Totem und SecureRing: Fehlertoleranter Multicast als Basis für fehlertolerante Anwendungen

Marc Pullmann

simapull@informatik.stud.uni-erlangen.de

Kurzzusammenfassung

Dieses Dokument stellt zwei Protokolle zum geordneten fehlertoleranten Nachrichtenaustausch in asynchronen verteilten Systemen vor. Die beiden Systeme, Totem und SecureRing, bilden damit einen Basisdienst für verteilte Anwendungen. Während Totem Resistenz gegen Netzwerkpartitionierung und nicht-byzantinische Fehler bietet, kann SecureRing auch bösartiges Verhalten einzelner Knoten erkennen.

1 Totem

1.1 Einführung, Systemmodell und Dienste

1.1.1 Umgebung

Totem modelliert ein Kommunikationsnetz als endliche Zahl von Broadcast-Domänen, die durch Gateways verbunden sind. Eine Broadcast-Domäne besteht aus einer endlichen Zahl von Prozessoren mit eindeutigen IDs. Kommunikation zwischen Prozessoren einer Domäne erfolgt durch Broadcasting von Nachrichten. Auf jeder Broadcast-Domäne ist ein logischer Token-Ring aufgesetzt, das Token regelt den Zugriff auf das Kommunikationsmedium, indem es als Punkt-zu-Punkt Nachricht zirkuliert. Ein Prozessor muss das Token besitzen, um eine Nachricht broadcasten zu können. Jede Nachricht enthält eine Sequenznummer, die aus einem Feld des Tokens gewonnen wird – die Sequenznummern steigen monoton an. Ein Prozessor empfängt auch seine eigenen Nachrichten. Jeder Ring hat einen „*Representative*“ (einer der Prozessoren) und eine ID bestehend aus einer Ring-Sequenznummer und der ID des Representative.

Die Kommunikation innerhalb einer Broadcast-Domäne regelt das Single-Ring-Protokoll, das hier genauer behandelt wird. Mehrere Broadcast-Domänen kommunizieren über das übergeordnete Multiple-Ring-Protokoll.

Durch Sequenznummern in den Nachrichten wird der Dienst zuverlässig und durch Zeitstempel können die Nachrichten geordnet werden.

Totale Ordnung: Alle Nachrichten werden geordnet unter Berücksichtigung *aller* anderen Nachrichten.

Nachrichtenauslieferung in Reihenfolge der Sequenznummern heißt „*agreed delivery*“, bei „*safe delivery*“ wird ein zusätzliches Feld im Token benutzt, um festzustellen, wann alle Prozessoren des Rings eine Nachricht erhalten haben.

Auf dem Protokollstack ist Totem zwischen Netzwerk- und Anwendungsschicht angesiedelt.

Das System ist auf nichtflüchtigen Speicher in jedem Prozessor angewiesen: Gespeichert werden je Prozessor seine Ring-Sequenznummer (um die Einzigartigkeit dieser Nummer und damit der Ring-ID zu sichern) und der aktuelle Zeitstempel von Nachrichten (um

sicherzustellen, dass nach einem Ausfall des Prozessors größere Zeitstempel als bei jeder vorherigen Nachricht dieses Prozessors verwendet.

Eine *Konfiguration* ist eine einzelne Sicht auf das System anhand der Mitgliedschaftsverhältnisse bzw. der Netzwerktopologie, wie sie der Anwendung dargestellt wird. Die *Mitgliedschaft (Membership)* ist eine Menge von Prozessor-IDs.

Begrifflichkeit: Eine Nachricht wird von der nächst niedrigeren Schicht des Protokollstacks *empfangen (receive)* und an die nächst höhere Schicht (hier die Anwendung) *ausgeliefert (deliver)*. Die Auslieferung kann verzögert stattfinden werden, um die korrekte Ordnung von Nachrichten herzustellen. Das *Erzeugen (originate)* einer Nachricht geschieht durch die Anwendung, wenn diese die Nachricht erstmals broadcastet.

Es gibt zwei Nachrichtentypen, die an die Anwendung geliefert werden: *Reguläre Nachrichten* kommen von der (verteilten) Anwendung und sind zur Auslieferung an die Anwendung bestimmt. *Configuration Change-Nachrichten* machen Änderungen der Mitgliedschaft bekannt, sie beenden eine alte Konfiguration und initiieren eine neue.

Das Netzwerk und die Prozessoren sind *asynchron* und bieten (unzuverlässige) Fehlererkennung durch Timeouts. Das Protokoll führt eine Flusskontrolle durch, so dass keine Nachrichten von Prozessoren verworfen werden.

1.1.2 Fehlermodell

Durch Nachrichtenverlust kann es vorkommen, dass Nachrichten nicht von allen Prozessoren empfangen werden; Neuübertragung ist möglich.

Die Prozessoren können ausfallen, langsam arbeiten (Überlastung) oder Anforderungen auslassen. Eine Prozessor-ID bleibt nach Fehlern und Neustart gleich. *Es gibt keine bösartigen (byzantinischen) Fehler.*

Das Netzwerk kann partitioniert werden, zuverlässige Nachrichtenübermittlung aber kann nur zwischen Prozessoren einer Partition geboten werden. Später kann die Kommunikation getrennter Komponenten wiederhergestellt werden. Das Kommunikationsmedium kann Nachrichten verwerfen und ungültige Nachrichten erkennen.

Prozessorausfälle oder Netzwerkpartitionierung bewirken, dass alle Kopien des Tokens verloren gehen, da ein ausgefallener oder abgetrennter Prozessor das Token nicht weitergeben kann. Der Verlust aller Kopien des Tokens löst den Membership-Algorithmus aus, der einen neuen Token-Ring bildet.

1.2 Total Ordering-Algorithmus

1.2.1 Allgemeines

Voraussetzungen dafür, dass der Total Ordering-Algorithmus läuft: Das Token geht nie verloren, es gibt keine Prozessorausfälle, der Ring wird nicht partitioniert, aber Nachrichten können verloren gehen. Die anderen möglichen Fehler müssen vom Membership-Algorithmus behandelt werden.

Eine reguläre Nachricht besteht aus folgenden Feldern:

- *sender_id*: ID des Prozessors, der die Nachricht erzeugt hat
- *ring_id*: ID des Rings, auf dem die Nachricht erzeugt wurde (besteht aus der ID des Representative und der Ring-Sequenznummer)
- *seq*: Sequenznummer der Nachricht
- *conf_id*: 0

- *contents*: Nutzdaten/Inhalt der Nachricht

ring_id, seq und conf_id bilden die Nachrichten-ID.

Ein reguläres Token besteht aus folgenden Feldern:

- *type*: „Regular“
- *ring_id*: ID des Rings, auf dem das Token zirkuliert (besteht aus der ID des Representative und der Ring-Sequenznummer)
- *token_seq*: Sequenznummer, die die Erkennung von redundanten Kopien des Tokens erlaubt
- *seq*: größte Sequenznummer der Nachrichten, die bisher auf dem Ring gebroadcastet wurden
- *aru (all-received-up-to)*: Sequenznummer zur Bestimmung, welche Nachrichten die Prozessoren des Rings empfangen haben; damit wird das Verwerfen von Nachrichten kontrolliert, die alle Prozessoren schon empfangen haben
- *aru_id*: ID des Prozessors, der *aru* auf einen Wert kleiner als *seq* gesetzt hat; damit wird ein Prozessor erkannt, der Fehler beim Empfang aufweist
- *rtr (retransmission request list)*: enthält eine oder mehrere Anforderungen für Neuübertragung

Folgende Werte werden von allen Prozessoren lokal aufbewahrt:

- *my_aru*: Sequenznummer der Nachricht, bis zu der der Prozessor alle Nachrichten empfangen hat (alle Sequenznummern kleiner oder gleich *my_aru*)
- *my_token_seq*: Wert der von *token_seq*, als der Prozessor das Token zuletzt weitergereicht hat
- *my_high_delivered*: Sequenznummer der letzten (an die Anwendung) ausgelieferten Nachricht

Alle drei Werte werden bei der Bildung des Rings auf 0 initialisiert; *my_aru* wird beim Empfang von Nachrichten aktualisiert, *my_token_seq* beim Empfang des Tokens und *my_high_delivered* beim Ausliefern von Nachrichten.

1.2.2 Empfang des Tokens

Beim Empfang des Tokens schließt ein Prozessor die Verarbeitung aller Nachrichten in seiner Eingangswarteschlange ab, so dass diese bei Beginn eines neuen Token-Umlaufs leer ist. Danach broadcastet er angeforderte Neuübertragungen (die im Token vermerkt sind) und neue Nachrichten, aktualisiert das Token und schickt dieses an den nächsten Prozessor im Ring. Für jede Nachricht, die ein Prozessor broadcastet, inkrementiert er das *seq*-Feld des Tokens und setzt als Sequenznummer der Nachricht diesen Wert ein.

Immer wenn ein Prozessor das Token empfängt, vergleicht er das *aru*-Feld mit *my_aru*; wenn *my_aru* kleiner ist, ersetzt er *aru* durch *my_aru* und trägt als *aru_id* seine Prozessor-ID ein.
Wenn beim Empfang des Tokens das *aru_id*-Feld genau der Prozessor-ID entspricht, dann wird *aru* auf *my_aru* gesetzt.
Wenn *seq* und *aru* gleich sind, werden *aru* und *my_aru* immer zusammen mit *seq* erhöht und dem *aru_id*-Feld ein negativer Wert zugewiesen (eine ungültige Prozessor-ID).

Wenn das *seq*-Feld des Tokens größer ist als *my_aru*, dann gibt es Nachrichten, die dieser Prozessor nicht empfangen hat; in Folge dessen setzt oder vergrößert er das *rtr*-Feld. Wenn ein Prozessor Nachrichten empfangen hat, die im *rtr*-Feld vorkommen, dann überträgt er diese neu (also auch wenn sie nicht von ihm selbst erzeugt wurden!), bevor er aktuelle Nachrichten broadcastet, und entfernt die Sequenznummer(n) dieser neu übertragenen Nachricht(en) aus dem *rtr*-Feld.

1.2.3 Empfang einer Nachricht

Wenn ein Prozessor eine Nachricht *m* empfangen hat, wenn er alle Nachrichten mit Sequenznummern kleiner oder gleich der von *m* empfangen und ausgeliefert hat, und wenn der Erzeuger von *m* „agreed delivery“ vorgesehen hat, dann liefert der Prozessor *m* in „agreed order“ an die Anwendung aus.

Wenn der Prozessor zusätzlich in zwei aufeinander folgenden Runden das Token mit einer *aru* größer oder gleich der Sequenznummer von *m* weitergegeben hat, [und wenn der Erzeuger von *m* „safe delivery“ vorgesehen hat,] dann kann *m* in „safe order“ ausgeliefert werden. In diesem Fall muss die Nachricht nicht mehr länger für eventuelle Neuübertragungen aufgehoben werden.
Der Total Ordering-Algorithmus kann nicht fortfahren, wenn das Token verloren geht; um die Wahrscheinlichkeit eines Token-Verlusts zu verkleinern, wurde die Neuübertragung des Tokens zusätzlich eingeführt. Immer wenn ein Prozessor das Token weitergibt, setzt er ein Timeout für die Neuübertragung des Tokens. Er bricht dieses Timeout ab, sobald er eine reguläre Nachricht (das deutet darauf hin, dass das Token nicht verloren ging) oder wieder das Token empfängt. Wenn das Timeout abläuft, wird das Token erneut an den nächsten Prozessor im Ring übertragen und das Timeout wieder gesetzt. Das kann nur wenige Male versucht werden, ansonsten muss der Membership-Algorithmus aktiv werden.

Durch das *token_seq*-Feld können redundante Tokens erkannt werden. Ein Prozessor akzeptiert ein Token nur, wenn dieses Feld größer oder gleich *my_token_seq* ist, andernfalls wird das Token als redundant verworfen. Wenn das Token akzeptiert wird, inkrementiert der Prozessor *token_seq* und setzt *my_token_seq* auch auf diesen neuen Wert.

1.3 Membership-Algorithmus

1.3.1 Allgemeines

Der Membership-Algorithmus erkennt den Verlust des Tokens, Prozessorausfälle und Netzwerkpartitionierung und konstruiert im Fehlerfall einen neuen Ring, auf dem der Totem-Ordering-Algorithmus weiterarbeiten kann. Ziel ist es, einen Konsens über den Mitgliederstatus des neuen Rings zu erreichen, ein neues Token zu generieren und Nachrichten wiederherzustellen, die von einigen Prozessoren noch nicht ausgeliefert worden waren, als das Fehlverhalten auftrat.

Es gibt vier Zustände des Algorithmus:

- **Operational State**: Nachrichten werden wie beim Total Ordering-Algorithmus beschrieben übertragen und ausgeliefert
- **Gather State**: Die Prozessoren tauschen „Join“-Nachrichten untereinander aus, um eine Einigung über die Ringmitgliedschaft zu erzielen. Jede solche Nachricht enthält eine Liste von Prozessoren, die vom Sender als Mitglieder des neuen Rings angesehen werden, und eine Liste von Prozessoren, die für fehlerhaft gehalten werden.
- **Commit State**: Durch Zirkulation des Commit-Tokens wird bestätigt, dass alle Prozessoren der Mitgliedschaft zustimmen und es werden Informationen gesammelt, um Nachrichten, die auf dem alten Ring nicht von allen Prozessoren ausgeliefert wurden, zu behandeln
- **Recover State**: Nachrichten vom alten Ring/von den alten Ringen werden neu übertragen

Es gibt sieben Ereignisse:

- Empfang einer fremden Nachricht (von einem Prozessor, der nicht Mitglied des Rings ist); dies aktiviert den Membership-Algorithmus in dem empfangenden Prozessor

- Empfang einer Join-Nachricht
- Empfang des Commit-Tokens (s.u.)
- Token Loss Timeout (s.o.); aktiviert den Membership-Algorithmus
- Join Timeout; wenn dieses abläuft, wird eine Join-Nachricht erneut übertragen (im Gather- oder Commit State)
- Consensus Timeout: Ein Prozessor, der an der Bildung des neuen Rings beteiligt ist, konnte nicht rechtzeitig einen Konsens erzielen (s.u.)
- Erkennung von Empfangsfehlern: Wenn sich nach mehreren Token-Umläufen *aru* nicht verändert hat, hat der Prozessor, der *aru* gesetzt hat, wiederholt Nachrichten nicht empfangen und wird als fehlerhaft eingestuft

Die Prozessoren bewahren lokal diverse Werte auf, u.a.

- *my_aru_count*: Zählt, wie oft der Prozessor das Token mit unverändertem *aru* empfangen hat (dazu wird *my_last_aru* gespeichert; Zweck: Erkennung von Empfangsfehlern)
- *my_proc_set*: Menge von Prozessoren, die für die Mitgliedschaft im neuen Ring in Frage kommen
- *my_fail_set*: Menge von Prozessoren, die während des Membership-Algorithmus Fehler gezeigt haben (Untermenge von *my_proc_set*)
- *consensus*: Boolean-Array, das für jeden Prozessor angibt, ob dessen *proc_set* und *fail_set* mit *my_proc*_set bzw. *my_fail_set* übereinstimmt oder nicht
- *my_trans_memb*: Menge von Prozessoren, die vom alten Ring zu einem neuen mit übergehen (s.u.)

1.3.2 Join-Nachrichten

Immer wenn ein Prozessor während des Gather State *my_proc_set* oder *my_fail_set* ändert, broadcastet er diese Nachricht. Join-Nachrichten können auch ohne Besitz des Tokens gesendet werden und sie werden nicht an höhere Schichten ausgeliefert. Sie bestehen aus folgenden Feldern:

- *type*: „Join“
- *sender_id*: Prozessor-ID des Senders
- *ring_seq*: Die größte Sequenznummer eines Rings, die dem Sender bekannt ist
- *proc_set*: bekommt den Inhalt von *my_proc_set*
- *fail_set*: bekommt den Inhalt von *my_fail_set*
- *rotation_count*: Anzahl, wie oft der Sender das Token weitergegeben hat, seit ein Konsens erreicht ist

Durch Senden einer Join-Nachricht versucht ein Prozessor einen Konsens über *proc_set* und *fail_set* zu erzielen.
Wenn ein Prozessor startet oder wieder anläuft, bildet er zuerst einen Ring, der nur ihn selbst enthält, und broadcastet dann eine Join-Nachricht.

1.3.3 Configuration Change-Nachrichten

Sie können den Wechsel von einer alten zu einer Übergangskonfiguration oder von einer Übergangs- zu einer neuen Konfiguration beschreiben und besteht aus folgenden Feldern:

- *ring_id*: ID der regulären Konfiguration, falls die Nachricht eine solche initiiert bzw. ID der vorherigen regulären Konfiguration, falls die Nachricht eine Übergangskonfiguration initiiert
- *seq*: 0, falls die Nachricht eine reguläre Konfiguration initiiert bzw. die größte Nachrichten-Sequenznummer der vorherigen regulären Konfiguration, falls die Nachricht eine Übergangskonfiguration initiiert

- *conf_id*: ID der Übergangskonfiguration, falls die Nachricht eine reguläre Konfiguration initiiert bzw. ID der Übergangskonfiguration, falls die Nachricht eine solche initiiert
- *memb*: Die Mitgliedschaft der Configuration, die die Nachricht initiiert

Configuration Change-Nachrichten werden lokal in jedem Prozessor generiert und nicht gebroadcastet, sondern nur an die Anwendung ausgeliefert, um dieser Mitgliedschaftsänderungen mitzuteilen.

1.3.4 Commit-Token

Das Commit-Token wird vom Representative eines Rings generiert, wenn dieser einen neuen Ring initiiert. Sein *type*-Feld steht auf „Commit“ und statt *rtr* enthält es diverse Informationen zu jedem Prozessor des neuen Rings, die für die Recovery-Phase benutzt werden. Diese Informationen tragen die Prozessoren beim ersten Umlauf des Commit-Tokens auf dem neuen Ring ein.

1.3.5 Operational State

Während dieses Zustands wird der Total Ordering-Algorithmus ausgeführt. Tritt ein „Membership Event“ (Verlust des Tokens, Join-Nachricht, fremde Nachricht) auf, dann wird der Algorithmus zur Bildung eines neuen Rings aufgerufen.

Wenn ein Token Loss-Timeout abläuft, broadcastet ein Prozessor eine Join-Nachricht, setzt Join- und Konsens-Timeouts und wechselt in den Gather State.

Wenn ein Prozessor eine fremde Nachricht erhält (die nicht im Recover State neu übertragen wurde), ergänzt er *my_proc_set* um die ID des Senders dieser Nachricht und wechselt in den Gather State.

Beim Empfang einer Join-Nachricht wird diese ignoriert, wenn der Empfänger in *fail_set* der Join-Nachricht enthalten ist, oder wenn der Sender in *my_proc_set* des Empfängers enthalten ist und *ring_seq* der Join-Nachricht kleiner als die Ring-Sequenznummer des Empfängers ist. Andernfalls wechselt der Empfänger in den Gather State.
Der Gather State wird auch betreten, wenn ein Prozessor Empfangsfehler aufweist. Dies wird erkannt, wenn *my_aru_count* eines Prozessors einen bestimmten (festzulegenden) Wert erreicht – der fehlerhafte Prozessor ist dann genau der, dessen ID in *aru_id* des Tokens steht.

1.3.6 Gather State

Ziel des Gather State ist es, eine möglichst große Mitgliedergruppe zu erreichen und dabei sicherzustellen, dass der Algorithmus terminiert. So eine Mitgliedergruppe ist eine Menge von Prozessor-IDs, auf die sich alle Prozessoren geeinigt haben und in der jeder Prozessor mit jedem anderen kommunizieren kann. Dazu werden Informationen über funktionierende und fehlerhafte Prozessoren gesammelt und mit Join-Nachrichten übermittelt.

Beim Empfang einer Join-Nachricht aktualisiert ein Prozessor *my_proc_set* und *my_fail_set*. Wenn diese sich ändern, verwirft er sein *consensus*, broadcastet eine Join-Nachricht mit den neuen Listen, setzt Join- und Konsens-Timeouts neu und verbleibt im Gather State.

Wenn die Listen der Join-Nachricht und die des Prozessors identisch sind, hält der Prozessor den Sender der Join-Nachricht als Befürworter des Konsenses über diese Listen fest (für die Sender-ID wird in consensus ein „true“ eingetragen).

Die Join-Nachricht wird ignoriert, wenn die ID des Senders in *my_fail_set* des Prozessors vorkommt. Dies muss so sein, weil ein Prozessor sich niemals selbst als fehlerhaft deklariert (und deshalb immer weiter Join-Nachrichten senden wird).

Wenn ein Prozessor eine Join-Nachricht empfängt, in der er selbst in *fail_set* enthalten ist, dann trägt er die ID des Senders in *my_fail_set* ein. Das ist angemessen, weil diese zwei Prozessoren keinen Konsens über eine Mitgliedschaft erreichen können, die (nur) einen der beiden ausschließt.

Wenn ein Prozessor eine Join-Nachricht empfängt, in der er nicht in *fail_set* enthalten ist, deren Sender nicht in *my_fail_set* enthalten ist und in deren *proc_set* bzw. *fail_set* IDs vorkommen, die der Empfänger noch nicht kennt, dann trägt er diese IDs in *my_proc_set* bzw. *my_fail_set* ein.

Immer wenn ein Prozessor eine Join-Nachricht broadcastet, setzt er ein Join-Timeout. Wenn dieses abläuft, übertragt er die Join-Nachricht erneut (um die Wahrscheinlichkeit zu erhöhen, dass Join-Nachrichten von allen aktiven Prozessoren in einer Konsens-Runde empfangen werden).

Ein Prozessor hat einen *Konsens erreicht*, wenn er Join-Nachrichten, in denen *proc_set* und *fail_set* seinen *my_proc_set* bzw. *my_fail_set* gleichen, von allen Prozessoren aus der Differenz dieser Mengen (also *my_proc_set* \ *my_fail_set*) empfangen hat (d.h. von allen fehlerfreien Prozessoren).

Ein Prozessor hat auch einen Konsens erreicht, wenn er ein Commit-Token empfängt mit derselben Mitgliedschaft wie *my_proc_set* \ *my_fail_set* ergibt. Die Prozessoren dieser Menge bilden den neuen Ring.

Nachdem ein Prozessor Konsens erreicht hat, wartet er auf das Commit-Token, welches der Representative generiert, nachdem er selbst Konsens erreicht hat. In das Commit-Token trägt er eine neue (höhere, basierend auf dem höchsten *ring_seq* aller Join-Nachrichten) Ring-Sequenznummer und alle Mitglieder des neuen Rings ein. Dadurch ist auch die Reihenfolge, in der das Token zirkulieren wird, vorgegeben. Nachdem er das Commit-Token abgeschickt hat, wechselt er in den Commit State.

Wenn die anderen Prozessoren das Commit-Token erhalten, führen sie einige Prüfungen durch, z.B. Vergleich der Mitgliederliste mit *my_proc_set* \ *my_fail_set*. Wenn sich Inkonsistenzen ergeben, wird das Commit-Token verworfen. Ansonsten trägt jeder Prozessor im Token Informationen zum Auslieferzustand von Nachrichten seines alten Rings ein und wechselt in den Commit State.

Wenn ein Konsens-Timeout abläuft, bevor ein Prozessor einen Konsens erreicht hat, fügt er alle Prozessoren aus *my_proc_set*, von denen er keine Join-Nachrichten mit seinen Listen entsprechenden *proc_set*/*fail_set* erhalten hat, *my_fail_set* hinzu. (Bedeutung: …) Dann beginnt der Gather State von vorn.

Beim Betreten des Gather State werden Token Loss-Timeouts abgebrochen. Wenn ein Prozessor Konsens erreicht hat, setzt er wieder ein Token Loss-Timeout und wartet auf das Commit-Token. Wenn dieses Timeout abläuft, wird versucht, einen neuen Konsens zu erreichen (Join-Nachrichten). Ist dieser neue Konsens identisch mit dem alten, wird derjenige Prozessor, der das Commit-Token am seltensten weitergereicht hat (ersichtlich aus *rotation_count* in den Join-Nachrichten) zu *my_fail_set* hinzugefügt und auf dieser Basis ein neuer Konsens gesucht.

1.3.7 Commit State

Im Commit State findet der zweite Umlauf des Commit-Tokens auf dem neuen Ring statt, um Daten für den Recovery-Algorithmus zu sammeln.

Aus dem Commit-Token gewinnt jeder Prozessor Informationen, welche anderen Prozessoren aus seinem jeweiligen alten Ring mit in den neuen Ring übergehen (alte Ring-ID und alte *aru*

sind dort für jeden Prozessor vermerkt); diese werden in *my_trans_memb* gespeichert. Dann wechselt der Prozessor in den Recover State.

Wird eine Join-Nachricht mit einer Ring-Sequenznummer größer als die des vorgeschlagenen neuen Rings empfangen, wechselt der Empfänger erneut in den Gather State. Das ist dann der Fall, wenn das Commit-Token oder das reguläre Token eines neuen Rings verloren gegangen ist.

Ein Prozessor verwirft ein Token mit einer Ring-Sequenznummer kleiner der des neuen Rings, denn so ein Token muss zu einem alten oder wieder verworfenen Ring gehören.

1.3.8 Recover State

Wenn der Representative des neuen Rings das Commit-Token nach seinem zweiten Umlauf wieder empfängt, wandelt er es in ein reguläres Token um. Dann ist der neue Ring gebildet, aber noch nicht installiert; vorher läuft der Recovery-Algorithmus ab.

1.4 Recovery-Algorithmus

Ziel des Recovery-Algorithmus ist es, Nachrichten wiederherzustellen, die bei Aktivierung des Membership-Algorithmus von einigen Prozessoren noch nicht ausgeliefert worden waren.
Der Algorithmus durchläuft folgende Schritte:

(1) Austausch von Nachrichten mit den Mitgliedern des alten Rings, so dass allen die gleiche Menge von Nachrichten vorliegt, diese aber noch nicht an die Anwendung ausgeliefert sind
(2) Ausliefern aller Nachrichten (des alten Rings), die in „agreed“ oder „safe order“ ausgeliefert werden können
(3) Ausliefern einer ersten Configuration Change-Nachricht, die in eine Übergangskonfiguration wechselt. Diese Übergangskonfiguration besteht aus den Prozessoren aus *my_trans_memb.*
(4) Ausliefern aller Nachrichten, die auf dem alten Ring nicht in „agreed“ oder „safe order“ ausgeliefert werden konnten, die aber in der kleineren Übergangskonfiguration ausgeliefert werden können
(5) Ausliefern einer zweiten Configuration Change-Nachricht, die in die neue Konfiguration wechselt
(6) Wechsel in den Operational State

2 SecureRing

2.1 Unterschiede zu Totem, Systemmodell

Da Totem als „Inspiration” für SecureRing diente, sind sich beide Systeme vom Prinzip her sehr ähnlich.

Hauptunterschied ist, dass SecureRing im Gegensatz zu Totem mit *byzantinischen Fehlern* (bösartigem Verhalten von Prozessoren) umgehen kann. Dazu arbeitet in jedem Prozessor ein unzuverlässiger Detektor für byzantinische Fehler. Prozessoren, die solche Fehler zeigen, werden ausgeschlossen. Es hat sich gezeigt, dass in Umgebungen mit insgesamt n Prozessoren mindestens $(2n + 1)/3$ (aufgerundet) Prozessoren korrekt sein müssen, d.h. höchstens $(n – 1)/3$ (abgerundet) byzantinische Prozessoren dabei sein dürfen. Einfach ausgefallene Prozessoren werden auch wie byzantinische Prozessoren behandelt.

Ein Public Key-Verschlüsselungsverfahren (z.B. RSA) ist enthalten: Jeder Knoten hat einen privaten Schlüssel, um Nachrichten digital zu signieren, und kann die öffentlichen Schlüssel der anderen Knoten bekommen, um signierte Nachrichten zu verifizieren. Signiert werden das Token, das Commit-Token, die Join-Nachricht und die Notify-Nachricht, aus Performance-Gründen jedoch nicht normale Nachrichten.

Das Token wird in SecureRing nicht weitergereicht, sondern wie auch Nachrichten gemulticastet, ein Prozessor ist in Besitz des Tokens, wenn der Sender des Tokens sein Vorgänger ist. Im Token selbst werden eine Übersicht der Nachrichten gespeichert, die der Token-Besitzer erzeugt hat, sowie das vorherige Token.

2.2 Unzuverlässiger Detektor für byzantinische Fehler

2.2.1 Allgemeines

Das Membership-Protokoll benötigt Fehlererkennung mit folgenden Eigenschaften:

- *Eventual Strong Byzantine Completeness*: Es gibt eine Zeit, nach der jeder byzantinische Prozessor dauerhaft von jedem korrekten Prozessor als solcher erkannt ist.
- *Eventual Strong Accuracy*: Es gibt eine Zeit, nach der jeder korrekte Prozessor nicht mehr von einem anderen korrekten Prozessor verdächtigt wird. In der Praxis ist es ausreichend, wenn dieser Zustand solange anhält, bis eine neue Konfiguration installiert ist.

In einem asynchronen System ist es theoretisch unmöglich, einen Fehlerdetektor mit diesen Eigenschaften zu realisieren. In der Praxis gibt es aber Systeme, die „meistens“ vernünftig funktionieren.

Der Fehlerdetektor überwacht gesendete Nachrichten und liefert als Ausgabe eine Liste von Prozessoren, die eines byzantinischen Verhaltens verdächtigt werden. Diese Liste teilt er dem Membership-Protokoll mit. Es gibt folgende Arten byzantinischer Fehler:

- Senden von mutierten, signierten Nachrichten
- Senden von signierten Nachrichten, die nicht richtig aufgebaut sind
- dauerhaftes Unterlassen der Bestätigung, dass Nachrichten oder ein Token empfangen wurden
- dauerhaftes Unterlassen des Sendes von Nachrichten, die vom Message Delivery- oder vom Membership-Protokoll benötigt werden

2.2.2 Behandlung von Mutierten Nachrichten

Hier gibt es z.B. die Fälle, dass das im Token gespeicherte vorherige Token nicht zum tatsächlichen vorherigen Token passt, oder dass ein Prozessor mehrere Join-Nachrichten mit derselben ID aber unterschiedlichem Inhalt sendet. Diese und einige ähnliche Fälle werden erkannt und die verantwortlichen Prozessoren als fehlerhaft eingestuft (das bedeutet dessen Aufnahme in *my_fault_set*, ähnlich wie bei Totem).

2.2.3 Behandlung von falsch aufgebauten Nachrichten

Der korrekte Aufbau der verschiedenen Nachrichtentypen (ihre Felder) ist festgelegt, kann also leicht überprüft werden. Der Sender einer falsch aufgebauten Nachricht wird als fehlerhaft eingestuft.

Weiterhin werden verschiedene Inhalte geprüft. Tokens (t_i) werden z.B. mit dem vorherigen Token (t_{i-1}) verglichen – die Sequenznummer von t_i muss sich von der in t_{i-1} genau in der Anzahl der vom letzten Token-Besitzer gesendeten Nachrichten unterscheiden (die ja aus dem

Token ersichtlich ist, s.o.). Das *aru*-Feld und *rtr_list* müssen gegenseitig konsistent sein. Das *aru*-Feld darf zwischen zwei aufeinander folgenden Tokens nicht kleiner werden.

Ein byzantinischer Prozessor könnte das *seq*-Feld im Token erhöhen und entsprechende Nachrichteneinträge vornehmen, ohne tatsächlich Nachrichten zu senden. Korrekte Prozessoren würden dann wegen Nicht-Empfang der (nicht existenten) Nachrichten als fehlerhaft angesehen werden. Dagegen wirkt der Mechanismus, dass wenn mindestens $(2n + 1)/3$ Prozessoren denselben *aru*-Wert gesetzt oder dieselbe ID in *rtr_list* eingefügt haben und diese Werte eine bestimmte Zahl von Token-Umläufen erhalten geblieben sind, die entsprechenden Nachrichten nicht-existent sein müssen. Der verantwortliche Prozessor wird als fehlerhaft eingestuft

2.2.4 Behandlung von fehlenden Bestätigungen von Nachrichten

Ein Prozessor, der eine festzulegende Anzahl Tokens sendet, in denen das *aru*-Feld unverändert ist, oder in denen dieselbe ID in *rtr_list* vorkommt, wird als fehlerhaft eingestuft

2.2.5 Behandlung von unterlassenem Senden

Es werden auch Timeouts zur Fehlererkennung benutzt. Immer wenn ein Prozessor ein Token empfängt, setzt er ein Token-Loss-Timeout, welches abläuft, wenn er nicht rechtzeitig ein Token vom nächsten Token-Besitzer empfängt. Dieser nächste Token-Besitzer wird dann als fehlerhaft eingestuft. Ein Token-Loss-Timeout wird z.B. auch beim Eintritt in die Recover-Phase gesetzt (der Representative muss ein Token senden).

Ein byzantinischer Prozessor, der gerade Token-Besitzer ist, könnte das Token selektiv multicasten und es dabei gerade nicht seinem Nachfolger senden. In diesem Fall würden andere Prozessoren, bei denen ein Token-Loss-Timeout abläuft, diesen Nachfolger anstatt des byzantinischen Prozessors verdächtigen. Gegen diesen Angriff gibt es die Neuübertragung von Tokens: Jeder Prozessor setzt beim Empfang oder beim Senden eines Tokens ein Token-Retransmission-Timeout, bei dessen Ablauf er das zuletzt empfangene bzw. gesendete Token erneut überträgt – das Retransmission-Timeout ist kürzer als das Token-Loss-Timeout.

Es gibt auch ein Konsens-Timeout, das gesetzt wird, wenn ein Prozessor mit mindestens $(2n + 1)/3$ Prozessoren einen Konsens gefunden hat, aber noch nicht mit allen Prozessoren der Menge *my_proc_set* \ *my_fault_set*. Wenn das Timeout abläuft, werden diejenigen Prozessoren als fehlerhaft eingestuft, mit denen kein Konsens erreicht werden konnte.

Literaturverzeichnis

Agar94 Deborah A. Agarwal, Totem: A Reliable Ordered Delivery Protocol for Interconnected Local-Area Networks, Department of Electrical and Computer Engineering, University of California, Santa Barbara, 1994

Kihl98 K. P. Kihlstrom, L. E. Moser, P. M. Melliar-Smith, The SecureRing Protocols for Securing Group Communication, Department of Electrical and Computer Engineering, University of California, Santa Barbara, 1998