Technische Universität Ilmenau
Fakultät für Informatik und Automatisierung
Fachgebiet Telematik
Prof. Dr. Dietrich Reschke

Hauptseminar Telematik
im SS 2003

Algorithmen zu verteilten Konsensfindung

bretreut von:
Dipl.-Inf. Thorsten Strufe

vorgelegt von:
Zielesniak, Christina
christina@zielesniak.de
Matrikel-Nr. 27983

1 Das Problem des Konsenses

1.1 Das Problem und seine Formulierung

Viele der Algorithmen, die man zur Implementierung von verteilten Systemen braucht, nutzen Broadcasting als einen Basismechanismus, um Informationen von einem Prozess oder Prozessor- diese Terme werden austauschbar zueinander verwendet - zu einem oder mehreren anderen zu schicken. Dies wird zum Beispiel gemacht, um Synchronisation in einem verteilten System zu erreichen, oder Konsistenz beim Update verteilter Daten, um Konsens in einem verteilten System zu erzielen, oder um Transaktionen zu handhaben, die sich auf eine verteilte Datenbasis beziehen. Wie auch immer das Netzwerk aussieht, das Problem ist trivial solange es selbst und alle beteiligten Prozessoren zuverlässig sind; falls das Netzwerk auf Broadcasting basiert ist eine einfache Anweisung bereits vorhanden. In einem Punkt-zu-Punkt-Netzwerk wird das Ergebnis erzielt, indem was auch immer für das Senden und Empfangen von Nachrichten notwendig ist, gemacht wird, je nach Topologie des Kommunikationssystems. Aber in Anbetracht von Unzuverlässigkeiten ist das Problem komplex und bildet eine der zentralen Sorgen der verteilten Systeme. Durch Pease, Shostak und Lamport wurde es als das Problem der interaktiven Konsistenz bekannt, durch Lamport als das byzantinische Abkommen oder als das Problem der byzantinischen Generäle. Andere Namen sind das Konsensproblem, das Einstimmigkeitsproblem und viele andere.

Um eine genaue Formulierung des Problems zu geben, nehmen wir ein Netzwerk von n Prozessoren, die miteinander nur über Nachrichten und bidirektionale Kanäle kommunizieren können. Das Ziel ist zuverlässiges Broadcasting der Nachrichten zu garantieren, d.h. wenn irgendein Prozessor Informationen verteilt, erhalten alle anderen Prozesse diese Informationen unverfälscht, haben also die gleiche Auffassung davon. Dies kann explizit in Form eines Bedingungspaares gemacht werden, welches in jeder Lösung des Problems respektiert werden muss:

C1: alle zuverlässig operierenden Prozesse erhalten die gleichen Daten
C2: wenn der verteilende Prozess zuverlässig arbeitet, sind die erhaltenen Daten identisch mit den verteilten Daten

Klar ist, wenn der verteilende Prozess zuverlässig ist, dann folgt C1 aus C2 und es ist einfach ein Protokoll zu schreiben, welches diese Bedingungen respektiert. Aber ein Prozess der eine Broadcastinformation erhält, weiß nicht von vorneherein, ob der verteilende Prozess zuverlässig ist oder nicht. Die Schwierigkeit des Problems liegt bei den Arten von Fehlern, die auftreten können und am Fehlen der Information, ob beteiligte Prozesse ausgefallen sind. Sogar in dem speziellen Fall, wenn alle Prozesse an einen Bus angeschlossen sind und damit die gleichen Nachrichten erhalten, kann ein Fehler C1 verletzen, weil einer der Prozesse die Nachricht empfangen hat andere Prozesse nicht.

Unter den verschiedenen Typen von Prozessfehlern, ist der Außer-der-Reihe-Halt der erste betrachtete. Dies kann entweder endgültig geschehen, wenn der Prozessor weder weitere Nachrichten weiterleitet, noch auf Anfragen von außen antwortet. Oder in Abständen vorkommend, also mit Pausen während denen er mit der Umgebung kontaktlos ist. Bei dem ersten Typ ist jede Annahme, die den ständigen Fortschritt des Prozesses betrifft, verletzt. Vorausgesetzt es gibt eine bekannte obere Schranke für die Übertragungszeit der Nachrichten zwischen Paaren von Prozessoren, welche berechnet werden kann, indem alle Nachrichten bestätigt werden müssen und das Intervall zwischen Senden und Empfangen der Benachrichtigung gemessen wird. Der zweite Typ stellt ein schwierigeres Problem dar. Für

einen Algorithmus, der diesen Fehlertyp löst, ist eine Synchronisation zwischen den verschiedenen Komponenten notwendig, um zu garantieren, dass die chronologische Reichenfolge der Nachrichten richtig ist. Dazu mehr im nächsten Kapitel. Außerdem gibt es die „Ungezogenheit" von fehlerhaften Prozessoren. Solche Prozessoren, von denen nicht bekannt ist, dass sie fehlerhaft sind, können beispielsweise andere, als die empfangenen Werte, weiterschicken, oder sogar gar keine. Bei all diesen Möglichkeiten können keine Annahmen über das Fehlverhalten von Prozessoren gemacht werden und die Aufgabe besteht darin, ein Protokoll auszuarbeiten, welches Konsens zwischen den Prozessoren herstellt.

1.2 Eigenschaften der Lösungen

Nachrichtenaustausch, Zustände

Die Bedingung C2 benötigt ein Protokoll, welches dem Empfänger erlaubt, selbst der Identität des empfangenen Wertes zuzustimmen. Dazu müssen die empfangenen Werte ausgetauscht werden. Jedoch wenn der Prozess nicht als vollständig zuverlässig vermutet werden kann, können Fehlfunktionen während des Austausches und Werte können übertragen werden, die verschieden von den empfangenen sind. Deshalb kann aus einer ersten Austauschrunde nichts geschlossen werden.

Die Anzahl der nötigen Austauschphasen in Hinblick auf die Erzielung von Konsens unter der Berücksichtigung von C1 und C2 ist ein Maß der Zeitkomplexität der Lösung. Ein anderes wichtiges Maß sind die Kosten, gezählt in der Anzahl der auszutauschenden Nachrichten.

Ein grundlegender Parameter

Sei t die maximale Anzahl gleichzeitig ausgefallener Prozesse - entweder dauerhaft oder vorübergehend - die der Algorithmus verkraften muß. Dies ist eine grundlegende Charakteristik des Algorithmus und Dolev und Strong haben gezeigt, dass jede Lösung mindestens t+1 Austauschphasen benötigt, um Konsens zu erzielen. Damit steht die Zeitkomplexität fest. Bleibt also nur noch die Anzahl an auszutauschenden Nachrichten in unterschiedlichen Lösungen zu optimieren.

Signierte und unsignierte Nachrichten

Wie bereits gesehen, liegt eine der Schwierigkeiten des Problems darin, dass Prozesse möglicherweise so fehlfunktionieren, dass sie die ihnen gesendeten Werte ändern. Die Frage ist nun, ob solch ein Verhalten vom empfangenden Prozess beseitigt oder, äquivalent, erkannt werden kann. Die Antwort ist, dass jeder Prozess dazu jede seiner Nachrichten signieren muß. Die Signatur wird jeder Nachricht hinzugefügt und beide werden so verschlüsselt, dass der Empfänger den Ursprung und die Echtheit überprüfen kann. Dabei darf kein Prozess die Signatur eines anderen Prozesses generieren. Dies erlaubt dem Empfänger jede Modifikation des Inhaltes zu erkennen. Solch ein Prozess kann erkennen, wenn ein empfangener Wert von einem fehlerhaften vermittelnden Prozess verändert wurde, und diesen ignorieren. Die Schlussfolgerung daraus ist, daß den Einflüssen von fehlerhaften Prozessen durch Verschlüsselung und Signaturen entgegnet werden kann. Deswegen ist der einzige noch zu betrachtende Fehlertyp das Ausfallen von Prozessen, entweder dauerhaft oder zeitweise. In einem aktuellen System, wo es keine absichtlich fehlfunktionierenden Prozesse gibt, also keine Sabotage, wird ein einfacher fehlerkorrigierender Code ausreichen, anstelle des Signaturmechanismus.

Die allgemeine Situation ist, daß die Schwierigkeiten des Konsensproblems von den Fehlertypen herrührt, gegen welche die Lösung sich durchsetzen muß und das Fehlen des Wissen darüber von vorneherein. Als nächsten wird auf die Problematik der Synchronisation eingegangen. Anschließend werden verschiedene Lösungen des Konsensproblems vorgestellt.

2 Synchronisation

In diesem Kapitel konzentriere ich mich darauf, wie Prozesse sich synchronisieren können. Es ist beispielsweise wichtig, dass mehrere Prozesse nicht gleichzeitig auf eine verteilte Ressource, z.B. einen Drucker, zugreifen, sondern kooperieren, indem sie sich gegenseitig zeitweise exklusiven Zugriff gewähren. Ein anderes Beispiel sind mehrere Prozesse, die sich manchmal auf eine Anordnung von Ereignissen einigen müssen, ob beispielsweise Nachricht m1 von Prozess P vor oder nach Nachricht m2 von Prozess Q gesendet wurde.

Wie sich herausstellt ist Synchronisation in verteilten Systemen meist schwieriger im Vergleich zur Synchronisation in Einzel- oder Mehrprozessorsystemen. Die Probleme und Lösungen die diskutiert werden, treten in vielen verschiedenen Situationen in verteilten Systemen auf.

2.1 Logische Uhren

Für viele Zwecke ist es ausreichend, dass sich alle Maschinen auf die gleiche Zeit einigen. Es ist nicht notwendig, dass diese Zeit mit der realen Zeit, wie sie jede Stunde im Radio angesagt wird. Deshalb ist für eine bestimmte Gruppe von Algorithmen die interne Übereinstimmung der Uhren wichtig und nicht wie nah sie der tatsächliche Zeit sind. Für diese Algorithmen ist es üblich von Uhren als logische Uhren zu sprechen.

Lamport zeigte 1978, dass obwohl sie möglich ist, Uhrensynchronisation nicht absolut ist. Wenn zwei Prozesse nicht interagieren, ist es nicht nötig ihre Uhren zu synchronisieren. Die fehlende Synchronisation ist nicht beobachtbar und kann deshalb keine Probleme verursachen. Darüber hinaus zeigte er auf, dass es reicht, wenn sich alle Prozesse auf die Reihenfolge der Ereignisse einigen und es nicht nötig ist, zu wissen welche Uhrzeit ist. Anschließen stelle ich Lamport's Algorithmus zur Synchronisation logischer Uhren vor.

2.1.1 Lamports Zeitstempel

Um logische Uhren zu synchronisieren hat Lamport eine Relation „happens-before" definiert. Der Ausdruck
a → b heißt a „happens-before" b und bedeutet, dass alle Prozesse sich einig sind, dass als erstes Ereignis a auftritt und anschließend Ereignis b. Die „happens-before" Relation kann direkt in zwei Situationen beobachtet werden:

1) Falls a und b Ereignisse desselben Prozesses sind, und a vor b auftritt, dann ist a → b wahr.
2) Falls a das Ereignis des Sendens einer Nachricht von einem Prozess ist und b ist das Ereignis des Empfangens der Nachricht durch einen anderen Prozess, dann ist a → b

auch wahr. Eine Nachricht kann nicht empfangen werden, bevor sie gesendet wurde, oder zur selben Zeit, da es einen endlichen, ungleich Null großen Betrag an Zeit dauert, um anzukommen.

„Happens-before“ ist eine transitive Relation, wenn also a → b und b → c, dann folgt daraus a → c. Wenn zwei Ereignisse, x und y, in verschiedenen Prozessen stattfinden, die keine Nachrichten austauschen (auch nicht indirekt über einen dritten Beteiligten), dann gilt weder x → y noch y → x. Diese Ereignisse werden als nebenläufig bezeichnet, was nur aussagt, dass nichts gesagt werden kann und auch nicht braucht darüber, wann die Ereignisse passierten und welches Ereignis zuerst passierte.

Was wir brauchen ist ein Weg die Zeit zu messen derart, dass wir für jedes Ereignis a einen Zeitwert C(a) bestimmen können, auf den sich alle Prozesse einigen. Diese Zeitwerte müssen die Eigenschaft haben, dass wenn a →b gilt, auch C(a) < C(b) zutrifft. Die vorher genannten Bedingungen neu formuliert, folgt also aus den Ereignissen a und b, die im gleichen Prozess stattfinden, und a findet vor b statt, dass C(a) < C(b). Gleichermaßen gilt wenn a das Senden einer Nachricht von einem Prozess und b ist das Empfangen dieser Nachricht durch einen anderen Prozess, dann müssen C(a) und C(b) so bestimmt werden, dass jeder den Werten von C(a) und C(b) mit C(a) < C(b) zustimmt. Darüber hinaus muß die Uhrzeit immer vorwärts (zunehmend) gehen und nie rückwärts (abnehmend). Korrekturen der Zeit können nur durch die Addition eines positiven Wertes gemacht werden, nie durch Subtraktion.

Betrachten wir nun den Algorithmus von Lamport um Ereignissen Zeiten zuzuweisen. Auf unterschiedlichen Maschinen laufen verschiedene Prozesse, die jeweils eine eigene Zeit haben und mit eigener Geschwindigkeit laufen. Jede Uhr läuft mit konstanter Frequenz, aber aufgrund von unterschiedlichen Quarzen sind die Frequenzen der Uhren unterschiedlich. Die folgende Situation muß verhindert werden: Eine Nachricht wird zur Zeit a gesendet und zur Zeit b empfangen, wobei a < b ist. Die Lösung dieses Problems folgt aus der „happens-before“-Relation unmittelbar, denn die Nachricht darf frühestens a+1 beim Empfänger ankommen. Dazu enthält jede Nachricht zusätzlich einen Zeitstempel mit der Zeit des Sendens, bezogen auf die Uhrzeit des Senders. Wenn nun also eine Nachricht beim Empfänger ankommt und die Uhr des Empfängers hat eine geringere Zeit als die des Zeitstempels, dann stellt der Empfänger seine Uhr auf mindestens eine Einheit nach dem Zeitstempel vor.

Um zusätzlich der Forderung nach globaler Zeit gerecht zu werden, ist nur eine kleine Änderung nötig. Zwischen jeweils zwei Ereignissen muß die Uhr mindestens um eine Einheit erhöhen. Falls ein Prozess zwei Nachrichten in schneller Abfolge sendet oder empfängt, muß die Uhrzeit der zweiten Nachricht um eins größer sein, als die der ersten.

In manchen Situationen ist eine zusätzliche Bedingung wünschenswert: zwei Ereignisse dürfen niemals zum gleichen Zeitpunkt auftreten. Um dieses Ziel zu erreichen, können wir die Prozessnummern, jeweils für den Prozeß in dem das Ereignis auftritt, ans Ende des Zeitstempels, durch einen Punkt getrennt, anhängen. Wenn nun also in Prozeß 1 und 2 jeweils zur Zeit 40 ein Ereignis auftritt, dann wird das erstgenannte mit 40.1 und das zweite mit 40.2 datiert.

Eine andere Möglichkeit logischer Uhren sind Vektorzeitstempel, darauf will ich aber im Rahmen dieser Arbeit nicht eingehen.

2.2 Physische Uhren

Computer haben jeweils ihre eigene physische Uhr. Diese Uhren sind elektronische Einheiten, die Schwingungen eines Quarzes in einer bestimmten Frequenz zählen und diese typischerweise aufteilen und das Ergebnis in einem Zählregister speichern. Wenn Anwendungen auf einem bestimmten Computer laufen, die nur an der Reihenfolge von Ereignissen interessiert sind, und nicht an der absoluten Zeit, zu der sie passierten, benötigen wir nur den Wert des Zählers um Ereignissen einen Zeitstempel zuzuordnen. Das Datum und die Uhrzeit kann aus den Zählerwerten berechnet werden, solange von einem früheren Zählerstand bekannt sind.

Um Zeitstempel, die von verschiedenen Uhren derselben physischen Bauweise auf verschiedenen Computern stammen, zu vergleichen, könnte man denken, reicht es den Zeitabstand zwischen den beiden Zeitzählern zu kennen. Unglücklicherweise basiert die Annahme auf einer falschen Prämisse: Computeruhren ticken in der Praxis höchst selten mit der gleichen Geschwindigkeit, unabhängig davon ob sie die gleiche physische Bauart haben.

2.2.1 Cristian's Methode zur Uhrensynchronisation

Eine Möglichkeit für einen zentralen Zeitserverprozess S in einem verteilten System Synchronisation zwischen Computern zu erreichen, ist die Zeit in Übereinstimmung mit seiner eigenen Uhr gegen Anfrage zu liefern. Der Zeitserverrechner kann mit einem passenden Empfänger ausgestattet sein, um sich mit der UTC zu synchronisieren. UTC (Universal Time Coordinated) ist ein internationaler Standard, der auf einer Atomuhr basiert. UTC-Signale werden synchronisiert und regelmäßig durch Radiostationen und Satelliten übertragen. Wenn also ein Prozess P die Zeit mittels einer Nachricht m_r anfordert und den Zeitwert t in einer Nachricht m_t erhält, dann könnte P prinzipiell seine Uhr auf die Zeit t + T_{trans} , wobei T_{trans} die Zeit ist, um m_t von S zu P zu übertragen. t wird als der letzte mögliche Zeitpunkt vor der Übertragung durch S's Computer in m_t eingesetzt.

Unglücklicherweise ist T_{trans} abhängig von Schwankungen. Allgemein kann man T_{trans} = min + x annehmen, mit $x \geq 0$. Der Minimumwert min ist der Wert, der erreicht wird, falls keine andereren Prozesse laufen und kein anderer Netzwerkverkehr existiert; min kann gemessen oder konservativ geschätzt werden. Der Wert x ist ein einem speziellen Fall unbekannt, obwohl eine Verteilung der Werte für eine spezielle Installation messbar wäre.

Cristian hat 1989 den Gebrauch solcher Zeitserver, die mit einem Empfänger für UTC Signale ausgestattet sind, vorgeschlagen, um Computer zu synchronisieren. Sei t die Zeit die in S's Nachricht m_t zurückgesandt wurde. Eine simple Abschätzung der Zeit, zu der P seine Uhr gestellt hat, ist t + T_{round} /2, wobei angenommen wird, dass die verstrichene Zeit gleichmäßig vor und nachdem S t in m_t platziert hat, aufgeteilt wird. Seien min + x und min +y die Zeiten zwischen dem Senden und Empfangen von m_r und m_t. Falls der Wert für min bekannt ist oder konservativ geschätzt werden kann, dann können wir die Genauigkeit dieses Ergebnisses wie folgt.

Der früheste Zeitpunkt, an dem S die Zeit in m_t platziert haben kann, ist min nachdem P m_r versendet hat. Der späteste Zeitpunkt dafür ist min bevor m_t bei P angekommen ist. Die Zeit nach S's Uhr zu der die Antwortnachricht ankam, liegt daher in dem Intervall [t + min, t+ T_{round} – min]. Die Breite dieses Intervalls ist T_{round} – 2 min, also ist die Genauigkeit ± (T_{round} /2 – min).

Schwankungen können gehandhabt werden, indem man mehrere Anfragen an T stellt und dann den Minimumwert von T_{round} nimmt, um die genaueste Schätzung abzugeben. Je größer die geforderte Genauigkeit ist, umso kleiner ist die Wahrscheinlichkeit dies zu erreichen. Das liegt daran, dass die genauesten Ergebnisse solche sind, bei denen beide Nachrichten in einer Zeit nahe bei min übertragen werden – ein untypisches Ereignis in einem verkehrsreichen Netzwerk.

2.2.2 Der Berkeley Algorithmus

Gusella und Zatti beschreiben 1989 einen Algorithmus für die interne Synchronisation, den sie für eine Gruppe Computer, auf denen Berkeley UNIX lief, entwickelt haben. Dazu wird ein Computer ausgewählt, der dann als Master agiert. Im Unterschied zu Cristian's Algorithmus befragt der Master regelmäßig die anderen Computer, deren Uhren synchronisiert werden sollen, diese werden Slaves genannt. Die Slaves senden ihren Uhrenwert zurück. Der Master schätzt ihr lokale Zeit durch die Beobachtung der round-trip-Zeit (ähnlich der Technik von Cristian), und bildet den Durchschnitt aus den erhaltenen Werten, dabei wird auch die Uhr des Servers mit in die Berechnung einbezogen. Der Ausgleich der Wahrscheinlichkeiten geschieht durch die Durchschnittsbildung, die die einzelnen Tendenzen der Uhren, schnell oder langsam zu gehen, ausgleicht. Die Genauigkeit des Algorithmus hängt von der maximalen round-trip-Zeit zwischen Master und Slave ab. Der Master sondert alle gelegentlichen Messungen aus, die mit einer größeren Zeit als die dieses Maximums zusammenhängen.

Anstatt jedem anderen Computer die aktualisierte Zeit zurückzuschicken, die weitere Unsicherheiten aufgrund der Nachrichtenübertragungszeit hinzufügen würde, sendet der Master jedem Slave den Betrag, um den diese ihr Uhren berichtigen müssen. Dies kann ein negativer oder positiver Wert sein.

Der Algorithmus sondert Messungen von Uhren, die zu sehr abweichen, die ausgefallen sind oder die falsche Messungen liefern, aus. Solche Uhren hätten einen signifikanten nachteiligen Einfluss, falls ein Durchschnittswert genommen würde. Der Master nimmt einen fehler-tolerierenden Mittelwert. Dieser ist von einer Untergruppe der Uhren genommen, die nicht mehr als um einen bestimmten Wert voneinander abweichen.

2.2.3 The Network Time Protocol

NTP definiert eine Architektur für einen Zeitdienst und ein Protokoll, um die Zeitinformationen über eine große Art von verbundenen Netzwerken zu verteilen. Es wurde als Standard angenommen zur Uhrensynchronisation innerhalb des ganzen Internets.

Der NTP Dienst wird von einem Servernetzwerk, die sich im ganzen Internet befinden, zur Verfügung gestellt. Primärserver sind direkt mit einer Zeitquelle, wie beispielsweise eine Radiouhr, die UTC empfängt, verbunden. Sekundärserver werden mit den Primärservern synchronisiert. Die Server sind in einer logischen Hierarchie verbunden, die Synchronisationsunternetz genannt wird und deren Ebenen strata genannt werden. Primärserver belegen Stratum 1, sie sind an der Wurzel. Stratum-2-Server sind direkt mit den Primärservern synchronisiert; Stratum-3-Server werden von den Stratum-2-Servern synchronisiert usw. Die Server auf der untersten Ebene (Blätter) stellen Workstations dar.

Es gibt drei Modi in denen NTP Server sich synchronisieren: Multicast, Procedure-call und den symmetrischen Modus. In allen Modi wird UDP genommen, die Nachrichten werden also unzuverlässig übertragen. Im Procedure-call-Modus und im symmetrischen Modus werden die Nachrichten paarweise ausgetauscht. Jede Nachricht trägt Zeitstempel der letzten Nachrichtenereignisse: die lokale Zeit, zu der zwischen dem Paar die vorherige NTP Nachricht gesendet und empfangen wurde und die lokale Zeit, zu der diese Nachricht übertragen wurde. Der Empfänger der NTP Nachricht vermerkt die lokale Zeit, zu der er die Nachricht empfangen hat.

Für jedes Nachrichtenpaar, das zwischen den zwei Servern übertragen wurde, berechnet das NTP Protokoll einen Offset o_i , der eine Schätzung des aktuellen Offsets zwischen den zwei Uhren ist und eine Verzögerung d_i, der die gesamte Übertragungszeit der beiden Nachrichten ist.

3 Algorithmen, die Konsens herstellen

3.1 Der Algorithmus von Lamport, Shostak und Pease

3.1.1 Annahmen

Annahmen, das Netzwerk betreffend

Wir nehmen an, dass es n Prozesse gibt, unter denen es maximal t gleichzeitig fehlfunktionierende gibt. Die Umgebung in der sie arbeiten ist ein zuverlässiges Kommunikationsnetzwerk, d.h. jeder betriebsfähige Prozess kann jederzeit eine Nachricht zu jedem anderen senden und die Nachrichten werden unverfälscht in richtiger Reihenfolge empfangen. Um diese Zuverlässigkeit der Kommunikation zu garantieren, müssen bei k gleichzeitigen Kanalausfällen mindestens k+1 unabhängige Pfade zwischen jedem Prozesspaar existieren. Die Kanäle werden als bidirektional angenommen und das Netzwerk kann als ein vollständiger Graph mit n Knoten beschrieben werden.

Annahmen, die Nachrichten betreffend

Wir nehmen die Nachrichten als unsigniert an, so dass ein fehlfunktionierender Prozess den Inhalt modifizieren kann. Die Identität des Senders einer Nachricht ist dem Empfänger bekannt, dies ist leicht mit Hilfe des vollständigen Netzwerkgraphen sichergestellt. Um den Auswirkungen von Komplettausfällen von Prozessen entgegenzuwirken, muß es möglich sein das Fehlen einer Nachricht zu erkennen, die empfangen werden sollte. Dies kann einfach erreicht werden, durch eine obere Schranke δ, für die Zeit um eine Nachricht zu erstellen und zu übertragen und ε für die Diskrepanz zwischen den Uhren des Senders und des Empfängers, welche initial synchronisiert wurden. Eine Nachricht, die zum Zeitpunkt T der Senderuhr generiert und übertragen wurde, sollte nicht später als $T+ \delta + \varepsilon$ beim Empfänger (nach dessen Uhr) ankommen. Falls die Uhren aus der Synchronisation driften, können sie mittels speziellen Algorithmen (siehe Kapitel 2) rückgesetzt werden. Der Konsensalgorithmus muß allen Teilnehmern zu Beginn bekannt sein und kann mit den initialen Variablenwerten übertragen werden.

3.1.2 Ein Unmöglichkeitskriterium

Viele Autoren haben gezeigt, dass es keine Lösung für das Problem gibt, außer wenn die maximale Anzahl t der gleichzeitig fehlfunktionierenden Prozesse echt kleiner als 1/3 der Gesamtgröße n des Netzwerks ist. Also $n \geq 3t + 1$. Anschließend wird an einem Beispiel mit drei Prozessen, von denen einer fehlerhaft ist, diese Zahl erklärt.

P_0, P_1 und P_2 sind drei Prozesse, wobei P_0 der initiale Sender und P_2 der fehlerhafte Prozess ist. In der ersten Phase senden P_0 den Wert a an P_1 und P_2 und in der zweiten Phase tauschen P_1 und P_2 die empfangenen Werte aus. Aber P_2, der fehlerhaft ist, ändert den Wert auf b und sendet ihn zu P_1. P_1 hat nun also in der Endphase die Werte a von P_0 und b von P_2 empfangen. Das zweite ist der Wert den P_2 vermutlich von P_0 in Phase 1 empfangen hat und der möglicherweise vom fehlerhaften Prozess geändert wurde. Falls fehlerhaft gearbeitet wurde, hat P_2 in Phase 2 nicht die von ihm geforderte Austauschoperation ausgeführt und b ist ein Defaultwert. P_1 weiß, durch das Empfangen von zwei verschiedenen Werten, dass fehlerhafte Prozesse im System vorhanden sind, er weiß aber nicht wie viele Prozesse dies sind oder welchen Prozess dies betrifft. P_1 muß nun die Konsensbedingungen C 1 und C2 beachtend, den Wert a wählen. Nun betrachten wir dasselbe System aber mit P_0 als den fehlerhaften Prozess, P_1 und P_2 arbeiten zuverlässig. In der ersten Phase sendet P_0 an P_1 den Wert a und an P_2 den Wert b. Anschließend tauschen letztere ihre Werte aus, P_1 sendet also an P_2 den Wert a und P_2 den Wert b an P_1. P_1 hat die gleiche Auffassung seiner Umgebung wie vorher und deshalb keine Mittel zwischen den beiden zu unterscheiden. Er muss also zu demselben Schluss wir zuvor kommen, also den Wert a akzeptieren. Nach demselben Argument wird P_2 den Wert b, den er vom fehlerhaften Prozess P_0 empfangen hat, als korrekt akzeptieren. So haben die zwei zuverlässigen Prozesse P_1 und P_2 keinen Konsens erzielt, denn die
Bedingung C1 wurde verletzt. Es gibt also keine Lösung für $n = 3$ und $t = 1$.

Es ist wichtig zu erkennen, dass in diesem Fall nichts durch weitere Austausche erreicht wird, diese könnten nur die bereits bekommenen Ergebnisse wiederholen und geben deshalb keine neuen Informationen.

3.1.3 Zugrunde liegende Prinzipien der Lösung

Außer zu zeigen, dass im Fall von drei Prozessen, von denen einer fehlerhaft ist, keine Lösung möglich ist, erklärt das Beispiel den in allen Lösungen benutzten Mechanismus, eine Folge von Phasen mit synchronisierten Nachrichtenaustauschen. Außerdem wurde festgestellt, dass bei bis zu t fehlerhaften Prozessen mindestens t+1 Phasen nötig sind. Die erste Phase besteht aus dem Senden des Wertes durch den Prozess P_0 zu allen Empfängerprozessen. In den folgenden Phasen werden diese Werte zwischen diesen ausgetauscht, wie folgt. Jeder Prozess hängt seinen Identifikator an jede Nachricht, die er verteilt. So wird ein zusätzlicher
Identifikator in jeder Phase an die Nachricht angehängt und diese sieht dann folgendermaßen aus: (Wert: x; P_0, P_{i1}, P_{i2}, ..., P_{ik}).

Wenn der Prozess P_i solch eine Nachricht erhält, interpretiert er sie wie folgt: in der Phase j hat der Prozess P_{ij} die Nachricht (Wert: x; P_0, P_{i1}, P_{i2}, ..., $P_{i(j-1)}$) empfangen. Er hat seinen eigenen Identifikator P_{ij} angefügt und sendet die Nachricht weiter zu den anderen Prozessen inklusive $P_{i(j+1)}$. Die Nachricht sagt nun aus: „P_{ik} sagt, dass $P_{i(k-1)}$ ihm gesagt hat, dass $P_{i(k-2)}$ gesagt hat, ... dass P_{i1} gesagt hat, dass P_0 den Wert x gesendet hat."

Alle zuverlässigen Prozesse arbeiten so in jeder Phase des Nachrichtenaustausches. Solche, die fehlerhaft arbeiten, können Nachrichten falsch ausliefern, sie verändern, sie überhaupt nicht ausliefern oder sie arbeiten manchmal korrekt und manchmal inkorrekt.

Es ist nötig, dass sich alle Prozesse am Anfang auf einen Defaultwert v_{def} für jede Handlung einigen. Dieser wird immer verwendet, wenn ein Wert erwartet wird, aber ausbleibt. v_{def} muß zu der Menge von Werten gehören, die übertragen werden.

3.1.4 Der Algorithmus

Wir setzen eine Funktion *mehrheit* voraus, die zu einer Menge von Werten als Argument den Wert zurückgibt, der am häufigsten auftritt, das Mehrheitsvotum. Falls kein solcher Wert existiert, gibt sie v_{def} zurück.

Wir betrachten eine Menge von n Prozessen, von denen bis zu t gleichzeitig fehlerhaft sein können. Der Algorithmus ist rekursiv gegeben, wobei der Rekursionsparameter der Wert t ist. Wir nennen ihn UM(t), was die unsignierten Nachrichten im Zusammenhang mit bis zu t fehlerhaften Prozessen beschreibt.

Der Algorithmus arbeitet entsprechend dem oben angegebenen Prinzip and benötigt t+1 Phasen des Nachrichtenaustausches. In der Phase 1 sendet der Senderprozess P_0 den Wert w zu den n-1 anderen Prozessen P_l, P_2, ..., P_{n-1} und fügt seinen Identifikator hinzu (w; P_0). In Phase 2 arbeitet jeder dieser anderen Prozesse, beispielsweise P_i, als Sender und hängt seinen Identifikator an w an und sendet dieses Paar zu den n-2 Prozessen P_l, P_2, ..., P_{i-1}, P_{i+1}, ..., P_{n-1}. In Phase 3 hat jeder dieser Prozesse n-1 Nachrichten der Form (w; P_0, P_i) erhalten. Er fügt seinen Identifikator an und sendet die Nachricht dann an die verbleibenden n-3 Prozesse, und so weiter.

Die rekursive Form des Algorithmus' beseitigt die Notwendigkeit an jede Nachricht eine Angabe über den Pfad, den sie genommen hat, hinzuzufügen. Dieser ist implizit in der Nachricht selbst enthalten. Es nehmen natürlich nur die zuverlässigen Prozesse teil, die anderen können irgendwie handeln.

Im folgenden Text ist der Algorithmus in einer allgemeinen Form gegeben, wie er für alle Prozesse gilt. Der sendende Prozess ist dabei P_0.

Algorithmus $UM_n(t)$

begin 1 Sender P_0 sendet seinen Wert zu jedem der anderen n-1 Prozesse P_1, ..., P_{n-1}

2 jeder Empfänger vermerkt den von P_0 empfangenen Wert oder speichert den Defaultwert v_def, falls er nichts empfangen hat

3 **if** t > 0 **then**

begin for jeder Empfängerprozess P_i

3.1 sei w_i der in Phase 2 von P_i vermerkte Wert;
P_i handelt als Sender und sendet w_i mittels $UM_{n-1}(t-1)$
an alle n-2 Prozesse P_1, ..., P_{i-1}, P_{i+1}, ..., P_{n-1};

3.2 sei w_j ($j \neq i$) der Wert den P_j von P_i am Ende der Phase 3.2 empfangen hat oder v_{def}, falls nichts empfangen wurde; P_i speichert die Wertemehrheit (v_1, v_2, …, v_{n-1});

end
end if
end

Implementierung des Algorithmus

Der Algorithmus ist in einer abstrakten Formulierung gegeben, die Rekursion benutzt. Die Implementierung ist keine triviale Angelegenheit: viele Nachrichten können während einer Phase kursieren und es ist wichtig, dass jede um Zweideutigkeiten zu vermeiden identifiziert wird. Das Hinzufügen der Pfadinformation zu dem übermittelten Wert ermöglicht es die Phase, zu der sie gehört, und den rekursiven Aufruf, auf dem sie beruht, exakt zu kennen. Am Ende von Phase k wird jede Nachricht durch eine geordnete Folge von k eindeutigen Identifikatoren fixiert. Es gibt in dieser Phase (n-1)*(n-2)*...*(n-k) solche Nachrichten. Sobald ein Prozess Nachrichten in dieser Phase empfängt, muß er überprüfen, ob

a) es höchstens k Identifikatoren gibt
b) diese alle unterschiedlich sind
c) die erste vom sendenden Prozess P_0 ist
d) die letzte vom letzten weitergebenden Prozess stammt, dies kann mit Hilfe der Annahmen bezüglich
des Netzwerks geschehen
e) der eigene Identifikator nicht in der Liste auftaucht

Wenn alle diese Bedingungen erfüllt sind, kann die Nachricht als gültig akzeptiert werden und kann an der nächsten Phase des Protokolls teilnehmen. Anderenfalls wird sie verworfen. Ein fehlerhafter Prozess kann eine Nachricht durch das Vertauschen der Reihenfolge in der Identifikatorliste verändern, sie bleibt dann aber immer noch gültig. Diese Veränderung wird in einer späteren Phase entdeckt.

Somit müssen die Nachrichten in jeder Implementation aus zwei Teilen bestehen, der erste bezüglich des auszutauschenden Wertes und der zweite ist ein Prüfbereich um die Route, der sie gefolgt ist, zu vermerken.

3.1.5 Andere Annahmen bezüglich des Netzwerks

Unverlässliche Netzwerke

Was kann über das Verhalten von UM(t) gesagt werden, wenn die Annahme der Netzwerkverlässlichkeit verletzt ist? Ein fehlerhafter Kanal kann eine Nachricht bei der Übertragung verlieren - insbesondere wenn der Kanal unterbrochen wird - oder er kann sie verändern, also den Inhalt modifizieren. Gewöhnlich hat ein empfangender Prozess keine Möglichkeit zwischen Kommunikationsfehlern und Fehlern des sendenden Prozesses zu unterscheiden. Wenn wir ein initial vollständiges Netzwerk haben mit n Prozessen, muß der Algorithmus UM(t), mit $n > 3t$, stabil gegen maximal t Fehler sein, egal ob diese von Prozessen oder vom Kanal kommen. Zu beachten ist, dass ein Fehler von einem oder allen Kanälen, mit denen ein Prozess verbunden ist, gleichwertig einem einzelnen Fehlverhalten, dem des Prozesses, ist.

3.2 Lösungen mit signierten Nachrichten

3.2.1 Annahmen, Bedeutung der Signaturen

Die Bedeutung signierte Nachrichten zu nutzen ist, dass gestörte Prozesse nicht berücksichtigt werden müssen, sondern nur vorübergehend und dauerhaft ausgefallene Prozesse. Das heißt, dass es keine Veränderungen von Nachrichten gibt, bzw. diese vom Empfänger erkannt werden. Mit signierten Nachrichten können einfachere Algorithmen entworfen werden, aber weiterhin sind t+1 Phasen notwendig. Diese untere Grenze gilt für alle Lösungen für Punkt-zu-Punkt-Netzwerke.

Das Netzwerk an sich wird als vollständig und zuverlässig angenommen. Der Kommunikationskanal verliert oder verändert weder Nachrichten, noch leitet er sie in falscher Reihenfolge weiter.

Viele Algorithmen mit signierten Nachrichten wurden vorgeschlagen. Ich gebe drei mit unterschiedlicher Nachrichtenkomplexität an, deren unterschiedliche Ansätze einen Vergleich interessant machen. Der Parameter t gibt wieder die maximale Anzahl gleichzeitig ausfallender Prozesse an.

3.2.2 Der Lamport, Shostak und Pease Algorithmus

Das Prinzip

Das Prinzip ist einfach. Der sendende Prozess P_0 sendet eine signierte Nachricht mit dem Wert v, den er broadcasten will, zu allen n-1 anderen Prozessen. Falls P_0 selbst fehlerhaft ist, sendet er möglicherweise verschiedene, willkürliche Werte mit verschiedenen Nachrichten. An diese hängt jeder empfangende Prozess seine eigene Signatur und übergibt die erweiterte Nachricht weiter an alle jene Prozesse, die diese noch nicht signiert haben, usw. Wenn in der (t+1)ten Phase ein Prozess keine weiteren Nachrichten empfängt, wendet er die Funktion „choice“ (Auswahl) auf die empfangene Menge von Werten, um den Konsenswert zu

ermitteln. Wie zuvor ist die k-te Phase in dieser Prozedur definiert als die Phase, in der die kursierenden Nachrichten k Signaturen tragen.

Falls ein fehlerhafter Prozess eine Nachricht signiert und diese später ändert und diese geänderte Nachricht versendet, wird dies vom Empfänger entdeckt, der die Nachricht verwirft.

Der Algorithmus

Algorithmus SM(t) - mit signierten Nachrichten und maximal t fehlerhaften Prozessen

1 Start der Phase 1:

Der sendende Prozess P_0 signiert die Nachricht, die den Wert enthält, und sendet ihn zu den anderen n-1 Empfängerprozessen.

2 In Phase k, $1 \leq k \leq t+1$:

when P_i (i ε 1, 2, ..., n-1) empfängt eine Nachricht m mit Wert v und k Signaturen

do $V_i := V_i$ u Wert v in der Nachricht;

if k < t+1 **then**

begin signiere Nachricht m;
übermittle m zu allen Prozessen, die m noch nicht signiert haben;
end

end if
end do

3 Am Ende der Phase t+1:

für alle i ε 1, 2, ..., n-1: P_i speichert den Auswahlwert (V_i);

3.2.3 Der Dolev und Strong Algorithmus

Das Prinzip

Die Anzahl der von UM(t) benötigten Nachrichten kann in praktischen Implementationen der Hauptnachteil sein. Deshalb gibt es einen Anreiz Algorithmen zu finden, die in diesem Bereich eine kleinere Komplexität haben. Beim Beschränken der Umstände, in denen ein Prozess eine empfangene Nachricht neu übertragen muß, erhalten Dolev und Strong einen

Algorithmus mit kleinerer polynomieller Komplexität, gemessen an den benötigten Nachrichten.

Diese Reduzierung wird wie folgt erreicht. Erstens überträgt ein Prozess eine empfangene Nachricht nur neu, wenn er nicht schon vorher den in der Nachricht enthaltenen Wert übertragen hat. Folglich empfangen die anderen Prozesse, falls der sendende Prozess P_0 fehlerfrei ist, den gleichen Wert und geben ihn nur einmal an jeden der verbleibenden n-2 Prozesse weiter. Also ist die Gesamtanzahl an Nachrichten (n-1) + (n-1)*(n-2), was eher in $O(n^2)$liegt als das vorherige $O(n^{t+1})$.

Zweitens wenn P_0 fehlerhaft ist und verschiedene Werte zu bestimmten Prozessen sendet, möglicherweise einen unterschiedlichen Wert zu jedem der n-1 Prozesse, werden die Empfänger wie vorher weiterleiten. Wenn in diesem Austausch ein zuverlässiger Prozess zwei verschiedene Werte empfängt, kann er folgern, dass P_0 fehlerhaft ist, denn signierte Nachrichten können nicht verändert werden, und er wird deshalb einen default-Wert auswählen, der für alle Prozesse derselbe ist. Dadurch wird die Gesamtanzahl dieselbe wie vorher sein.

Dieser Algorithmus, den wir DS(t) nennen, basiert eigentlich auf denselben Prinzipien des Werteaustausches wie SM(t), und ist eine Verbesserung dieses Algorithmus'.

Der Algorithmus

Jeder Prozess P_i ist mit einer geordneten, initial leeren, Menge V_i ausgestattet, in der er die empfangenen Werte in der Reihenfolge der ersten Ankunft speichert.

Algorithmus DS(t)

1 Start der Phase 1:

Der sendende Prozess P_0 signiert die Nachricht, die den Wert enthält, und sendet ihn zu den anderen n-1 Prozessen. (P_0 wird als korrekt arbeitend vorausgesetzt)

2 In Phase k, $1 \leq k \leq t+1$:

when P_i ($i \in 1, 2, ..., n-1$) eine Nachricht m mit Wert v und k Signaturen empfängt

do $V_i := V_i \cup$ Wert v in der Nachricht;

if v ist einer der zwei ersten empfangenen Werte
and v wurde noch nicht von P_i übermittelt
and if $k < t+1$ **then**

begin signiere Nachricht m;
übermittle m zu allen Prozessen, die m noch nicht signiert haben;

end

end if
end do

3 Am Ende der Phase t+1:

für alle $i \in 1,2,...,n-1$:
if $|V_i| = 1$ **then** P_i speichert den Auswahlwert in V_i
else P_i speichert v_{def}
end if;

Kommentare

Die Anzahl der Phasen ist wieder t+1. Jeder Prozess sendet mindestens zwei Nachrichten zu jedem anderen, also ist die Gesamtanzahl der Nachrichten ist mindestens $2n^2$. Der von einem Prozess gespeicherte Wert ist entweder der vom sendenden Prozess P_0 gesendete Wert v, falls P_0 fehlerfrei ist (oder so von den fehlerfrei-empfangenden Prozessen angesehen wird), oder der vereinbarte Default-Wert v_{def}, falls P_0 als fehlerhaft angesehen wird.

3.2.4 Der Dolev und Reischtug Algorithmus

Dolev und Reischtug geben 5 Algorithmen an, die Konsens mittels signierten Nachrichten garantieren, das Ziel ist wieder die Anzahl der ausgetauschten Nachrichten zu reduzieren, was sie erreichen, indem sie den Ausgleich zwischen der Anzahl der Nachrichten und der Anzahl der Austauschphasen berücksichtigen. Ich gebe den ersten Algorithmus, DR(t) genannt, an.

Die Annahmen sind die folgenden:

a) abgesehen von den Signaturen trägt die Nachricht nur die Werte 0 und 1
b) das Netzwerk mit n Prozessoren mit n = 2t+1, wird in drei Subnetze aufgeteilt: der sendende Prozess und zwei Teilmengen, A und B, jeweils mit t Prozessen
c) die Struktur ist wie folgt: der sendende Prozess ist direkt mit jedem anderen Prozess verbunden und jeder Prozess aus A ist direkt mit jedem Prozess aus B verbunden und umgekehrt.

Algorithmus DR(t)

1 Start der Phase 1:

Der sendende Prozess P_0 sendet den Wert (0 oder 1) zu jedem der anderen 2t Prozesse.

2 In Phase k, $1 \leq k \leq t+2$:

when ein Prozess in Teil A (B) eine Nachricht m empfängt
if der Wert in m 1 ist und dies das erste Mal ist, dass 1 empfangen wurde
then begin signiere die Nachricht;
sende m an alle Prozesse in Teil B (A);
end;
end if;

3 Am Ende der Phase t+2:

für alle Prozesse in A oder in B:
if der Wert 1 empfangen wurde
then speichere 1 als Konsens
else speichere 0
end if;

Der Algorithmus findet nur beim Broadcast von binären Werten Anwendung, aber er kann verallgemeinert werden um mit allgemeinen Werten zu arbeiten, zum Beispiel wenn man diese allgemeinen Werte binär darstellt. Die Komplexität, gemessen an der Anzahl der Nachrichten, wird dann eine Funktion der numerischen Größe des Wertes. Es ist interessant vom Gesichtspunkt der Methode, wozu man ihn mit DS(t) vergleichen sollte und er hat praktische Bedeutung wenn binäre Werte gebroadcasted werden.

Allgemein hängen andere Algorithmen davon ab, wie die Struktur des Netzwerks angenommen wird, dessen Zuverlässigkeit, den möglichen Fehlertypen, der Möglichkeit der Austauschzustände zu handhaben, usw.

Alle bis jetzt betrachteten Algorithmen betreffen Punkt-zu-Punkt-Verbindungen. Im nächsten Abschnitt betrachte ich die Situation, wenn alle Prozessoren über einen Bus verbunden sind.

3.3 Broadcasting in einem durch einen Bus verbundenen System

3.3.1 Das Problem, die Annahmen

Die Einführung von Signaturen ermöglicht es fehlfunktionierende Prozesse, im Sinne der fehlerhaften Übertragung von Werten, zu ignorieren und nur Fehler beim übermitteln der Nachricht zu betrachten. Aber alle diese Algorithmen brauchen t+1 Phasen des Nachrichtenaustausches um gegen maximal t ausfallende Prozesse immun zu sein. Es taucht die Frage auf, ob man diese Anzahl verringern kann, wenn man die Art der Verbindung von Punkt-zu-Punkt in eine andere ändert? In einem Bussystem kann ein einzelner Basisbefehl einen Wert zu jedem anderen mit dem Bus verbundenen Prozess senden, broadcasting ist also mit einer Phase erreicht. Das Konsensproblem ist sofort gelöst, wenn der Bus und alle Prozesse und Verbindungen zuverlässig sind. Falls dies nicht der Fall ist, brauchen wir einen Algorithmus und eine Architektur, die immun gegen Fehler sind.

Im Folgenden nehmen wir an, dass ein Wertbroadcast von jedem anderen Prozess am Bus unverändert empfangen wird, für den die Busverbindung zuverlässig arbeitet. Solch ein Prozess erhält den Wert nicht, wenn seine Verbindung ausfällt oder wenn es einen Fehler im relevanten Busabschnitt gibt. Die Annahmen C1 und C2 beachtend, sind davon die Zuverlässigkeit des Busses und Verbindung zwischen Bus und Prozess abhängig. Ein Weg um die Zuverlässigkeit gegen solche Fehler sicherzustellen, ist Redundanz einzuführen, indem der Bus und die Verbindungen repliziert werden. Dazu werden k Buslinien bereitgestellt, wobei jede eine unabhängige Verbindung zu jedem Prozess hat. Diese k-Redundanz des mittels Bus verbundenen Systems ist natürlich nur ein Äquivalent zur k-Verbundenheit eines Punkt-zu-Punkt-Systems, welches Sicherheit gegen k-1 gleichzeitig ausfallenden Kanälen gibt. Seien k_1 und k_2 die maximale Anzahl ausgefallener Prozessverbindungen und Busabschnitte bezüglich eines Wertes $k > k_1 + k_2$, dann ist immer mindestens eine zuverlässige Verbindung zwischen jedem Prozesspaar gesichert.

Angenommen das Bussystem ist in Abschnitten aufgebaut, dann ist ein Basisbefehl vorgesehen, der gewährleistet, dass ein Wertbroadcast von jedem Prozess von allen Prozessen empfangen wird, die zuverlässig mit dem selben Abschnitt verbunden sind wie der sendende Prozess. Es gibt keinen Basisbefehl für das gleichzeitige Broadcasten über verschiedene Busse, solch ein Befehl muß bei Bedarf konstruiert werden.

3.3.2 Der Babaoglu und Drummond Algorithmus

Babaoglu und Drummond haben zwei Algorithmen für zuverlässiges Broadcasting unter den beschriebenen Umständen angegeben, von denen ich den ersten hier angebe. Er setzt ein zuverlässiges Bussystem voraus und der einzige Prozessfehler ist die Unfähigkeit, entweder

zeitweise oder dauerhaft, eines Prozesses oder seiner Verbindungen einen Wert zu übertragen. Fehlfunktionen aufgrund veränderter Nachrichten werden also nicht berücksichtigt. Wir haben gesehen dass dies im Fall eines zeitweisen Ausfalls reduziert werden kann, indem signierte Nachrichten verwendet werden.

Das Prinzip

Es gibt zwei Phasen. Die erste beginnt mit den k Broadcasts eines Wertes durch den sendenden Prozess, der falls er fehlerhaft ist, verschiedene Werte auf verschiedenen Linien sendet, möglicherweise auch nichts auf manchen. Sie endet mit dem Empfang durch die anderen Prozesse. In der zweiten Phase senden diese Prozesse sich gegenseitig die empfangenen Werte, dadurch können die Prozesse, die nichts empfangen haben, den gesendeten Werte folgern.

Der Algorithmus

Sei P_1 der sendende Prozess, v der Wert, den P_1 broadcastet, falls er zuverlässig ist und v_{def} der Defaultwert, der von den anderen Prozessen verwendet wird, falls P_1 als fehlerhaft beurteilt wird. Jeder Prozess P_i ist mit zwei Variablen b_1 und b_2 ausgestattet, wobei b_j die Menge von Buslinien angibt, über die P_i identische Nachrichten in Phase j empfangen hat. Der Basisbroadcastbefehl (v, c) bewirkt das Broadcasten des Wertes v über den Bus mit der Nummer c, wobei die Busse von 1 bis k nummeriert sind.

Algorithmus BD

```
1 Phase 1.

        < P1 sendet den Wert c >: für alle c ε {1, 2, …, k} broadcaste (v, c);
        < P1 speichert den Wert v >;
        < Pi (i ε 2, 3, ... , n) empfängt von 0 bis k Nachrichten, die v enthalten >

    2 Phase 2.

        für alle Pi (i ε 2, 3, .., n)
        if b1 ≠ 0 then
            begin  sei x der in Phase 1 empfangene Wert;
                   für alle c ε {1, 2, ..., k} – b1 do broadcaste (x, c);
                   end do;
                    speichere x als Konsenswert;
            end;
        end if;

    3 Am Ende der Phase 2.

        für alle Pi (i ε 2, 3, ..., n)
        if Pi hat noch keinen Wert gespeichert then
         if b2 ≠ 0 then
           begin sei y der in Phase 2 empfangene Wert; speichere y als den
                 Konsenswert;
           end;
           else speichere vdef,
         end if;
end if;
```

Kommentare

Wenn p die Anzahl der fehlerhaften Prozesse ist, gewährleistet der Algorithmus zuverlässiges Broadcasting, wenn die Gesamtanzahl der Prozesse $n > k_1 + p$. Um das zu beweisen sind vier Fälle zu betrachten:

a) der sendende Prozess P_1 ist zuverlässig,
b) P_1 ist fehlerhaft und sendet nichts,
c) P_1 ist fehlerhaft und ein zuverlässiger Prozeß empfängt v,
d) P_1 ist fehlerhaft und nur die fehlerhaften Pozesse empfangen v.

Die ersten beiden Fälle sind trivial, die Konsenswerte sind jeweils v und v_{def}

Beim dritten Fall beobachten wir, dass $k_2 = 0$, da der Bus zuverlässig ist, also $k > k_l$.
Daraus folgt, dass die Prozesse zu dem Konsenswert v gelangen.

Im vierten Fall ist die Anzahl der zuverlässigen Prozesse n – p und mit $n > k_1 + p$ ist dies größer als k_l. Der Konsenswert ist also entweder v oder v_{def}, abhängig von dem Fazit über P_1, zu dem diese Prozesse gekommen sind als Ergebnis der Informationen, die sie von den p fehlerhaften Prozessen empfangen haben.

4 Zusammenfassung

Das Konsensproblem ist von fundamentaler Bedeutung, wenn es gilt ein zuverlässiges verteiltes System, mit nicht zu garantierender Zuverlässigkeit der Komponenten, zu bauen. Abgesehen von der praktischen Bedeutung des Konsensproblems gibt es auch großes Interesse vom Gesichtspunkt der Algorithmentheorie. Erstens, hilft es die Basismechanismen zu verstehen, die in vielen verteilten Algorithmen benutzt werden. Beispielsweise die Austauschphasen, die benötigt werden, um Konsens zu erzielen: die Übertragung von Informationen zwischen Prozessen ist analog zum Verbreiten einer „Welle des Wissens“ und ermöglicht es jedem Prozeß sein Wissen zu erweitern, um dann dieses vermehrte Wissen zu anderen zu verteilen. Zweitens wirft es das Problem auf, die Annahmen bezüglich der Rechen- und Kommunikationsteilnehmer genau anzugeben. Um Ausfallsicherheit gegenüber Fehlern zu erreichen, muß man sich genau mit den Fehlern auseinandersetzen. Die Arbeiten an diesem Problem haben gezeigt, dass Fehlertoleranz, die auf Redundanz oder auf dem Mehrheitsvotum basiert, nur eine partielle Lösung darstellt, die keine endgültige Antwort auf die Frage der Zuverlässigkeit geben kann.

Folglich ist das Problem des Konsens’ ein fundamentales, für welches man viele Modelle für verteiltes Rechnen und verteiltes Überwachen ableiten kann. Die Beschäftigung mit und das Verstehen von den erreichten Lösungen sind zwingend erforderlich für das Verständnis von vielen verteilten Algorithmen.

Literaturverzeichnis

Michel Raynal: Distributed Algorithms and Protocols
John Wiley Sons, 1. Auflage, Chichester u.a. 1988

George Colouris, Jean Dollimore, Tim Kindberg: Distributed System, Conepts and Desgin
Addison Wesley, 2. Auflage, Harlow, Engalnd u.a. 1994

Andrew S. Tanenbaum, Marteen van Steen: Distributed Systems, Principles and Paradigms
Prentice Hall, 1. Auflage, London u.a., 2002