Dependability and Fault Tolerance

1. Erläutern Sie die grundlegenden Begriffe der Dependability: Nennen Sie die fünf wesentlichen Attribute (bzw. Requirements) eines "dependable system". Was ist der Unterschied zwischen Availability und Reliability? Erläutern Sie die "dependability threats" Failure, Error und Fault sowie den Zusammenhang zwischen den drei. Erläutern Sie "permanent", "transient" und "intermittent" faults anhand von Beispielen.

Definition von Dependability:

- Die Fähigkeit eines Systems, Services zu liefern, denen man begründet („justifiable" – „mit gutem Grund") vertrauen kann.
- Die Fähigkeit eines Systems, Failures zu verhinden, die häufiger oder schwerwiegender sind, als akzeptiert werden kann.

Attribute:

- **Availability / Verfügbarkeit:** ist die Wahrscheinlichkeit, mit welcher ein Service zu einem beliebigen Zeitpunkt korrekt arbeitet. Sagt jedoch nichts darüber aus, wie lange ein System durchgehend verfügbar ist
- **Reliability / Ausfallsicherheit**: Definiert, wie lange (Zeitspanne) ein System ohne Ausfall korrekt arbeiten kann (MTBF – Mean time between failure). **Unterschied zur Availability:** Ein System, welches 2 Wochen in Jahr gewartet wird, sonst jedoch nie ausfällt, hat eine hohe Reliability, jedoch nur eine Availability von 96%
- **Safety/Sicherheit:** Definiert, wie sicher ein System im Falle eines Fehlers agiert, sodass keine katastrophalen Schäden an Menschen oder Umwelt passieren.
- **Integrity/Integrität:** Das System nimmt keine ungültigen Systemzustände an.
- **Maintainability/Wartbarkeit:** Gibt an, wie leicht ein System modifiziert oder repariert werden kann. Eine hohe Maintainability kann zu einer hohen Verfügbarkeit führen, da im Falle eines Fehlers diese sogar möglicherweise automatisch erkannt und repariert werden kann.

Dependability threats:

- **Fault:** Ist die Ursache eines Errors/Fehlers
- **Error:** Ein Teil des System Status, welcher durch einen Fault ausgelöst wurde. Der Error gehört noch zur Spezifikation und kann behandelt werden.
- **Failure:** Ein Failure tritt ein, wenn ein Error nicht mehr behandelt werden kann, und ein System daher seine Dienste nicht mehr vollständig oder überhaupt nicht mehr anbieten kann.

Fault Tolerance bedeutet, dass ein System auch dann noch sein Service fehlerfrei anbieten kann, wenn Faults im System vorhanden sind (diese müssen **maskiert** werden)

Klassen von Faults:

- **Transient fault:** Treten zufällig, jedoch nicht wiederholt auf. Bsp: Ein Vogelschwarm stört die Funkverbindung.
- **Intermittent fault:** treten zufällig auf, verschwinden wieder, und kommen dann wieder; diese Klasse der Faults ist besonders schwer zu identifizieren. Bsp: ein loser Netzwerkstecker, welcher zu einem Wackelkontakt führt.
- **Permanent fault:** Treten solange dauerhaft auf, bis die fehlererzeugende Komponente ersetzt wurde. Bsp: Hardwaredefekte, Software Bugs

Ein **Dormant fault ist** zB ein kaputter Speicherbaustein, der aber nicht verwendet wird.

2. Wozu benötigt man Fehlermodelle ganz allgemein? Geben Sie verschiedene Fehlermodelle für "fail-controlled systems" an und diskutieren Sie diese v.a. hinsichtlich des benötigten Aufwandes für die Maskierung. Inwiefern ist es u.U. heikel zu spezifizieren, daß ein System "k-fault-tolerant" sein soll?

Wozu:

Fehler werden in Klassen (Fehlermodelle) unterteilt, um die Auswirkungen besser einschätzen zu können

Fehlermodelle (laut Tanenbaum)

- **Crash failure (Absturzausfall)**: Ein Server stürzt ab und gibt keine Rückmeldungen mehr, bis er neu gestartet wird. Bis zum Zeitpunkt des Absturzes arbeitet der Server korrekt. Mithilfe von mehreren redundanten Servern lässt sich dieser Fehler maskieren (da leicht feststellbar), jedoch steigt dadurch der Traffic und es ergeben sich neue Probleme wie z.B. Konsistenz
- **Omission failure (Dienstausfall)**: Ein Server antwortet nicht auf Anfragen. Lässt sich mit redundanten Servern leicht maskieren. Der Failure kann verschiedene Gründe haben:
  - **receive omission**: Der Server hat die die Anfrage nie erhalten
  - **send omission:** Der Server hat die Anfrage erhalten und beantwortet, ist aber nicht in der Lage, die Antwort zu versenden.
- **Timing failure (Zeitbedingter Ausfall)**: Eine Antwort benötigt länger als eine definierte Response-Time. Maskierung abhängig von der definierten Responsetime. Zu kurz gewählt: Fehler tritt oft auf. Zu groß gewählt: Fehler wird zu spät erkannt.
- **Response failure (Ausfall korrekter Antwort)**: Die Antwort eines Servers ist inkorrekt. Es gibt 2 Arten von response Fehlern:
  - **Value failure**: Der Server liefert eine falsche Antwort. zB Suchmaschine liefert Webseiten, nach welchen nicht gesucht wurde.
  - **State transition failure:** Der Server reagiert unerwartet auf eine Anfrage, d.h. er weicht vom Programmablauf ab. z.B.: ein Server erhält eine Anfrage, welche er nicht erkennen kann und startet daraus Aktionen, welche niemals passieren dürften. Da sich dieser Fehler nur schwer erkennen lässt (meist nur durch den User) lässt er sich meist nicht maskieren.
- **Inconsistent failure/byzantine failure:** Ein Server kann zu gleichen Anfragen verschiedene Antworten liefern. Wenn der Fehler vom System nicht intern erkannt wird, liegt ein byzantischer Fehler vor. Da sich diese Fehler nicht erkennen lassen, lassen sie sich auch nicht maskieren.

k-fault-Toleranz:

Ein k-fault-tolerant System ist ein System, bei welchen bis zu k Komponenten ausfallen können, ohne dass das System davon beeinträchtigt wird, und immer noch korrekte Ergebnisse liefert. Das Problem ist, dass die fehlerhaten Prozesse nch weiterhin arbeiten können, jedoch falsche Daten liefern können. --> Dies ist ein byzantinischer Fehler; es sind insgesamt mindestens 3k+1 Prozesse nötig, um k-fault-tolerant zu sein.

3. Wieso benötigt man Redundanz zur Maskierung von Fehlern? Welche Arten von Redundanz gibt es?

Mit Redundanzen kann man fehlerfreies Verhalten zusichern, auch wenn Failures aufgetreten sind.

Arten von Redundanz:

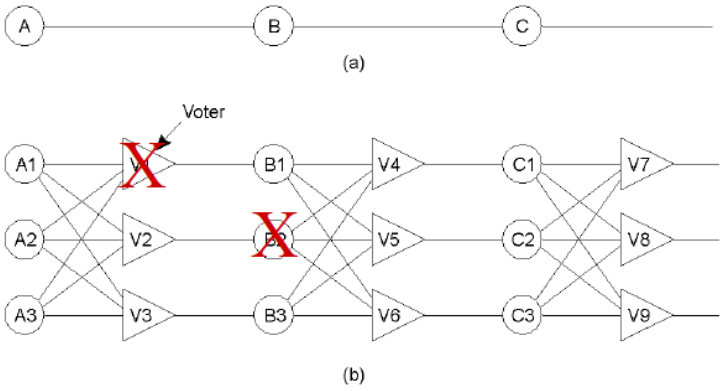
- **Zeitlich:** z.B. mehrmalige Übertragung von Nachrichten
- **Datentechnisch:** Senden von Prüfsummen, Hamming-Distanzen, etc.
- **Physikalisch**: (doppelte Ausführung eines Servers wie z.B. im DNS)

Physikalischen Redundanzen:

- **Aktiv:** Das redundante System läuft ständig im Gesamtsystem mit (zB Standby-Server mit Datenbank)
- **Passiv:** Das redundante System wird nur im Falle eines Faults angekoppelt werden. Weitere Unterscheidung:
  - **Hot standby:** Die (nicht angeschlossenen) redundanten Teile laufen nebenbei mit
  - **Cold standby:** Die (nicht angeschlossenen) redundanten Teile laufen nicht mit

Beispiel: Triple Modular Redundancy (TMR).

Ein Beispiel für eine redundante Systemarchitektur ist **Triple Modular Redundancy (TMR).** Dies ist ein Schaltkreis, bei dem jedes Element einer sequentiellen Aneinanderreihung von Komponenten dreifach ausgeführt wird. Als Input bekommt jedes Element das (von einem Voter ermittelte) Mehrheitsergebnis der vorherigen Komponente. Die Voter müssen, da diese selbst fehlerhaft sein können, ebenfalls repliziert sein. Eine fehlerhafte Komponente kann auf diese Weise ausgeglichen (maskiert) werden.



4. Erläutern Sie die Aussage des "two-army" Problems.

Die Aussage ist, dass 2 Prozesse mit unzuverlässiger Kommunikation nicht zu einem gemeinsamen Konsens kommen können.

Situation:

- Asynchrones Nachrichtensystem. 2 Knoten müssen sich koordinieren (einen Konsens schaffen)
- Die Prozesse sind zuverlässig
- Nachrichten werden nicht verfälscht
- Jedoch ist der Ausfall von Nachrichten möglich (Kanal ist unzuverlässig)
- Es genügt nicht, die Nachricht nr zu bestätigen.
- Beweis von FLP: Es gibt keinen Algorithmus, mit dem ein deterministischer Konses sicher geschafften werden kann

- Beispiel: Verabredung zum Essen per e-mail --> man kann nie wissen, ob die letzte Nachrichten angekommen ist

5. Erläutern Sie die Aussage der "Byzantinischen Generäle".

Achtung: Die Darstellung im Buches ist fehlerhaft und beschreibt nur eine abgewandelte Form des Three Byzantine Problems.

Problembeschreibung:

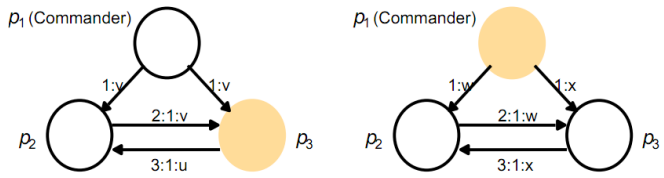
- Es gibt ein System mit n Prozessen (n Knoten)
- Die Kommunikation zwischen den Knoten ist synchron und zuverlässig.
- Jeder Knoten kommuniziert mit jedem
- k Prozesse sind fehlerhaft und liefern falsche Daten.

Lösung:

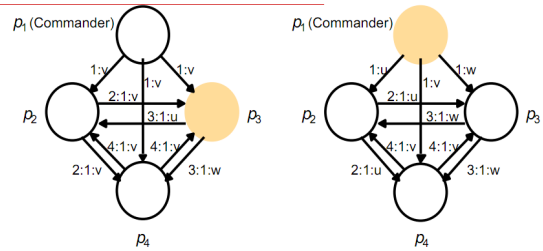
- Es muss gelten: n>=3k+1, d.h. es müssen mindestes 3k+1 Prozesse vorhanden sie, um k falsche Knoten erkennen zu könnne.
- Bei dieser Konstellation ist nur garantiert, dass das System insgesamt korrekt läuft, also k falsche Knoten werden verkraftet. Man kann jeoch nicht erkennen, welche Knoten fehlerhaft sind.

Abbildungen:

- Erklärung der Notation: 1:2:v bedeutet: Prozess1 sagt, dass Prozess2 Befehl "v" sagt:
- Abb 1: 3 Knoten sind nicht ausreichend, um eine Mehrheit zu bilden
- Abb 2: 4 Knoten sind ausreichend



Faulty processes are shown shaded



Faulty processes are shown shaded

6. Erläutern Sie die Fehlerklassen in RPC-Client/server-Umgebungen. Gehen Sie besonders auf das "lost reply" Problem ein.

Client kann Server nicht finden

- zB Server down, falscher client stub.

Client-Anfrage geht verloren.

Server stürzt nach Erhalt der Nachricht ab.

- Problem: Client kann nicht feststellen, ob Absturz vor oder nach Bearbeitung der Nachricht erfolgt ist.

Client stürzt nach Versenden der Anfrage ab.

Lost reply:

Die Antwort des Server geht verloren. Der Client kann nicht feststellen, ob die Antwort oder die Anfrage verloren gegangen ist, oder ob der Server in der Zwischenzeit abgestürzt ist.

Lösungen:

- Anfrage erneut stellen. Dies ist nur sinnvoll bei idempotenten Operationen (Operationen welche wiederholt ausgeführt werden können, z.B.: Daten lesen, während eine Überweisung von einem Konto zu einem anderen nicht idempotent ist).
- Jeder Anfrage eine ID geben, sodass der Server erkennen kann, ob die Anfrage schon bearbeitet wurde, oder ob sie eine neue ist. In diesem Fall ist zu definieren, wie lange sich der der Server die Anfragen merken soll.
- Zusätzlich kann der Client ein Flag anhängen, das definiert, ob die Nachricht neu ist, oder eine wiederholte Anfrage darstellt.

7. Was versteht man unter reliable bzw. ordered multicast (group communication) in statischen Gruppen von Prozessen? Was muss man bedenken, wenn sich die Gruppen dynamisch verändern können? Erläutern Sie das Prinzip des "atomic multicast" ("virtual synchrony").

Reliable bzw. ordered multicast:

- Diese Service garantieren, dass Nachrichten an alle Mitglieder einen Prozessgruppe gesendet werden.
- Eine gesicherte point-to-point Verbindung wird i.A. von der Transportschicht bereitgestellt.
- Es muss gewährleistet sein, dass alle Mitglieder die Nachrichten in der gleichen Reihenfolge erhalten. Um die Ordnung einzuhalten, kann eine Sequenznummer (Timestamp) an die Nachrichten angehängt werden.
- Es müssen berücksichtit werden, dass Mitglieder abstürzen können oder neu hinzukommen.
- Gibt es N Empfänger, kommen in der Regel auch N Acknowledgements zurück, dadurch kann der Sender "überschwemmt" werden. Eine Möglichkeit ist, dass Empfänger den Sender nur informieren, wenn sie eine Nachricht nicht erhalten haben. Aber in diesem Fall wird der Sender die Nachrichten nicht ewig speichern, sondern sie nach einer gewissen Zeit löschen.

Es gibt 4 verschiedene Arten, Multicasts zu ordnen:

- **Unordered Multicasts:** Es ist nichts definiert
- **FIFO ordered Multicasts:** Nur Nachrichten vom selben Prozess werden in einer definierten Reihenfolge zugestellt (nämlich in der, in der sie gesendet wurden).
- **Causally ordered Multicasts:** Kausal abhängige Nachrichten werden in der Reihenfolge der Abhängigkeit zugestellt. 2 Nachrichten sind abhängig, wenn sie im Sinne von FIFO abhängig sind oder wenn eine gesendet wurde nachdem eine andere empfangen wurde – siehe Vector Timestamps
- **Totally ordered Multicasts:** Alle Nachrichten werden bei allen Prozessen in der gleichen Reihenfolge zugestellt

Atomic Multicast

Meistens ist es nötig zu garantieren, dass Nachrichten entweder alle oder kein Empfänger erhalten, und
dass die Nachrichten bei allen in der gleichen Reihenfolge ankommen. Das wird "Atomic Multicast
Problem" genannt.
Bsp.: An eine Gruppe von Prozessen werden Mutlicast Update-Nachrichten verschickt. Ein Replica
crasht. Das Update wurde aber auf den anderen Replicas trotzdem durchgeführt. Wenn das Replica
wieder aktiv wird, wird es einige Updates vermissen. Bei Atomic Multicast wird das Update erst
durchgeführt, wenn alle anderen Mitglieder sich einig sind, dass das abgestürzte Replica nicht mehr Teil
der Gruppe ist. Wenn es wieder aktiv wird, bekommt es keine Updates, bis es sich wieder als Mitglied
registriert hat und somit alle Daten von einem anderen System geholt hat.

Virtual Synchrony

ist eine stärkere Art von Atomic Multicast: Jede Message wird allen nicht fehlerhaften Group Members zugestellt oder keinem, in derselben Reihenfolge Es soll sichergestellt werden, wenn ein Prozess die Gruppe verlässt (z.B. durch einen Crash), sollen die Nachrichten zu allen Verbleibenden geschickt werden oder zu keinem.

Synchrony Multicasting kann wie folgt realisiert werden. Jeder Prozess in der Gruppe buffert sämtliche Mutlicast-Nachrichten. Sobald ein Prozess die Gruppe verlassen will, beitreten will

oder den Ausfall eines Prozesses entdeckt hat schickt er eine View-Change-Nachricht an alle in der Gruppe. Das veranlasst die Prozesse sämtliche von ihnen gebufferten Nachrichten in die Gruppe auszuschicken und mit einer Flush-Nachricht abzuschließen (was bedeutet, dass der Prozess keine gebufferten Nachrichten mehr hat). Damit wird gewährleistet, dass vor dem View Change alle Nachrichten an alle Prozesse in der Gruppe gehen, falls diese mindestens einen Prozess erreicht haben (eventuell haben sie, als sie ursprünglich ausgeschickt wurden, nicht alle Prozesse erreicht, weil der Sender abgestürzt ist bevor er den Multicast abschließen konnte). Sobald ein Prozess die Flush-Nachricht von allen anderen erhalten hat kann er sicher sein dass er keine an diese Gruppe adressierten Nachrichten verpasst hat und kann somit den View Change durchführen.