Dependability and Fault Tolerance

Buch: 8.1 - 8.4

Fragen:

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.**

Eine Eigenschaft von verteilten System ist, dass es keinen single-point of failure gibt, fällt eine Komponente aus, so werden möglichweise einige andere Operationen beeinträchtigt, jedoch nicht das komplette System. Fehlertoleranz bedeutet, dass ein System auch im Falle eines Fehlers seine Dienste zur Verfügung stellt.

Attribute/Requirements

- *Availability*: wird als Wahrscheinlichkeit angegeben, mit welcher ein Service/System korrekt arbeitet zu einer beliebigen Zeit, sagt jedoch nichts darüber aus, wie lange ein System am Stück Verfügbar ist.
- *Reliability*: Definiert wie lange (Zeitspanne) ein System ohne Ausfall korrekt arbeiten kann. Wenn ein System jede Stunde für 1 ms ausfällt, hat es zwar eine sehr hohe Availability, ist jedoch höchst unzuverlässig.
- *Safety*: Definiert, wie sicher ein System im Falle eines Fehlers agiert, sodass keine größeren Schäden passieren.
- *Integrity*: Das System soll nicht in einen ungewünschten/nicht definierten Zustand gelangen.
- *Maintainability*: Gibt an, wie leicht ein System im Falle eines Fehlers repariert werden kann. Eine hohe Maintainability kann zu einer hohen Verfügbarkeit führen, da im Falle eines Fehlers dieser 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 und schlussendlich zu einem failure führt
- *Failure*: Ein Failure tritt ein, wenn ein System durch einen Error seine Dienste nicht mehr vollständig oder überhaupt nicht mehr anbieten kann.

Klassifizierung von Faults

- *transient*: Treten zufällig, jedoch nicht wiederholt auf (zB Vogelschwarm stört Funkverbindung)
- *intermittent*: Treten zufällig auf, verschwinden, kehren aber wieder (zB Wackelkontakt)
- *permanent*: Treten solange dauerhaft auf, bis die fehlererzeugende Komponente ersetzt wurde (zB Hardware-Defekt)

2. **Geben Sie verschiedene Fehlermodelle ("failure models") an und diskutieren Sie diese. Wozu benötigt man überhaupt Fehlermodelle? Inwiefern ist es u.U. heikel zu spezifizieren, daß ein System "k-fault-tolerant" sein soll?**

Fehlermodelle werden dazu verwendet, um klassifizieren zu können, wie schwerwiegend ein bestimmter Fehler ist, d.h. um mögliche Auswirkungen besser einschätzen zu können.

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	A server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

K-fault-tolerant
Diese Eigenschaft trifft auf Systeme zu, bei welchen bis zu k Komponenten ausfallen können, ohne dass das System davon beeinträchtigt wird und immer noch korrekte Ergebnisse geliefert werden. Das Problem bei der k-fault-tolerance ist, dass Prozesse auch weiterhin aktiv sein können, nachdem sie in einen fehlerhaften Zustand geraten sind. Gegeben der Fall, es tritt ein *byzantinischer Fehler* auf und liefern die fehlerhaften k Prozesse sowohl korrekte, als auch falsche Ergebnisse. Dann sind mindestens 2k+1 Prozesse nötig, um die k fehlerhaften Prozesse zu überstimmen und somit k-fault-tolerant zu sein. In der Realität ist es jedoch ohne statistische Analysen kaum möglich, exakt zu bestimmen, wieviele Prozesse fehlerhaft sind.

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

Um auch zum Zeitpunkt eines Fault ein fehlerfreies Verhalten garantieren zu können, sind Redundanzen notwendig. Diese können entweder *zeitlich* (zB mehrmalige Übertragung wie bei TCP), *datentechnisch* (Prüfsummen, Hamming-Distanzen, etc.) oder *physikalisch* (doppelte Ausführung eines Servers wie zB im DNS) realisiert sein.
Zeitliche Redundanz ist besonders bei transient oder intermittent Faults hilfreich. Im Falle von physikalischen Redundanzen kann man unterscheiden, ob die redundanten Systeme ständig im Gesamtsystem mitlaufen (aktiv) oder nur im Falle eines Faults angekoppelt werden (passiv). Ein Beispiel für eine redundante Systemarchitektur ist Triple Modular Redundancy (TMR). Dabei wird jedes Element einer sequentiellen Aneinanderreihung von Komponenten 3 Mal ausgeführt. Als Input bekommt jedes Element das (von einem Voter ermittelte) Mehrheitsergebnis der vorherigen Komponente. Die Voter müssen, da sie selbst fehlerhaft sein können, ebenfalls repliziert sein.

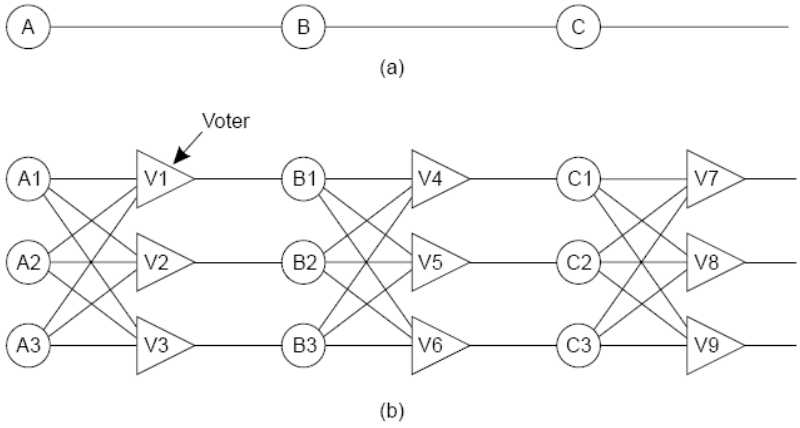


Figure 7-2. Triple modular redundancy.

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

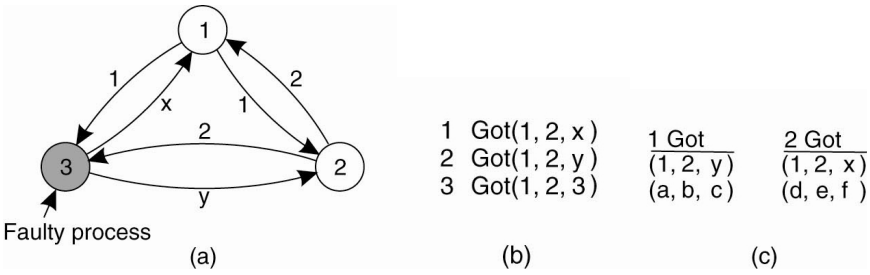
Im two-army Problem geht es darum, dass mit unzuverlässiger Kommunikation 2 Prozesse nicht zu einem gemeinsamen Konsens kommen können. Grundannahme ist, dass es eine rote und eine blaue Armee gibt. Die blaue Armee, bestehend aus 2 Generälen, an verschiedenen Orten, möchte sich einen Angriffzeitpunkt ausmachen. So wird eine Nachricht mittels eines Boten von General A zu General B geschickt, wann dieser Angriff erfolgen soll. General B bekommt die Nachricht und schickt sie an A zurück. A bemerkt, dass B nicht weiß, ob die Nachricht bei A angekommen ist und so möglicherweise nicht angreift. Deswegen schreibt A zurück an B, welcher sich bei Erhalt der Nachricht denkt, dass A nicht sicher weiß, ob die Nachricht angekommen ist. Dies führt zu einem hohen Nachrichten-Aufwand und zu der Feststellung, dass es bei unsicheren Verbindungen keine Sicherheit über den Erhalt der Nachricht gibt. Damit ist es hier nicht möglich, zu einer Übereinkunft zu kommen.

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

Das Problem der *Byzantinischen Generäle* sieht folgende Situation vor: Eine Stadt wird von einer roten Armee gehalten, während auf jedem der umliegenden Berge ein General mit seiner Armee auf den Angriff wartet. Nun sollen die Truppen zwischen den Generälen so ausgetauscht werden, dass die Armee eines jeden Generals die gleiche Truppenstärke aufweist. Im Gegensatz zu dem two-army Problem geht man hier davon aus, dass eine *verlässliche Verbindung* (Telefon) verwendet wird, immer 2 Generäle kommunizieren direkt und verlässlich miteinander und teilen sich ihre Truppenstärke mit. Jeder General berichtet seine wahre Truppenstärke, bis auf einen, der versucht die Kommunikation zu stören und bei jeder Nachricht eine

andere Truppenstärke liefert. Schließlich weiß jeder General über die Truppenstärke der anderen Bescheid und teilt den anderen seine gesammelten Informationen mit, wobei der falsche General wieder fehlerhafte Werte nennt. Es werden nun die Ergebnisse verglichen, die Mehrheit gewinnt, hat kein Wert die Mehrheit, so wird der Wert als unbekannt maskiert. Nun sollte jeder General die Truppenstärke der anderen wissen, nur die des falschen Generals ist unbekannt, er hat es nicht geschafft seine Manipulation erfolgreich durchzusetzen.

Damit dieses Schema (nach Lamport) funktioniert, benötigt man bei k fehlerhaften Prozessen mindestens 2k+1 Prozesse, welche Ordnungsgemäß arbeiten, somit werden *3k+1 Prozesse benötigt*, um k fehlerhafte zu tolerieren. Die folgende Grafik verdeutlicht: Bei 3 Teilnehmern und einem fehlerhaften davon kann es zu keinem Konsens kommen, da keiner eine Mehrheit aus 2 gleichen Vektoren bilden kann. Die Werte a-f werden von General 3 „erfunden“.



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

Client kann Server nicht finden
Dies ist der Fall wenn zB alle Server down sind oder sich die Interfaces in der Zwischenzeit verändert haben.

Anfrage von Client ist verloren gegangen
Eine Möglichkeit um einem solchen Fehler zu begegnen ist, die Nachricht in einer gewissen Periode noch einmal zu senden, sollte keine Antwort vom Server eintreffen.

Server crasht nach Erhalt einer Anfrage
Hier sind 2 Fälle zu unterscheiden: Entweder kommt es vor oder nach der Bearbeitung der Anfrage zum Crash.

Antwort des Servers ist verloren gegangen (lost reply)
Falls der Client keine Antwort vom Server erhält, kann er kaum beurteilen, ob seine Anfrage bereits durchgeführt wurde oder nicht. So könnte seine ursprüngliche Anfrage den Server nie erreicht haben oder aber die Anfrage wurde bereits ausgeführt und nur die Bestätigung ist verloren gegangen. Auch kann die Anfrage den Server zwar erreicht haben, dieser aber gleich danach abgestürzt sein.
Eine Möglichkeit wäre es, die Anfrage erneut zu stellen, was aber nur bei idempotenten Operationen (Operationen welche wiederholt ausgeführt werden können, zB Daten lesen) sinnvoll ist. Man könnte nun versuchen, alle Request idempotent aufzubauen, was aber nicht immer möglich ist (zB Banküberweisung).
Eine weitere Möglichkeit ist, jeder Anfrage eine ID zu vergeben, sodass der Server erkennen kann, ob die Anfrage schon bearbeitet wurde, oder eine neue ist. In diesem Fall stellt sich die Frage, wie lange der Server sich die Anfragen merken soll. Zusätzlich könnte der Client bei jeder Nachfrage ein Bit anhängen, um zu markieren, ob es sich um eine neue Nachricht oder eine Retransmission handelt.

Client crasht nach Senden einer Anfrage
Dies hat ein Verwaisen der Rückantwort zur Folge, was zu unnötiger CPU Last oder im schlimmsten Fall zum Blockieren von Files oder Ressourcen führen kann.

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 Multicasting-Schemes bezeichnen Services, die garantieren, dass Nachrichten an alle gewünschten Prozesse verlässlich versendet werden bzw. dort ankommen. Eine sichere Übertragung an wenige Prozesse kann einfach durch point-to-point Verbindungen (TCP) bewerkstelligt werden. Schwieriger ist es allerdings, ein gesichertes Multicasting anzubieten, vor allem wenn sich die Zahl der Mitglieder einer Gruppe dynamisch ändert, etwa neue Mitglieder hinzukommen oder das Crashen eines Prozesses. Hier muss vor dem

Versenden der Nachricht die Gruppen-Zusammensetzung geklärt sein (*siehe Atomic Multicast*). Zudem sollten die Mitglieder alle versendeten Nachrichten in der gleichen Reihenfolge erhalten. Dies kann man erreichen, indem der sendende Prozess jeder Nachricht eine Sequenz-Nummer anhängt. Ein Receiver kann somit leicht erkennen, ob eine an ihn adressierte Nachricht verloren gegangen ist und entweder den Erhalt bestätigen oder einen Verlust melden.

Ein Hauptproblem bei dieser Form von Multicasting ist die *schlechte Skalierbarkeit*. Gibt es N Empfänger, kommen in der Regel auch N Acknowledgements zurück, die den Server buchstäblich überschwemmen. Eine Möglichkeit ist, dass Empfänger den Sender nur in jenen Fällen informieren, wenn sie eine Nachricht nicht erhalten haben. Jedoch kann der Server aus Performancegründen nicht ewig auf eine solche Fehlermeldung warten sondern wird die ursprüngliche Nachricht nach einer bestimmten Zeit löschen. Dadurch kann es passieren, dass Nachrichten einzelne Empfänger nie erreichen.

Atomic Multicast
Ein Verteiltes System muss sicher stellen, dass alle Nachrichten die Prozesse in einer bestimmten, gleichen Reihenfolge erreichen. Außerdem muss garantiert werden, dass entweder alle Nachrichten oder keine einzige Nachricht versendet werden. Diese Anforderungen werden als *Atomic Multicast Problem* bezeichnet. Sei nun eine verteilte Datenbank gegeben, in der Prozesse zu Gruppen zusammengefasst sind. Somit erhält jede Replica einen eigenen Prozess. Update-Operationen werden nun an alle Prozesse weiter geleitet (Multicasting) und sofort durchgeführt. Wenn während dieses Update eine Kopie crasht, bleibt diese auf dem alten Stand während sich alle anderen Replikas korrekt upgedatet haben. Wird die eine Kopie wieder aktiv, kann es sein, dass sie mehrere Updates versäumt hat.

Im Gegensatz dazu sieht *Atomic Multicasting* vor, dass die restlichen Prozesse nur dann ein Update durchführen, wenn sie sich darauf geeinigt haben, dass der fehlerhafte Prozess nicht Teil ihrer Gruppe ist. Nach dessen Wiederherstellung (auf den alten Stand) muss er der Gruppe wieder beitreten. Dann wird sein Datenbestand dem der anderen Gruppenmitglieder angeglichen. Zusammenfassend wird mit diesem System sichergestellt, dass funktionierende Prozesse einen einheitlichen/konsistenten Eindruck von der Datenbank haben.

Virtual Synchrony
Diese (strengere) Ausprägung eines reliable Multicast garantiert, dass eine Nachricht, die an eine Gruppe gesendet wird, alle funktionierenden Prozesse erreicht. Crasht der sendende Prozess während des Sendevorganges, erhalten entweder alle übrigen Prozesse die Nachricht, oder sie wird, in den Fällen wo sie bereits eingegangen ist, vom Empfänger ignoriert. Virtual Synchrony sowie Atomic Multicast im Allgemeinen basieren auf einer sogenannten Group View - das ist eine Liste, die alle Prozesse einer bestimmten Gruppe zum Zeitpunkt des Multicast, basierend auf dem Kenntnisstand des Senders, beinhaltet. Verändert sich die Gruppenzusammensetzung, findet ein *view change* statt. Das Modell der virtual Synchrony bestimmt, dass ein Multicast, während dessen Übermittlung ein view change auftritt, fertig gestellt werden muss, bevor die Meldung eines veränderten Views an alle Prozesse übermittelt und der View Change statt findet.