Byzantinische Fehlertoleranz durch Gruppenkommunikation am Beispiel des Rampart-Toolkit

Frank Mattauch
Frank.Mattauch@informatik.stud.uni-erlangen.de

Kurzzusammenfassung

Das Rampart-Toolkit ist ein Werkzeug, dessen Kern aus Protokollen besteht. Es soll die Entwicklung von fehlertoleranten Diensten erleichtern. Diese Protokolle befassen sich mit grundlegenden Problemen verteilter Netzwerke:

- Asynchrone Gruppenmitgliedschaft
- Zuverlässiger Multicast, auch bekannt als Byzantinische Einigung.
- Atomarer Multicast

Für die Entwicklung von fehlertoleranten Diensten wird die Technik des State Machine Replication eingesetzt. Das Rampart-Toolkit erweitert diese Technik mittels des Server Output-Votings. In diesem Dokument wird ein kurzer Überblick über das Rampart-Toolkit gegeben.

1 Einführung

Der Begriff *Rampart-Toolkit* kommt aus dem Englischen. Er setzt sich aus den Begriffen *Rampart*, was soviel wie Wehrgang, Wall oder auch Schutzschild bedeutet und *Toolkit* für Werkzeug zusammen. Rampart-Toolkit lässt sich daher als Schutzschildwerkzeug oder Werkzeug für einen Schutzschild übersetzen. Rampart war das erste System, das die Machbarkeit von Toleranz bei *zuverlässigen* und *atomaren Gruppenmulticast* gegenüber böswilligen Prozessen in lose gekoppelten, verteilten Systemen zeigte. Frühere Systeme demonstrierten dies nur auf der Basis von synchronen Netzwerken, bei denen die maximalen Übertragungszeiten von Nachrichten, die maximalen Ausführungszeiten von Prozessen und die relative Uhrendrift (*clock drift*) bekannt sind.

Viele Techniken, die zum Ausführen sicherheitskritischer Funktionen benutzt werden, basieren auf vertrauenswürdigen Diensten. Diese Dienste müssen vor Angreifern geschützt werden, so dass sie überhaupt vertrauenswürdig sein können. Die Funktionsfähigkeit und Sicherheit des Systems, das diese Dienste umgibt, kann von deren Verfügbarkeit abhängen. Aus diesem Grund müssen sie repliziert werden um hochverfügbar zu sein. Ein Problem ist, dass diese Anforderungen miteinander im Konflikt stehen. Eine Lösung dieses Problems besteht darin die Sicherheit und Verfügbarkeit derart auszubalancieren, so dass sie noch immer korrekt arbeiten und verfügbar bleiben, auch wenn ein böswilliger Angriff auf das System stattfindet.

Die am meisten angewandte Technik für fehlertolerante Systeme wird *State Machine Replication* genannt. Bei diesem Ansatz wird ein Dienst durch mehrere identische, deterministische Server implementiert, wobei jeder mit demselben Zustand initialisiert wird. Clients senden Anfragen unter der Benutzung eines *atomaren Multicastprotokolls* (siehe Kapitel

"Atomarer Multicast") an den Server. Dies stellt sicher, dass alle korrekt arbeitenden Server dieselbe Anfrage in der selben Reihenfolge bearbeiten und dadurch die gleiche Ausgabe für jede Anfrage erzeugen. Falls die Server mit richtiger Ausgabe, in ausreichender Zahl überlegen sind, dann kann ein Client die richtige Ausgabe durch *Output-Voting* herausfinden. Ein beträchtliches Hindernis State Machine Replication in feindseligen Umgebungen zu Nutzen, ist das Vertrauen in den atomaren Multicast.

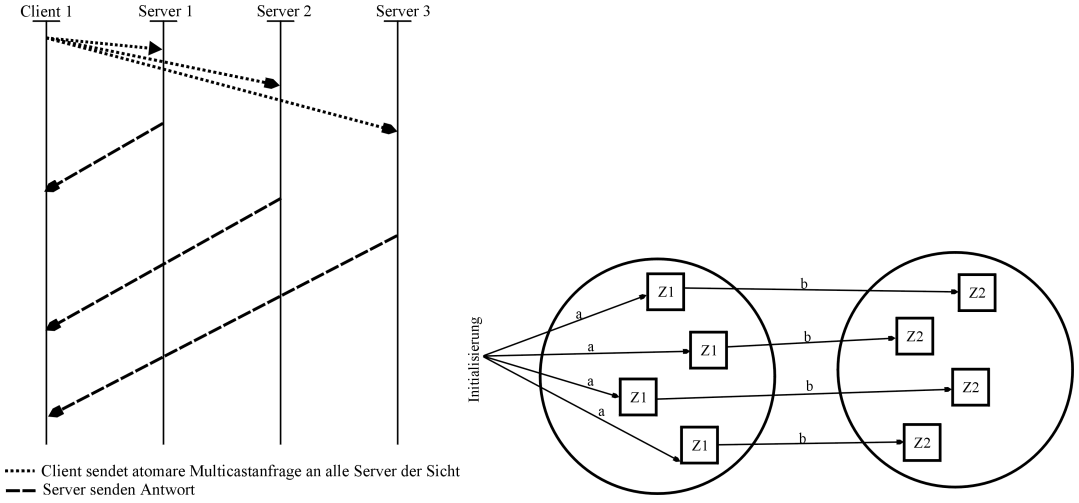


Abbildung 1: Ablauf des State Machine Replication. State Machine Replication überführt alle Server von einem Zustand in den nächsten.

2 Atomarer Multicast

Welche Aufgabe hat *atomarer*, welche hat *zuverlässiger Multicast*, der auch als *byzantinische Einigung* (Definition - siehe Anhang) bekannt ist?

- *Zuverlässiger Multicast* stellt sicher, dass alle korrekt arbeitenden *Gruppenmitglieder* trotz böswilliger Multicasts durch verfälschte Mitglieder dieselben Nachrichten liefern.
- *Atomarer Multicast* ist für die Fähigkeit zuständig, dass korrekt arbeitende Mitglieder diese Nachrichten in derselben Reihenfolge liefern.

  Es gibt in Rampart zwei atomare Multicastarten:

- Atomarer Multicast von *Servern* zu Servern
- Atomarer Multicast von *Clients* zu Servern

Atomarer Multicast von *Clients* zu Servern ist unter der Benutzung eines atomaren Multicastprotokolls von *Servern* zu Servern implementiert. Atomarer Multicast von Clients zu Servern ist also eine Erweiterung des atomaren Mulicastprotokolls von Servern zu Servern. Das letztere Protokoll wird durch zwei weitere Protokolle, dem *Gruppenmitgliedschaftsprotokoll* und einem *zuverlässigen Multicastprotokoll* implementiert.

Die Protokolle in Abbildung 3 machen folgende Annahmen zur ihrer Umgebung:

- Es werden digitale Signaturen verwendet. (In diesem Fall Verschlüsselung mittels RSA.)

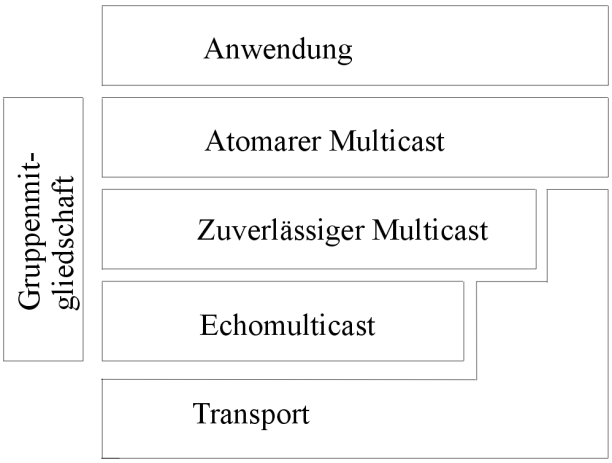


Abbildung 2:

- Alle Serverprozesse besitzen einen privaten Schlüssel, alle anderen den öffentlichen Schlüssel.
- Die erste Schlüsselverteilung kann manuell durch einen Operator erfolgen.
- Nach einem Ausfall sind die öffentlichen Schlüssel durch nichtflüchtigen Speicher oder eine Einrichtung zum Verteilen der Schlüssel wieder herstellbar.
- Der Schlüssel eines neuen Servers wird verteilt, bevor dieser in Aktion tritt.
- Es existiert eine Instanz, die den Nachrichtentransport in einem zuverlässigen, authentifizierbaren Kommunikationskanal mit Punkt-zu-Punktverbindung übernimmt.

Die Multicastimplementierung von Rampart macht keine Annahmen über die Art des Fehlers, die Server oder Clients zeigen können, da es sich hier um *Byzantinische Fehler* (Definition - siehe Anhang) handelt. Die Anzahl der Fehler muss auf einen konstanten Anteil an Servern limitiert werden. Das gesamte System ist asynchron, es werden daher keine Annahmen bzgl. maximaler Nachrichtenlaufzeit und Uhrendrift gemacht.

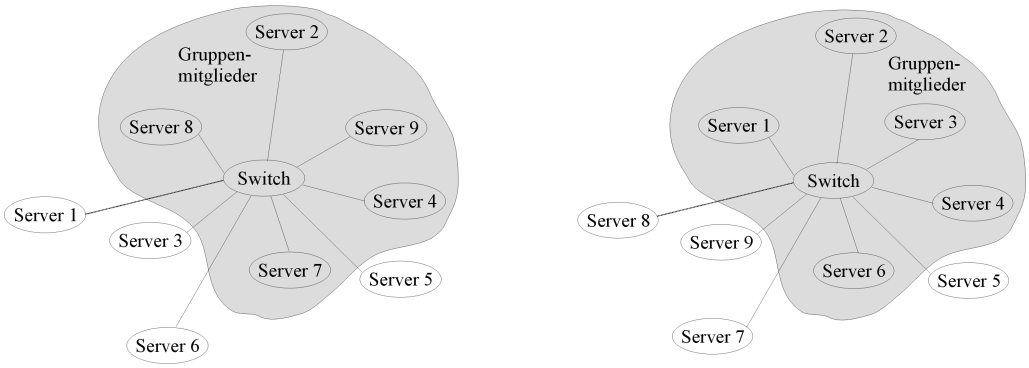
2.1 Gruppenmitgliedschaft

Die Aufgabe des *Gruppenmitgliedschaftsprotokolls* ist es, den korrekten *Gruppenmitgliedern* zu ermöglichen gemeinsam Mitglieder, die nicht während eines Multicasts reagieren, zu entfernen.

Grundlage des atomaren Multicasts ist das Gruppenmitgliedschaftsprotokoll. Es generiert eine Reihe von *Gruppensichten* (*group views*), von denen jede aus einem Vektor, in dem die Gruppenmitglieder gespeichert werden, besteht. Die Gruppensichten können sich dahingehend ändern, dass wahrgenommene Fehler oder neu hinzugefügte Server widergespiegelt werden. Das Protokoll liefert jede dieser Sichten zu allen Mitgliedern der betreffenden Sicht. Mit Hilfe eines von ihm bereitgestellten Interfaces können Prozesse zu einer Gruppe hinzugefügt oder wieder entfernt werden.

Diese Funktionen habe nur dann eine Wirkung, wenn es genügend Gruppenmitglieder gibt, die das verlangen. Eine mögliche Vorgehensweise ist z.B. das Erstellen einer neuen Gruppensicht, die diesen Server nicht mehr enthält, wenn er entfernt werden soll. Umgekehrt

Unterschiedliche Gruppensichten



Gruppensicht 1 Gruppensicht 2

Abbildung 3:

Hinzufügen eines Elements zur Gruppensicht 2 Löschen eines Elements aus der Gruppensicht 2

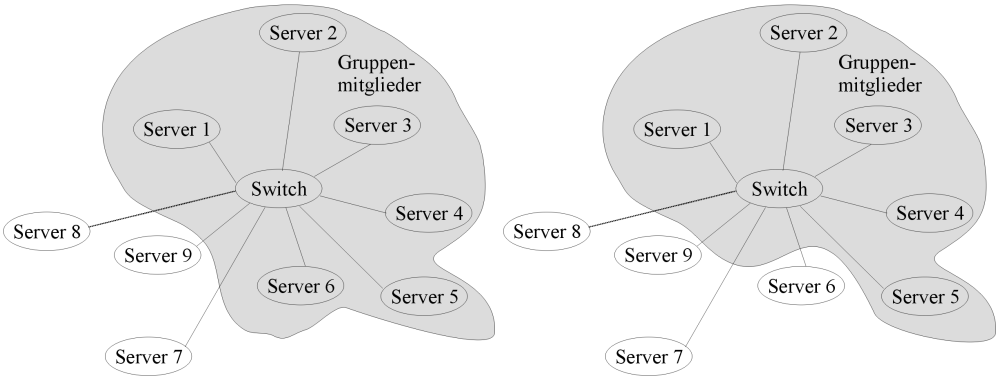


Abbildung 4:

funktioniert das Hinzufügen analog. Das Protokoll stellt sicher, dass böswillige Mitglieder auf diesem Weg keine Veränderung der Gruppenzugehörigkeit verursachen oder notwendige Veränderungen verhindern können. Die einzige Möglichkeit durch böswillige Absicht eine Veränderung der Gruppenmitgliedschaft herbeizuführen, ist ein *Denial-of-Service Angriff* (Definition - siehe Anhang), der Nachrichten zwischen korrekten Mitgliedern derart verzögert, so dass einige korrekte Gruppenmitglieder als nicht erreichbar erscheinen. In diesem Fall werden die nicht erreichbaren Mitglieder aus der Gruppe entfernt um den Fortschritt des Protokolls nicht zu gefährden.

Die Semantik, die durch das Gruppenmitgliedschaftsprotokoll bereitgestellt wird ist von der Annahme abhängig, dass weniger als ein Drittel der Mitglieder jeder Gruppe fehlerhaft sind. Dieses Protokoll ist eine Art Einigungsprotokoll, da es eine Übereinkunft über die Abfolge der Gruppensichten herbeiführt. Es legt also fest, wann welche Sicht für die Gruppe gültig ist. Dies muss allen Gruppenmitgliedern mitgeteilt werden. Es kann auch vorkommen dass es keine Einigung gibt, da sie evtl. unmöglich ist. Aus dem Protokoll wird die Beschränkung

deutlich, dass das Erstellen einer neuen Sicht nur dann garantiert werden kann, wenn ein korrektes Gruppenmitglied existiert. Die Entfernung dieses Gruppenmitgliedes darf nicht durch mehr als zwei Drittel der aktuellen Gruppenmitglieder gefordert werden, so dass es lange genug erreichbar ist. Keine Einigung wird in dem Spezialfall erzielt, dass permanent alle Mitglieder der Gruppe zu schnell wechseln und daher auch nicht lange genug zur Verfügung stehen. Geschieht dies unendlich oft, so ist eine Einigung unmöglich.

Die Änderung der Gruppenmitgliedschaft ist eine Operation die lange dauert, dafür für die meisten Anwendungen relativ selten vorkommt. Die Kosten für diese Operation werden bei der derzeitigen Implementierung durch RSA-Operationen verursacht.

2.2 Echomulticast

Echomulticast ist eine Kernkomponente des zuverlässigen und des atomaren Multicastprotokolls. Die Aufgabe des Echomulticasts besteht darin sicherzustellen, dass jeder korrekte Prozess einer Gruppensicht dieselbe Nachricht an die nächsthöhere Schicht, dem zuverlässigen Multicast, liefert.

Ablauf des Echomulticast:

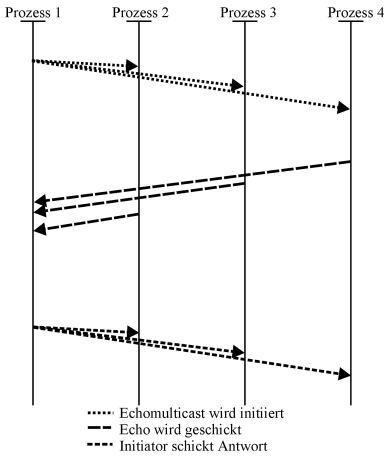


Abbildung 5:

1. Der Prozess, der den Multicast initiiert, schickt eine Nachricht zusammen mit Verwaltungsdaten als Datenpaket alle Mitglieder seiner Sicht.

2. Wenn ein Gruppenmitglied diese Nachricht vom Initiator des Echomulticasts erhält, fügt er weitere Verwaltungsdaten hinzu. Dann schickt er dieses Datenpaket signiert als ein *Echo* an den Initiator zurück. Falls das Gruppenmitglied vom Initiator weitere Nachrichten erhält, so werden diese nicht als Echo beantwortet.

3. Sobald der Initiator von mehr als zwei Drittel der Gruppenmitglieder ein Echo als Antwort auf seine Nachricht bekommen hat, sendet er diese Echos (alle Echos, die ihn erreicht haben) in einer neuen Nachricht an alle Gruppenmitglieder. Wir erinnern uns: Ein Drittel der Gruppenmitglieder darf fehlerhaft bzw. manipuliert sein.

4. Wenn ein Prozess schließlich diese gesendeten Echos erhält, führt er bei allen Echos folgende Überprüfungen durch:

- Gibt es ausreichend korrekte Signaturen?
- Ist der Prozess in derselben Sichtnummer?

Trifft alles zu wird die Nachricht an die nächsthöhere Protokollschicht geliefert. Ist eine Bedingung nicht erfüllt, passiert nichts.

Aus Effizienzgründen wird beim Echomulticast nicht die Nachricht selbst, sondern nur eine Hashwert der Nachricht geschickt. Dadurch, dass alle Signaturen geprüft werden, wird sichergestellt, dass alle Prozesse dieselbe Nachricht erhalten haben und weitergeben können.

2.3 Zuverlässiger Multicast

Zuverlässiger Multicast wird mit Hilfe des Gruppenmitgliedschaftsprotokolls und des Echomulticasts implementiert. Das zuverlässige Gruppenmulticastprotokoll stellt eine Schnittstelle bereit, über die Gruppenmitglieder Multicastnachrichten an ihre Servergruppe schicken können. Das Protokoll liefert eine Folge von Nachrichten an jedes Gruppenmitglied, wobei jede Nachricht entweder eine zuverlässige Multicastnachricht eines Gruppenmitglieds oder eine lokal vom Gruppenmitgliedschaftsprotokoll empfangene Gruppensicht ist. Das zuverlässige Multicastprotokoll liefert Gruppensichten zu jedem korrekten Mitglied in der Reihenfolge, wie sie vom Mitgliedschaftsprotokoll empfangen wurden. Die Reihenfolge der Nachrichten, die an die Server ausgeliefert wird, ist dieselbe für jeden Server, der Mitglied der Gruppe ist.

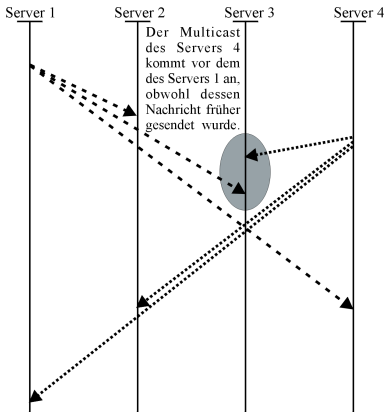


Abbildung 6:

Annahmen, auf denen das zuverlässige Multicastprotokoll basiert: Auf Grund der Verwendung des Mitgliedschaftsprotokolls und anderer Details des Multicastprotokolls selbst sind weniger als ein Drittel der Mitglieder jeder Gruppensicht fehlerhaft. In einigen Fällen ist der Fortschritt des zuverlässigen Multicastprotokolls auf die Entfernung eines Mitlieds aus der Gruppe angewiesen. Deshalb ist der garantierte Fortschritt unseres zuverlässigen Multicastprotokolls abhängig vom garantierten Fortschritt des Gruppenmitgliedschaftsprotokolls.

Wie funktioniert zuverlässiger Multicast? Falls es keine Änderung der Gruppenzugehörigkeit gibt, dann wird der zuverlässige Multicast durch einen einzelnen Echomulticast implementiert:

- Falls ein Prozess mittels zuverlässigem Multicast eine Nachricht innerhalb seiner Sicht an die Gruppe schicken möchte, dann führt er einen zuverlässigen Echomulticast an seine Gruppensicht aus.

- Falls ein Prozess eine Nachricht von der Schicht des Echomulticast bekommt, so gibt er sie an die nächsthöhere Schicht, den atomaren Multicast weiter.

Für den Fall, dass sich Änderungen der Gruppenmitgliedschaft ergeben, wird alles etwas komplizierter:

Betrachten wir den Fall, dass sich nur ein Sichtwechsel ergibt. Sobald die neue Gruppensicht von einem Prozess p empfangen wird, der in der alten sowie in der neuen Gruppensicht enthalten ist, unterlässt dieser Prozess p neue zuverlässige Multicasts. Außerdem schickt er eine spezielle Nachricht "*Ende*" über Echomulticast an die alte Gruppensicht. Dadurch wird bekannt gegeben, dass der Prozess p in der alten Sicht nicht mehr für zuverlässige Multicasts zur Verfügung steht. Der Prozess p wartet nun, bis er von allen eine "Ende"-Nachricht bekommen hat. Anschließend werden vom Prozess p alle seine bisher empfangenen Nachrichten, die möglicherweise noch nicht überall angekommen sind, samt deren Verwaltungsdaten (von den Echomulticasts) als *"Flush"-Nachricht* an alle Prozesse der Gruppensicht über Echomulticast verteilt. Der Prozess wartet nun wie bei den "Ende"-Nachrichten, darauf, dass er von allen anderen Prozessen der Gruppensicht ebenfalls "Flush"-Nachrichten erhält. Danach wird die neue Sicht an die atomare Multicastschicht weitergereicht und der Prozess p nimmt den zuverlässigen Multicast wieder auf.

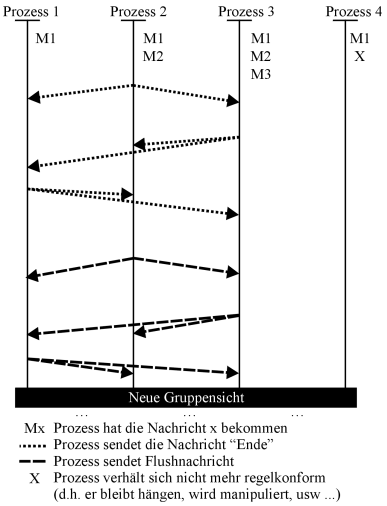


Abbildung 7:

Auf diese Weise ist eine Änderung der Gruppenmitgliedschaft für ein einzelnes Mitglied möglich. Manchmal ist es aber notwendig während der Ausführung des obigen Protokolls Änderungen der Mitgliedschaft durchzuführen, damit noch ein Fortschritt erzielt werden kann. Ein solcher Fall ist z.B. wenn von einem Prozess nie eine "Ende"- oder "Flush"-Nachricht empfangen wird. Dieser Prozess muss dann aus der Gruppe entfernt werden.

Hierzu verfährt man wie folgt:

Um Prozesse entfernen zu können, dürfen nach dem Empfang einer neuen Sicht keine neuen Prozesse mehr aufgenommen werden. Dies geschieht solange, bis die neue Gruppensicht an die nächsthöhere Schicht des zuverlässigen Multicastprotokolls weitergereicht wird. Währendessen können korrekte Prozesse nicht reagierende Mitglieder entfernen und dadurch neue Sichten erstellen. Für jede neu empfangene Sicht wird dieses Protokoll ausgeführt. Dieser Vorgang stoppt, wenn eine bestimmte Gruppensicht empfangen wird. Sie muss die Eigenschaft haben, dass es möglich ist Teile der Verwaltungsinformation des Echomulticastprotokolls an

dessen nächsthöhere Schicht zu liefern. Die Information die dabei geliefert wird stammt von jedem Mitglied, das in jeder der bisherigen Sichten während des obigen Vorgangs enthalten war. Schließlich kann dann die anfangs neue empfangene Sicht x' an die nächsthöhere Schicht des zuverlässigen Multicastprotokolls geliefert werden.

Da atomarer Multicast unter der Benutzung von zuverlässigem Gruppenmulticast implementiert ist, ist es entscheidend, dass zuverlässiger Multicast schnell funktioniert. Wie bei dem Gruppenmitgliedschaftsprotokoll werden die Kosten beim zuverlässigen Multicast durch RSA-Operationen verursacht. Um diese Kosten zu senken werden kryptografische Schlüssel mit zeitlich begrenzter Gültigkeit verwendet, die nur bedingt sicher sind, sich aber schneller verarbeiten lassen. Um dise Schwachstelle auszubessern werden Protokolle verwendet, die diese Schlüssel regelmäßig auswechseln.

2.4 Atomarer Multicast

Atomarer Gruppenmulticast wird mit Hilfe des zuverlässigen Gruppenmulticastprotokoll implementiert. Wie der zuverlässige Multicast stellt der atomare Multicast eine Schnittstelle für die Mitglieder bereit um Multicastnachrichten an die Gruppe zu schicken. Er liefert die Ereignisse in einer eindeutigen Reihenfolge an jedes korrekte Mitglied, wobei jedes Ereignis eine Gruppensicht oder eine Nachricht ist.

Wie funktioniert atomarer Multicast? Beim atomaren Multicast existiert in jeder Sicht ein ausgewähltes Mitglied einer Gruppe, das *Sequencer* (= Ablaufsteuerung) genannt wird. Dieser legt die Reihenfolge der Nachrichten fest, in der sie an die Anwendung weitergegeben werden. Der Sequencer in der jeweiligen Sicht wird durch einen beliebigen deterministischen Algorithmus ausgewählt. Eine Möglichkeit ist zum Beispiel, dass man das Mitglied mit der lexikografisch kleinsten Bezeichnung wählt.

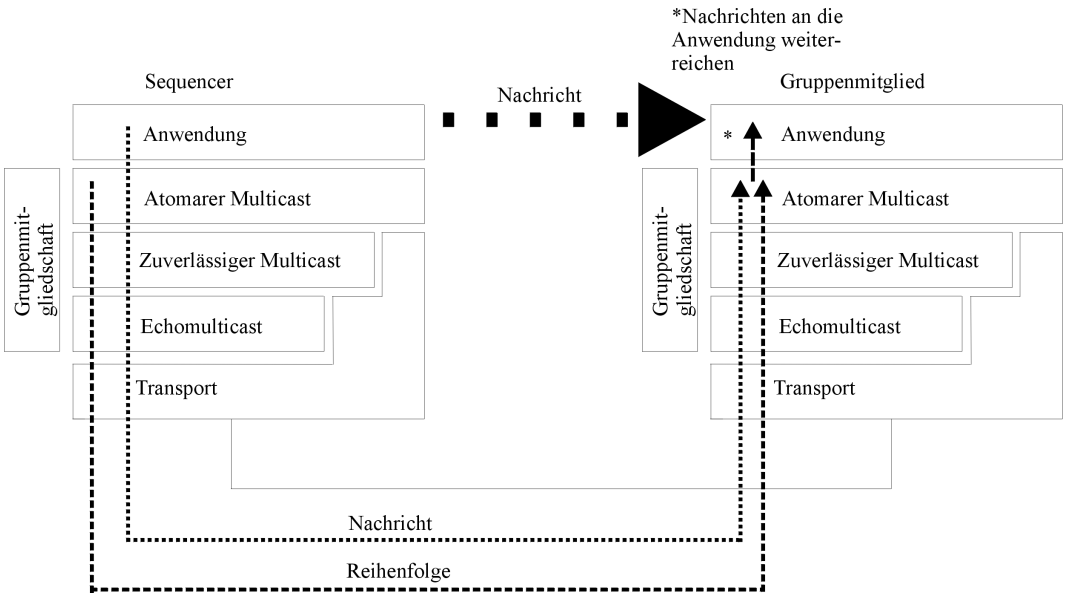


Abbildung 8:

Will nun ein Prozess eine Nachricht mit atomaren Multicast verschicken, so wird dies mit der Hilfe des zuverlässigen Multicastprotokolls erledigt. Sobald der Sequencer die Nachricht

empfängt, bestimmt er eine Reihenfolge, in der die Nachrichten an die Anwendung weitergereicht werden sollen. Dies gibt er der Gruppe bekannt, indem er eine spezille Nachricht, die *Reihenfolge-Nachricht*, über zuverlässigen Multicast sendet.

Betrachten wir den Fall, dass an die atomare Multicastschicht eine neue Sicht geliefert wird. Es könnte durchaus noch einige Nachrichten geben, die noch in der alten Sicht, aber ohne Angabe der Reihenfolge an die atomare Multicastschicht geliefert wurden. Es ist daher die Reihenfolge unbekannt, in der sie an die Anwendung weitergereicht werden sollen. Diese noch nicht weitergeleitete Nachrichtenmenge ist bei allen korrekten Prozessen der Gruppe garantiert gleich. Aus diesem Grund können die verbleibenden Nachrichten in jeder deterministischen Reihenfolge an die Anwendung geliefert werden. Sobald alle ungeordneten Nachrichten, die in der alten Sicht der atomaren Multicastschicht zugestellt wurden, der Anwendung übergeben sind, kann die neue Sicht der Anwendung weitergereicht werden.

Es werden noch weitere Schritte benötigt, um sicherzustellen, dass ein manipulierter Sequencer nicht das Weiterreichen einer Nachricht an die Anwendung verhindern kann. Der Sequencer könnte sich einerseits weigern anzugeben, wann die Nachricht der Anwendung übergeben werden sollen, andererseits kann er Nachrichten in seine Reihenfolge-Nachricht aufnehmen, die gar nicht existieren. Im letzten Fall würden alle Prozesse versuchen die nicht existierende Nachricht zu bekommen, was unmöglich ist. Die Nachrichten, die in der Reihenfolge-Nachricht später zur Lieferung anstehen würden nie zugestellt werden, da sie auf die nicht existierende Nachricht warten müssen.

Einem solchen Fehlverhalten kann man entgegenwirken: Falls ein korrekter Prozess p eine Nachricht nicht innerhalb einer festgelegten Zeitspanne an die nächsthöhere Schicht, die Anwendung, weiterreicht, nachdem er sie an die atomare Multicastschicht geliefert hat, dann verlangt Prozess p, dass der Sequencer von der Gruppe entfernt wird. Diese Forderung ist gerechtfertigt, da der Sequencer bald nachdem er eine Nachricht bekommen und weitergeleitet hat, eine Reihenfolge-Nachricht schickt.

Sobald der Prozess p eine neue Sicht an die atomare Multicastschicht liefert, kann er jede beliebige nicht an die Anwendung weitergeleitete Nachricht wie vorher behandeln. Also z.B. einfaches Weiterreichen der Nachrichten in irgendeiner deterministischen Reihenfolge an die Anwendung.

Wenn das Mitgliedschafts-Protokoll keine neue Sicht erzeugt, der Sequencer funktioniert also korrekt, dann ist sichergestellt, dass der Prozess p die fehlenden Nachrichten von der zuverlässigen Multicastschicht an die atomare Multicastschicht schickt. Die fehlenden Nachrichten können eine Reihenfolge-Nachricht für bereits erhaltene Benutzernachrichten sein, die wegen noch fehlender Angabe der Reihenfolge nicht an die Anwendung ausgeliefert werden können, oder noch ausstehende Nachrichten sein, die noch vor anderen, bereits vorhandenen Nachrichten, für die es schon eine festgelegt Reihenfolge gibt ausgeliefert werden müssen.

2.5 Clients

In ihrer derzeitigen Implementierung schicken Clients Anfragen an den Dienst, indem sie sie an einen einzelnen Server schicken, der die Anfrage mittels atomaren Multicast an die Servergruppe weiterleitet. Dieser Ansatz hat einige attraktive Eigenschaften. Erstens muss der Client nicht wissen wo sich jeder der Server befindet. Das bedeutet, dass der Client nur in der Lage sein muss einen der korrekten Server, die er lokalisieren kann, zu erreichen. Dies kann z.B. durch das Broadcasten einer Frage geschehen, auf die die Server antworten, oder indem er von irgendjemanden einen Hinweis auf die Adresse eines Servers bekommt, wobei dies nicht notwendigerweise eine vertauenswürdige Quelle sein muss. Zweitens ist dieser Ansatz sehr effizient, besonders, da ein Server viele Anfragen von Clients an die Servergruppe in einem einzigen atomaren Multicast weiterleiten kann.

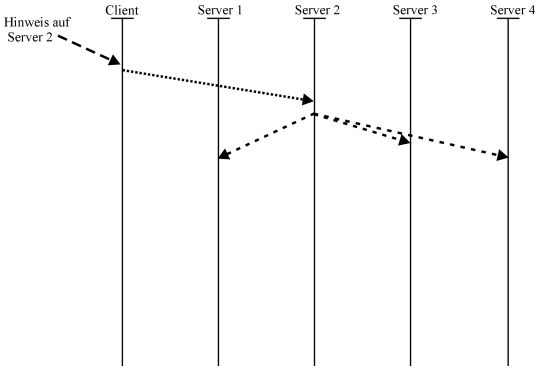


Abbildung 9:

Das Hauptrisiko dieses Ansatzes ist, dass ein Server, an den ein Client seine Nachfrage schickt, manipuliert sein könnte. In diesem Fall könnte dem Client der Dienst verweigert werden oder der Server könnte unbemerkt die Anfrage des Clients verändern, während er die Nachricht weiterschickt. Der letztere Angriff kann von den korrekten Servern entdeckt werden, indem sie die Anfrage des Clients unter der Benutzung von kryptografischen Standardtechniken auf der Anwendungsschicht authentifizieren. Falls ein Dienst verweigert wird, dann muss der Client einen anderen Server finden, der auf seine Anfrage antwortet.

3 Output Voting

Server die für Anwendungen zuständig sind schicken ihre Antworten mit Hilfe von *Output-Voting-Protokollen* zu den Clients. Dies stellt sicher, das die Antworten, die den Clients zugestellt werden nur solche sind, die durch korrekte Server erzeugt wurden. In Rampart sind zwei Output-Voting-Protokolle implementiert. Im ersten wird die Antwort jedes Servers zu dem Client geschickt, der die Anfrage gestellt hat. Der Client führt dann auf die übliche Art und Weise das *Output-Voting* durch. Ein Nachteil dieses Ansatzes ist, dass der Client in der Lage sein muss jeden Server selbst zu identifizieren und zu authentifizieren. Wichtig hierbei ist, dass die Menge von Server, die der Client identifizieren muss, eine andere ist als die potentiell dynamischere Servergruppe, die durch das Gruppenmitgliedschaftsprotokoll zum Zwecke des zuverlässigen und atomaren Multicasts definiert wurde, diese aber umfasst.

Bei dem zweiten in Rampart implementierten Protokoll wird das Output-Voting bei den Servern durchgeführt. Es ist daher transparent für die Clients. Das bedeutet für einen Client, der die Antwort eines Dienstes mit diesem Protokoll verifizieren will, dass er nicht wissen muss, wie viele Server es gibt oder wie er die einzelnen Server authentifizieren kann. Er besitzt nur einen öffentlichen Schlüssel für den Dienst, wobei dieser nicht zu verwechseln ist mit den öffentlichen Schlüsseln der einzelnen Server. Mit diesem Schlüssel muss er nur eine Antwort vom Dienst verifizieren. Der Aufwand für den Client ist eben gerade so groß, wie wenn der Dienst nicht repliziert worden wäre. Dies wird durch kryptografische Methoden erreicht, mit denen der private Schlüssel, der zum öffentlichen Schlüssel des Dienstes gehört, unter den Servern "aufgeteilt" werden kann. So wird die Kooperation von einer ausreichenden Anzahl an Servern benötigt um eine Antwort zu signieren. Dieses Protokoll bietet Vorteile gegenüber dem üblichen Ansatz des Output-Votings, falls es keinen vertrauenswürdigen Weg für Clients gibt einen einzelnen Server zu identifizieren oder authentifizieren. Der Nachteil dieses letztern Ansatzes ist, dass die kryptografischen Techniken, die von ihm eingesetzt werden, kostenin-

tensiv sind: Falls der öffentliche RSA-Schlüssel des Dienstes ein 512 Bit Modul beinhaltet, dann dauert (mit der Technik von 1995) die Latenzzeit des Server-Output-Votings mit vier Servern ungefähr 200 ms. Außerdem kann dieser Schlüssel nicht regelmäßig ausgewechselt werden um seine Größe zu reduzieren. Dies würde die Clients zwingen sich immer die neuesten Schlüssel zu beschaffen. Dadurch erfordert dieser Ansatz des Output-Votings für viele Anwendungen evtl. Spezialhardware bei den Servern um die RSA-Operationen auszuführen. Falls keine solche Hardware vorhanden ist, sollten Standardtechniken des Output-Votings verwendet werden.

4 Anhang

Byzantinische Armee bzw. Generäle

Diese Konstellation von Prozessoren resultiert aus der Analogie der Situation einer Byzantinischen Armee, die bestehend aus mehreren räumlich getrennten Divisionen einen gemeinsamen Schlachtplan entwerfen muss. Jede Division wird von einem General geführt. Die Generäle können untereinander nur durch den Austausch von Botschaftern kommunizieren. Eine Zusammenkunft aller zur Beratung eines Plans ist aus strategischen Gründen ausgeschlossen. Eine Entscheidung (etwa sofortiger Angriff oder nicht) muss daher von jedem General aus den Vorschlägen seiner Kollegen nach einem vorher festgelegten Verfahren vor Ort getroffen werden. Das Problem dabei ist, dass einige wenige eventuell nicht loyale Generäle, die heimlich mit dem Feind paktieren, durch unterschiedliche Vorschläge an die anderen Generäle eine einmündige Entscheidung der übrigen verhindern können. Weiter unterscheidet man noch zwischen Nachrichten mit Authentisierung (jeder Prozessor kann seine Nachrichten mit einer Unterschrift versehen, so dass Nachrichten von anderen (fehlerhaften) Prozessoren nicht unbemerkt erzeugt oder verändert werden können) und ohne Authentisierung. Werden nämlich sämtliche Nachrichten authentisiert, zum Beispiel dadurch, dass mit Hilfe eines kryptografischen Public-Key Verfahrens eine fälschungssichere digitale Unterschrift erzeugt wird, so kann die Urheberschaft einer Nachricht verifiziert werden.

Byzantinische Einigung

Ein Problem der theoretischen Informatik der nebenläufigen Prozessoren und der kommunikativen Prozessverarbeitung. Die Benennung des Problems bezieht sich auf das Paradox der Byzantinischen Generäle(Armee). Die Byzantinische Einigung bezeichnet eine Methode der Übereinkunft bei Nachrichten von Prozessen aus verschiedenen Systemen. Das Ergebnis soll ein gemeinsamer Beschluss (z.B. über den Wert einer Variablen bzw. über die Entscheidung eines Angriffs der Armee) sein. Prozesse, die Fehler enthalten und gestörte Kommunikationsverbindungen können dabei Nachrichten unterdrücken oder verfälschen. Die zur Erzielung einer Byzantinischen Einigung notwenigen Algorithmen sind abhängig von den jeweiligen Konstitution der vorliegenden Rechnernetze. Die Byzantinische Einigung ist z. B. für ein voll vermaschtes Netz mit n Knoten erreichbar, wenn die Anzahl der fehlerhaften Prozessse oder Kommunikationsverbindungen kleiner als n/3 ist.

Byzantinische Fehler

Diese Art von Fehlern bei mehreren Prozessoren in einem Rechner treten auf, wenn die sogenannte Fail-Stop Eigenschaft nicht erfüllt ist. Ein Prozessor ist ausgefallen, aber er verhält sich nicht, wie gefordert völlig ruhig, sondern produziert weiterhin Nachrichten, die ins Netz

fließen und andere Rechner erreichen. Ein fehlerhafter Prozessor kann beliebige falsche Nachrichten generieren oder Nachrichten anderer Prozessoren, die er als Relaisstation übermittelt, verändern oder erfinden. Man bezeichnet derart auftretende Fehler als Byzantinische Fehler.

Denial of Service Angriff

Mit Denial of Service Angriffen meint man, einen Server/Host zu blockieren oder vom Netzwerk abzuschneiden, so dass er keine Anfragen mehr bearbeiten kann. Dies kann durch Ausnutzung eines Fehlers in der Software oder durch bloße Überlastung erfolgen.

Der allererste (bedeutende) DoS-Angriff war der Morris-Wurm im Jahr 1988. Etwa 5000 Rechner waren für mehrere Stunden betriebsunfähig.

Funktionsweise der DoS-Angriffe:

- Das Ziel eines DoS-Angriffs ist klar: einen Server/Host/User aus dem Netz kicken oder zu blockieren. Anhand einiger Beispiele will ich nun die Funktionsweise erklären:

- Beispiel 1: Der OOB-Angriff

  Er ist sehr simpel, was allerdings mit der Microsoft Netbios-Implementierung unter Windows 95 zusammenhängt. Wenn man an den Port 139 ein paar unsinnige Zeichen oder Daten schickt, dann führt das schon zu einem "Blue Screen of death". Diese Methode funktioniert allerdings nur mit Windows 95; bei Windows 98, ME, NT oder 2000 hat sie keine Wirkung.

- Beispiel 2: Ping-Flood

  Ping-Flood ist noch einfacher als OOB, setzt allerdings einen schnellen Internetzugang voraus. Man macht einfach möglichst viele Ping-Anfragen in einer möglichst kleinen Zeit. Der Zielrechner versucht alle zu bearbeiten, was zu einem erheblichen Performanceverlust oder gar zum Crash führen kann. Wie gesagt, man sollte eine gute Internetanbindung haben (auf jedem Fall schneller als die des Opfers, z.B. SDSL oder eine Standleitung).

Literatur

[SAP] Michael K. Reiter, "Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart", AT&T Bell Laboratories, Holmdel, New Jersey, USA, 1995

[Ramp] Michael K. Reiter, "The Rampart Toolkit for Building High-Integrity Services", AT&T Bell Laboratories, Holmdel, New Jersey, USA, 1995

[Def] http://en.os2.org/tips/glossary/?section=BXS

[DoS] http://www.securitypages.de/dos_s.php