmarkausführender (Benchmark performer) ist, der die Untersuchung durchführt. Diese beiden Gruppen haben unterschiedliche Einblicke in das System und unterschiedliche Vorstellungen vom Ergebnis der Untersuchungen. Diese müssen in Einklang gebracht werden. Außerdem wird dargestellt, welchen Zweck die Benchmarkerhebung erfüllen soll, bzw. welche Erkenntnis man aus der Untersuchung gewinnen möchte.

- **Maßeinheitenspezifikation**

  Im nächsten Schritt werden die Maßeinheiten im Bezug auf die Kategorisierung gewählt. Diesen Schritt nennt man *Maßeinheitenspezifikation* (Measure dimension). Als erstes muss die *Art der Maßeinheit* gewählt werden, handelt es sich um ein qualitatives oder um ein quantitatives Attribut.

  Es gibt *globale Maßeinheiten*, die alle durch Modelle und experimentell gewonnenen Erkenntnisse als Ergebnis für ein Attribut ermittelt, z.B. erhält man nach einer Benchmarkuntersuchung ein globales Maß für die Verfügbarkeit. Anders verhält es sich mit *spezifischen Einheiten*. Beispielsweise gibt es für die Fehlererkennung zwei spezifische Maßeinheiten, den Erkennungsfaktor (die Wahrscheinlichkeit, dass ein im System vorhandener Fehler erkannt wird) und die Latenzzeit (die durchschnittliche Zeit, die vergeht, um einen Fehler zu erkennen).

  Außerdem muss festgelegt werden, wie der Wert ermittelt wurde, durch ein Modell oder durch ein Experiment.

- **Testumgebung**

[6] Hier werden die englischen Wörter Workload und Faultload verwendet, da sie auch im Deutschen gebräuchlich sind und die Übersetzung Arbeitslast und Fehlerlast eher zu Missverständnissen führen könnten.

Der letzte Teil der Spezifikation ist die *Testumgebung* (Experimentation Dimension). Hier wird das „*System Under Benchmarking*“ (SUB) angegeben, also die Hardware- und die Softwareumgebung der zu untersuchenden Systemkomponente.

Weiter muss die Eingabe für das System spezifiziert werden. Wichtig ist die Eingabe, die ein normales Arbeitsprofil simuliert, der so genannte *Workload.* Ein Workload muss für den angegebenen Arbeitsbereich des Systems kreiert werden. Er sollte die zu erwartenden Arbeitsbedingungen, wie Lasten und Datenmengen, Anzahl von Transaktionen angemessen simulieren, um brauchbare Ergebnisse für den späteren Gebrauch in der echten Arbeitsumgebung zu liefern. In der Regel gibt es zwei Arten von Workloads, einmal den Hintergrund-Workload, der die Aktivitäten während des normalen Betriebs eines Systems simuliert, zum anderen den Vordergrund-Workload, der dazu konzipiert wird Situationen der außergewöhnlichen Last oder extrem umfangreichen Anfragen an das System oder auch von Fehlersituationen zu simulieren. Diese beiden Workloads verlaufen parallel in einer Testumgebung, wobei der Vordergrund-Workload eine gute Beobachtbarkeit besitzen sollte, um Messungen für Extremsituationen durchführen zu können. Es gibt schon für viele Arbeitsumgebungen vordefinierte Workloads, die von Gremien der Computerindustrie entwickelt und überprüft wurden und es ist ratsam, diese auch zu verwenden Beispiel für einen solchen Workload ist der TPC-C-Depend, eine Erweiterung des TPC-C-Workloads um Zuverlässigkeitskriterien [TPCC05].

Um alle Attribute der Zuverlässigkeit messen zu können benötigt man aber auch noch einen *Faultload*. In ihm werden Fehler und Ausnahmebedingungen für das System simuliert. Es ist oft schwierig Fehlerursachen des Systems adäquat zu simulieren, z.B. muss man für *Hardware-Fehler* wie zum Beispiel einen defekten Speicherbaustein eine spezielle „software implemented fault injection“ SWIFI-Technik benutzen, um Speicherbereiche zu löschen oder zu verändern. Dies ist nötig, um den Fehler auf der gleichen Ebene einzuschleusen, auf der er in Wirklichkeit auch auftreten kann. *Software-Fehler* müssen in der Ebene der API eingeschleust werden. Das Emulieren von Benutzerfehler ist relativ einfach, da es in der selben Schnittstelle passiert, über die auch ein Benutzer mit dem System kommuniziert.

5.2 Benchmarkdurchführung

Zur Durchführung einer Benchmark ist es unerlässlich, ein geeignetes Szenario zu erstellen um die erforderlichen Metriken in korrekter Form ausarbeiten zu können. Außerdem werden ein Benchmark-Management-System und spezielle Prozeduren und Regeln benötigt, um den Ablauf einer Benchmark ausführen und überwachen zu können.

- **Szenarios**

Das Erstellen einer Benchmark geschieht in drei Schritten. Im ersten Schritt, der Analyse, werden die Maßeinheiten herausgestellt. Einige können direkt in der Testumgebung implementiert werden, für andere ist es notwendig ein Modell des Systems zu erstellen um Ergebnisse erlangen zu können. Der zweite Schritt, die Modellierung, hängt sehr speziell von dem zu untersuchenden System ab. Der dritte Schritt ist die eigentliche Testumgebung, hierzu gehören außer dem Zielsystem auch der Workload und der Faultload. In Abbildung 5.1 ist ein solches Szenario dargestellt.

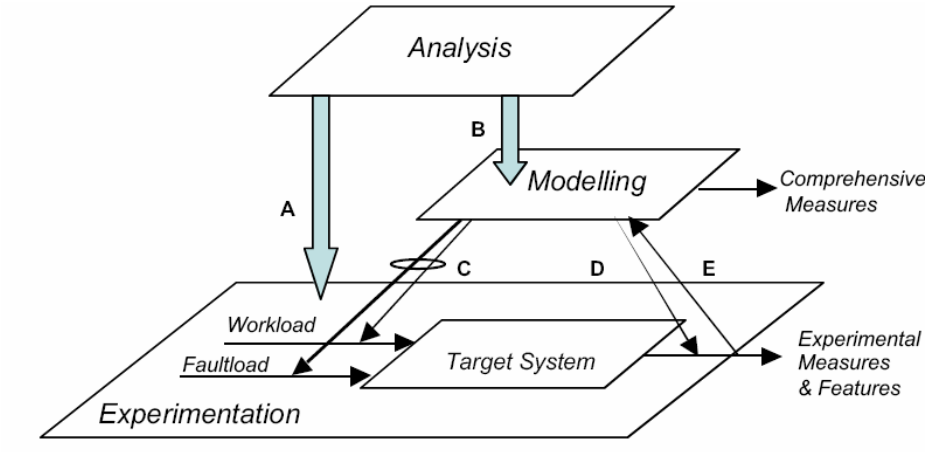

Abb. 5.1 Das Szenario einer Benchmarkdurchführung [LAAS04]

- **Benchmark-Management-System**

Um das Dependability-Benchmarking durchführen zu können, benötigt man zusätzlich zum SUB noch weitere Komponenten. Diese Komponenten sind allerdings ausschließlich dazu gedacht, die Interaktion mit dem SUB und dem Benchmarking-Target zu erleichtern. Aufgabe des *Benchmark-Management-Systems* (BMS) sind die Einspeisung des Work- und Faultloads und die Ermittlung der gewünschten Ergebnisse.

Das Ziel eines BMS ist es, die Experimente des Benchmarking-Prozesses zu kontrollieren und zu automatisieren. Die Anforderungen an das BMS sind in der Benchmark-Spezifikation definiert und hängen sehr speziell von der aktuellen Benchmark ab.

- **Prozeduren und Regeln**

Es ist wichtig, Prozeduren und Regeln vorzugeben, um den Ablauf einer Benchmark-Durchführung zu steuern. Diese sind in der Regel sehr speziell, abhängig von der Benchmark. Einige allgemeine Gesichtspunkte werden hier aufgelistet.

- Es werden Prozeduren benötigt, die den in der Spezifikation allgemein definierten Work- und Faultload in eine für das aktuelle System verständliche Eingabe umformen
- Um zu ermöglichen, eine Benchmark auf Systeme unterschiedlicher Größe anzuwenden ist es nötig, den Workload an das entsprechende System anzupassen. Hierfür werden Skalierungsregeln verwendet.
- Regeln für die einzelnen Messungen, wie die erlaubten Messinstrumente und deren Eigenschaften.
- Aus den Messergebnissen müssen die Resultate bestimmt werden, die für die Benchmark wichtig sind. Hierzu braucht man Formeln und Fehlerberechnungen und auch Zeitintervalle, in denen statistische Erhebungen gemacht werden sollen.

– Es ist erforderlich, die genaue Systemkonfiguration offen zu legen, damit eine exakte Interpretation der Benchmark-Ergebnisse möglich ist.
– Regeln, die es verhindern, dass Entwickler ein System genau an die Benchmark anpassen und so sehr optimistische (bzw. pessimistische) Ergebnisse für das System erlangen.

5.3 Validierung

Um eine brauchbare Benchmark zu entwickeln ist es nötig, wichtige Bedingungen zu erfüllen. Diese Bedingungen müssen schon während der Spezifikation und Implementierung berücksichtigt werden und diese müssen auch verifiziert werden um eine Vertrauenswürdigkeit der Benchmark zu gewährleisten. Dies ist besonders wichtig, um Systeme vergleichen zu können und aus den Ergebnissen der Benchmark richtige Schlüsse ziehen zu können.

In diesem Abschnitt werden einige Richtlinien zur Validierung gegeben.

- **Repräsentativität**

Um brauchbare Ergebnisse zu erlangen, ist es nötig, dass die Maßeinheiten die gewählt wurden, auch die zu überprüfende Systemeigenschaft widerspiegeln. Außerdem ist es wichtig, dass der spezifizierte Workload und Faultload den wirklichen Belastungen des Systems entspricht. Deshalb ist es ratsam, vordefinierte Workloads zu verwenden, da diese speziell entwickelt und getestet wurden.

- **Wiederholbarkeit und Reproduzierbarkeit**

Wiederholbarkeit bedeutet, dass bei einer mehrfachen Ausführung der Benchmark (mit dem selben SUB und Workload/ Faultload) statistisch vergleichbare Ergebnisse zu stande kommen.

Reproduzierbarkeit heißt, dass andere Personen statistisch äquivalente Ergebnisse erhalten, wenn sie eine Benchmark mit den selben Spezifikationen auf dem selben SUB ausführen. Dazu ist es einerseits nötig, die Spezifikationen allgemein genug zu halten, um sie auf eine Klasse von Systemen anwenden zu können, andererseits müssen sie auch speziell genug sein, damit sie direkt wieder verwendet werden können, ohne sie für andere Systeme anpassen zu müssen.

- **Übertragbarkeit**

Die Übertragbarkeit ist notwendig, damit man die Benchmark auf verschiedenen Systemen ausführen kann, damit man vergleichende Ergebnisse bekommt. Zum Beispiel müssen der Workload und Faultload so spezifiziert sein, dass sie auf andere Systeme übertragbar sind.

- **Eindringung**

Die Implementierung eine Benchmark soll so wenig wie möglich in das SUB eindringen, da sonst die Benchmark selbst die Ergebnisse verfälschen kann. So sollte z.B. vermieden werden, dass der Faultload direkt im Zielsystem eingespeist wird, sondern von Außerhalb.

- **Dauer und Kosten**

Die Durchführung eine Benchmark ist nur sinnvoll, wenn der Gewinn durch die Benchmark größer ist als ihre Kosten. Darum ist es wichtig, die Kosten und die Ausführungszeit (Setup, Ausführung und Analyse) zu minimieren. Dabei stehen die Ge-

sichtpunkte ausführliche Untersuchung des Systems und schnelle Ausführung im Widerspruch, es wird meistens ein Mittelweg gesucht.

5.4 Ergebnisse der Benchmarkdurchführung

Ziel des Dependability-Benchmarking ist es, Vergleichsmöglichkeiten von Attributen der Zuverlässigkeit zwischen verschiedenen Systemen zu erlangen. Es ergeben sich zwei Hauptschwierigkeiten:

- Wie soll man die große Menge an Daten analysieren, die durch die Experimente erzeugt wurden, besonders wenn viele Aspekte des Setups (z.B. SUB, Systemkonfigurationen, Fault- und Workload) miteinbezogen werden müssen?
- Wie sollen die Ergebnisse von verschiedenen Experimenten auf verschiedenen System verglichen werden? Besonders schwierig ist das, wenn verschiedene Hilfsmittel oder inkompatible Datenformate verwendet wurden.

Als Lösung für dieses Problem wird „On-Line Analytical Processing“-Technologie (OLAP) verwendet. Die Daten, die aus den verschiedensten Experimenten auf unterschiedlichen Systemen erhalten wurden, werden in einer multidimensionalen Datenstruktur gespeichert und können dann durch Data-Warehousing und OLAP-kompatible Hilfsmittel analysiert werden.

Der Vorteil, der sich bietet ist, dass es einfach ist, Vergleiche und Zusammenhänge aus verschiedenen Experimenten aufzustellen. Dafür stehen schon vorhandene OLAP-Prozeduren zur Verfügung. Außerdem ist es einfach die Ergebnisse zur Verfügung zu stellen. Die multidimensionale Datenbank kann ins Netz gestellt werden und mit OLAP-Hilfsmitteln von jedem Interessierten analysiert werden.

6 – Zusammenfassung

Zuverlässigkeit ist ein wichtiges Kriterium für Informationssysteme. Die Gewährleistung der geforderten Eigenschaften eines Systems ist eine der wichtigsten Aufgaben eines Systementwicklers. Besonders durch die wachsende Bedrohung durch Viren und Würmer werden Zuverlässigkeitsverfahren wie Fehlertoleranz und Fehlervorhersage immer bedeutender.

Im Bereich der Dienstgüte muss man Abstriche machen. Sobald eine Netzwerkstruktur das Internet umfasst wird eine Gewährleistung von Übertragungsraten unmöglich, da in bestehenden Transportprotokollen keine Aussage über Qualitätsmerkmale getroffen werden kann. In diesem Bereich wird sich QoS eher auf der Ebene der Anwendungsanpassung abspielen. In lokalen Netzen (z.B. ein internes Firmennetz) ist eine Einführung von QoS-Architekturen aber durchaus vorstellbar.

Die Durchführung einer Dependability-Benchmark ist sicher sinnvoll für einen Anwender, auch um die Entscheidung zu unterstützen, welches System eingesetzt werden soll. Allerdings dürfen die Kosten für eine solche Benchmark (Zeit und Geld) nicht den Nutzen übersteigen, den man daraus ziehen will.

Literaturverzeichnis

[ALLR03] Algirdas Avinžienis, Jean-Claude Laprie, Brian Randell und Carl Landwehr: “Basic Concepts and Taxonomy of Dependable and Secure Computing“ IEEE Computer Science 2003.

[AsBD04] Victor Basili, Paolo Donzelli, Sima Asgari: “A Unified Model of Dependability: Capturing Dependability in Context” IEEE Computer Sience 2004.

[AuCH96] Cristina Aurrecoechea, Andrew Campbell, Linda Hauw: „A Survey of QoS Architectures“ Columbia University, New York 1996.

[BeGe99] Christian Becker, Kurt Geihs: “QoS as a Competitive Advantage for Distributed Object Systems: From Enterprise Objects to a Global Electronic Market” University of Frankfurt/ Main 1999.

[GTUW04] Kurt Geihs, Andreas Tanner, Andreas Ulbrich, Torben Weis: “Dienstgüte in verteilten Anwendungen” Technische Universität Berlin 2004.

[Härd05] Theo Härder: “DBMS Architecture – New Challenges Ahead” Datenbank-Spektrum 2005.

[ITU97] ITU: „ITU-T Recommendation X.641“ 1997.

[LAAS04] “DBench Dependability Benchmarking” LAAS-CNRS (France) 2004 technical Report.

[Lapr92] Jean-Claude Laprie: “Dependability: Basic Concepts and Terminology” Springer-Verlag 1992.

[LaPS82] Leslie Lamport, Marshall Pease, Robert Shostak: “The Byzantine Generals Problem”, ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, July 1982, Pages 382-401

[MuSW02] Don Wilson, Brendan Murphy, Lisa Spainhower: “Progress on Defining Standardized Classes for Comparing the Dependability of Computer Systems” Proccedding DSN Workshop on Dependability Benchmarking 2002.

[OMG97] Object Management Group (OMG): “Quality of Service: OMG Greenpaper“ Version 0.4a – 1997.

[Siem06] Siemens Online Lexikon: networks.siemens.de/solutionprovider/_online_lexikon Stand: Januar 2006.

[TPCC05] Transaction Processing Performance Council „TPC Benchmark C Standard Specification” Revision 5.5 Oktober 2005.