Überblick

- Verteilte Algorithmen für fehlertolerante Programme
  - Grundbegriffe
    - Prozess, Verbindung, Fehler, Korrektheit
  - Grundlagen für die Beschreibung verteilter Algorithmen
    - Modularisierung
    - Raum-Zeit Diagramm
  - Modellierung von Systemeigenschaften
    - Eigenschaften von Punkt-zu-Punkt Verbindungen
    - Synchrone, partiell synchrone und asynchrone Systeme
  - Basisabstraktionen
    - Ausfallerkennung
    - Wahl eines Anführers
  - Multicast
    - Zuverlässigkeit
    - Ordnung von Nachrichten

Grundbegriffe

- *Prozess* oder auch *Knoten, Rechner, Prozessor*
  - Meist synonym gebraucht; bezeichnen eine einzelne aktive Instanz im verteilten System, die mit anderen Instanzen durch Nachrichtenaustausch über eine *Verbindung* interagiert
  - Prozesse sind eindeutig identifizierbar
    - In der Regel ist eine statische Menge von Prozessen gegeben
    - Unter Umständen auch (monoton) aufsteigend nummeriert
- *Globaler Zustand*
  - Verteilter Zustand des Systems zu einem Zeitpunkt der realen Zeit, ausgedrückt durch einen Vektor $S = [S_1, ..., S_n]$ aus den Zuständen S_i aller n im System vorhandenen Knoten
- *Schritt*
  - Atomarer Übergang von einem globalen Zustand in einen Folgezustand

Grundbegriffe *(Gutmütige Fehler, Wdh.)*

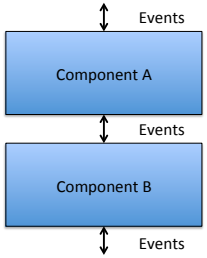
- *Crash Stop:* Ein Knoten fällt komplett aus
  - *Fail Stop*: Jeder korrekte Knoten erfährt innerhalb endlicher Zeit vom Ausfall eines Knotens
  - *Fail Silent*: Keine perfekte Ausfallerkennung möglich (asynchrones oder partiell synchrones Modell)
- *Crash Recovery*: Ein korrekter Knoten kann endlich oft ausfallen und wieder anlaufen. Ein Knoten wird nur dann als fehlerhaft betrachtet, wenn er dauerhaft ausfällt oder unendlich oft ausfällt und wieder anläuft.
  - Schwierigkeit: Stabiler Speicher, um Zustand von korrekten Knoten über Ausfälle hinweg zu erhalten
  - Zwei Spezialfälle
    - *Omission Failure:* Bei Ausfall eines korrekten Knotens gehen nur Nachrichten verloren, kompletter Zustand bleibt erhalten
    - *Timing Failure:* Es gehen weder Nachrichten noch Zustand verloren, bei einem Ausfall kann aber die spezifikationsgerechte Ausführung verzögert werden

Grundbegriffe

- Korrektheit
  - *Sicherheit* („Integrität", „safety")
    - Der Algorithmus liefert keine falschen Ergebnisse
  - *Lebendigkeit* („liveness")
    - Der Algorithmus verklemmt sich nicht
  - Das Vorliegen von Sicherheit wird auch als *partielle Korrektheit* bezeichnet
  - Von *totaler Korrektheit* spricht man, wenn sowohl Sicherheit als auch Lebendigkeit erfüllt ist

Beschreibung verteilter Algorithmen

- Beschreibung von Algorithmen durch Pseudocode
  - Ermöglicht eine von Implementierungsdetails befreite Erfassung der Kernfunktionalität von Algorithmen
- Entwicklung und Strukturierung mittels *Komponenten*
  - Reduziert die Komplexität der aktuellen Betrachtungen
  - ! In der Realität muss jedoch oft auf eine entsprechende Gliederung zu Gunsten von Effizienz verzichtet werden



Komponenten und Nachrichten

- Komponenten werden zu einem *Stapel* innerhalb eines *Prozesses* kombiniert
- Austausch von Informationen zwischen benachbarten Komponenten, sowie Prozessen erfolgt durch Nachrichten (*Events*)
- Komponenten entsprechen einer Zustandsmaschine
- Nachrichten werden an Komponente (z.B. *co*) adressiert
  $\langle\ co,\ EventType\ |\ \text{Attributes}, \ldots\ \rangle$
- Abhängig von der Ausprägung einer Nachricht gibt es innerhalb einer Komponente spezifische Behandlungsroutinen
  upon event $\langle\ co,\ EventType\ |\ \text{Attributes}, \ldots\ \rangle$ **do**
- Verarbeitung einer Nachricht kann die Erzeugung weiterer auslösen
- Nachrichten innerhalb **eines** Prozesses werden in FIFO-Ordnung zwischen Komponenten übertragen

Bearbeitung von Nachrichten

- Es wird zu einem Zeitpunkt pro Prozess nur eine Nachricht bearbeitet und diese vollständig (atomarer Schritt)
- Es wird periodisch überprüft, ob es unbearbeitete Nachrichten gibt und fair die nächste Nachricht ausgewählt
- Beispiel:

upon event $\langle\ co_1,\ EventType_1\ |\ att_1^1, att_1^2, \ldots\ \rangle$ do
  do something;
  trigger $\langle\ co_2,\ EventType_2\ |\ att_2^1, att_2^2, \ldots\ \rangle$

upon event $\langle\ co_1,\ EventType_3\ |\ att_3^1, att_3^2, \ldots\ \rangle$ do
  do something else;
  trigger $\langle\ co_2,\ EventType_4\ |\ att_4^1, att_4^2, \ldots\ \rangle$

- (Interne) Bedingungen

upon *condition* **do**
  do something;

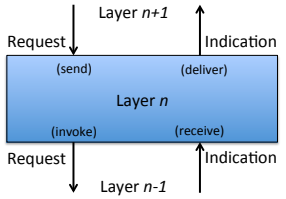
upon $\langle\ co_1,\ EventType_1,\ att_1^1, att_1^2, \ldots\ \rangle$ **such that** *condition* **do**
  do something;

Bearbeitung von Nachrichten

- Zur Initialisierung von Variablen gibt es die *Init* Nachricht, die automatisch bei der Instanziierung einer Komponente zugestellt wird

upon event $\langle\ co,\ Init\ \rangle$ **do**
  do something;

- Schnittstelle einer Komponente
  - *Request* Nachrichten
    - Nachrichten stellen Eingaben für tiefer liegende Komponenten da
  - *Indication* Nachrichten
    - Nachrichten sind Eingaben für höher liegende Komponenten



Module

- Module als Beschreibung einer abstrakten Komponentenschnittstelle
  - Ermöglicht die *Implementierung* eines Moduls in verschiedenen Ausprägungen
  - Beispiel:

Module:
Name: JobHandler, **instance** jh.

Events:
Request: ⟨ *jh*, *Submit* | job ⟩: Requests a job to be processed.

Indication: ⟨ *jh*, *Confirm* | job ⟩: Confirms that a given job has been *or* will be processed.

Properties:
JH1: *Guaranteed response*: Every job is eventually confirmed.

Module

- Beispiel für die Realisierung des JobHandler-Moduls:

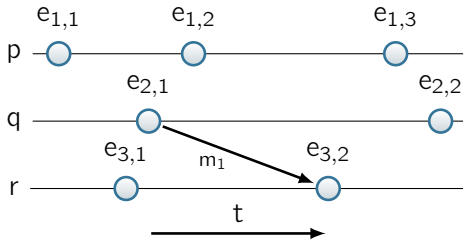
Implements:
JobHandler, **instance** jh.

upon event ⟨ *jh*, *Submit* | job ⟩ **do**
process(job);
trigger ⟨ *jh*, *Confirm* | job ⟩

- Wie würde eine asynchrone Variante aussehen?

Zeit-Raum Diagramm

- Graphische Darstellung von lokalen Ereignisse und Interaktionen aller Prozessoren



- Konventionen für Bezeichner
  - Prozesse: Buchstaben ($q, p, r, s \ldots$ oder $P_1, \ldots$)
  - Ereignisse: Kleinbuchstaben ($a, b, c, \ldots$ oder $e_1, e_2, e_3, \ldots$)
  - Nachrichten: Kleinbuchstaben ($m_1, m_2, \ldots$)
    Übertragungsrichtungen als (Sender, Empfänger) (z.B. $m_1^{2,3}$)

Punkt-zu-Punkt-Verbindungen

- Es wird davon ausgegangen, dass alle Teilnehmer eines Algorithmus eine bidirektionale Verbindung zueinander haben
- Es wird angenommen, dass Zusammenhänge wie Anfrage und resultierende Antwort bestimmt werden können (z.B. durch eindeutige Identifikatoren)
  ! Vereinfacht die Beschreibung von Algorithmen
- Punkt-zu-Punkt Verbindungen können verschiedene Eigenschaften (in Abhängigkeit des Fehlermodells) haben, welche explizit zu modellieren sind:
  - Ausfälle (Crash Stop)
    - unzuverlässige Verbindungen durch einen *fair-loss link*
    - zuverlässige Verbindungen durch einen *perfect link*
  - Ausfälle mit Wiederanlauf (Crash Recovery)
    - zuverlässige Verbindungen durch einen *logged perfect link*

Modellierung von unzuverlässigen Verbindungen

- Eigenschaften einer unzuverlässigen Kommunikationsverbindungen („*fair-loss link*“)
  - Keine „beliebige“ Unzuverlässigkeit (sonst keine sinnvolle Validierung von Algorithmen möglich), sondern realistische Annahmen

Module:
Name: FairLossPointToPointLinks, **instance** *fll*.

Events:
Request: ⟨ *fll*, *Send* | *q*,*m* ⟩: Requests to send message *m* to process *q*.

Indication: ⟨ *fll*, *Deliver* | *p*,*m* ⟩: Delivers message *m* sent by process *p*.

Properties:
FLL1: *Faire Verluste*: Eine von *p* unendlich oft gesendete Nachricht *m* kommt bei *q* auch unendlich oft an.

FLL2: *Endliche Verdopplung*: Wenn *m* endlich oft von *p* nach *q* gesendet wird, kommt *m* nicht unendlich oft bei *q* an.

FLL3: *Keine Erfindung*: Erhält *p* eine Nachricht *m*, so wurde *m* zuvor von einem Prozess *q* gesendet.

Modellierung von zuverlässigen Verbindungen (1)

- Verschatten von Nachrichtenverlusten durch *ständig* wiederholtes zustellen

Module:
Name: StubbornPointToPointLinks, **instance** *sl*.

Events:
Request: ⟨ *sl*, *Send* | *q*,*m* ⟩: Requests to send message *m* to process *q*.

Indication: ⟨ *sl*, *Deliver* | *p*,*m* ⟩: Delivers message *m* sent by process *p*.

Properties:
SL1: *'Ständige' Zustellung*: Eine von *p* gesendete Nachricht an *q* wird von *q* unendlich oft zugestellt.

SL2: *Keine Erfindung*: Erhält *p* eine Nachricht *m*, so wurde *m* zuvor von einem Prozess *q* gesendet.

Modellierung von zuverlässigen Verbindungen (2)

- Einfache Realisierung eines „*stubborn link*“

Implements:
StubbornPointToPointLinks, **instance** *sl*.

Uses:
FairLossPointToPointLinks, **instance** *fll*.

upon event ⟨ *sl*, *Init* ⟩ **do**
  sent:=∅;
  starttimer(Δ);

upon event ⟨ *Timeout* ⟩ **do**
  forall (*q*,*m*) ∈ *sent* **do**
    trigger ⟨ *fll*, *Send* | *q*,*m* ⟩;
  starttimer(Δ);

upon event ⟨ *sl*, *Send* | q,m ⟩
  trigger ⟨ *fll*, *Send* | q,m ⟩;
  sent := *sent* ∪ {(q,m)};

upon event ⟨ *fll*, *Deliver* | p,m ⟩ do
  trigger ⟨ *sl*, *Deliver* | p,m ⟩;

Modellierung von zuverlässigen Verbindungen (3)

- Eigenschaften einer zuverlässigen Kommunikationsverbindungen („*perfect link*“)

Module:
Name: PerfectPointToPointLinks, **instance** *pl*.

Events:
Request: ⟨ *pl*, *Send* | *q*,*m* ⟩: Requests to send message *m* to process *q*.

Indication: ⟨ *pl*, *Deliver* | *p*,*m* ⟩: Delivers message *m* sent by process *p*.

Properties:
PL1: *Zuverlässige Übermittlung*: *p* sendet Nachricht *m* an *q*. Falls weder *p* noch *q* ausfällt, erhält *q* die Nachricht *m* in endlicher Zeit.

PL2: *Keine Verdopplung*: Keine Nachricht wird einem Prozess mehr als einmal zugestellt.

PL3: *Keine Erfindung*: Erhält *q* eine Nachricht *m*, so wurde *m* zuvor von einem Prozess *p* gesendet.

Modellierung von zuverlässigen Verbindungen (4)

- Realisierung einer zuverlässigen Verbindung durch die Unterdrückung von Nachrichtenduplikaten

Implements:
PrefectPointToPointLinks, **instance** *pl*.

Uses:
StubbornPointToPointLinks, **instance** *sl*.

upon event ⟨ *pl*, *Init* ⟩ **do**
delivered:=∅;

upon event ⟨ *pl*, *Send* | q,m ⟩
trigger ⟨ *sl*, *Send* | q,m ⟩;

upon event ⟨ *sl*, *Deliver* | p,m ⟩ do
if $m \notin delivered$ **then**
$delivered := delivered \cup \{m\}$;
trigger ⟨ *pl*, *Deliver* | p,m ⟩;

Modellierung von zuverlässigen Verbindungen (5)

- In der Realität verlieren Verbindungen Nachrichten, duplizieren diese („*fair-loss link*“) oder es treten Veränderungen an den Nachrichten selbst auf (behandelt durch die Verwendung von Prüfsummen).
- Es kann jedoch durch die beschriebenen Abstraktionen davon ausgegangen werden, dass durch entsprechende Protokollschichten (z.B. TCP) im Rahmen von Algorithmen von zuverlässigen Verbindungen ausgegangen werden kann („*perfect link*“).
- Für die effizientere Beschreibung bzw. Implementierung von Algorithmen kann selektiv auf schwächere Abstraktionen zurückgegriffen werden.
  - Z.B. um in die Sequenzierung einzugreifen.
- Weiterhin ist es so, dass Rahmenbedingungen wie Topologie, Flusskontrolle und Leistungsunterschiede unter Umständen zusätzliche Maßnahmen erfordern.

Punkt-zu-Punkt Verbindung Nachrichtenreihenfolge

- **Ungeordnet**
Keinerlei Aussage über Empfangsreihenfolge von Nachrichten
- **FIFO-Ordnung** (Sequentielle Ordnung, Senderordnung)
Nachrichten, die von einem Knoten in einer bestimmten Reihenfolge ausgesendet werden, werden von anderen Knoten in genau dieser Reihenfolge empfangen
- Weder *fair-loss link* noch *perfect link* machen eine Aussage zur Reihenfolge, in der Nachrichten ankommen

„*Synchron*“ und „*Asynchron*“

- Oft sind hier unterschiedliche Dinge damit gemeint:
  - Entfernte Methodenaufrufe:
    - synchron: Aufrufer wartet (blockierend) auf Ergebnis
    - asynchron: Aufrufer wartet nicht (nicht-blockierend)
  - Koordinierung
    - „Synchronisierung“ als Synonym gebraucht; „nicht gleichzeitig“
  - Ausführung von verteilten Aktivitäten:
    - synchron: Synonym zu „gleichzeitig“: Aktivitäten werden mit Synchronisierung ausgeführt (durch gemeinsame Uhren oder andere Synchronisierungsmechanismen gesteuert)
    - asynchron: Keine Synchronisierung vorhanden

„*Synchron*“ und „*Asynchron*“

- Im Kontext von verteilten Algorithmen übliche Bedeutung

  asynchron Unabhängigkeit von physikalischer Zeit

  synchron Alle Komponenten des Systems halten zeitliche Schranken ein

- Systemeigenschaften eines synchronen Systems

  SYN1 *Synchrone Verarbeitung*: Für alle Verarbeitungsschritte gibt es eine obere Zeitschranke

  SYN2 *Synchrone Kommunikation*: Für alle Nachrichtenlaufzeiten existiert eine obere Schranke

„*Synchron*“ und „*Asynchron*“

- Eigenschaften und Vorteile synchroner Systeme
  - Ablauf in Runden: Im synchronen Modell ist es möglich, Algorithmen verteilt in Runden ablaufen zu lassen
  - Ausfallerkennung durch Zeitschranken ist möglich
  - Zeitbasierte Koordination (Leases) kann verwendet werden
  - Eine Synchronisation physikalischer Uhren kann mit bestimmbarer Genauigkeit durchgeführt werden
  - Performanz-Analyse ist möglich
- Nachteil eines synchronen Systemmodells
  - In vielen Situation (z.B. internetbasierte, verteilte Systeme) realitätsfern

Partielle Synchronität

- Problem
  - synchron ⇒ Einfache Algorithmen, aber realitätsfern
  - asynchron ⇒ Deckt Realität ab, aber komplexe bzw. *nicht* existente Algorithmen
- Versuch eines Auswegs: **Partielle Synchronität**
  - Meistens werden die Eigenschaften eines synchronen Systems erfüllt. Manchmal kann es aber Perioden geben, in denen sich das System beliebig (asynchron) verhält.
- Abstrakte Modellierung
  - Es gibt einen (unbekannten) Zeitpunkt, ab dem sich das System synchron verhält

Ausfallerkennung

- Ausfallerkennung durch ein Orakel
  - Beliebte Abstraktion für Algorithmen im verteilten System
    - Orakel liefert Aussagen über den Zustand von entfernten Rechnern
    - Orakel erfüllt formal (einfach) fassbare Eigenschaften
- Ziele/Vorteile
  - Vereinfachung von Algorithmen durch Modularisierung
  - Algorithmen können ohne die (vom Systemmodell abhängigen) Realisierung der Ausfallerkennung implementiert und verifiziert werden

Ausfallerkennung

- Definition des „Perfect Failure Detetor“ ($\mathcal{P}$)

Module:
Name: PerfectFailureDetector , **instance** $\mathcal{P}$.

Events:
Indication: $\langle\ \mathcal{P},\ Crash\ |\ p\ \rangle$: Detects that process p has crashed.

Properties:
PFD1: *Letztendliche Vollständigkeit*: Es gibt einen (unbekannten) Zeitpunkt, ab dem jeder fehlerhafte Prozess von jedem korrekten Prozess dauerhaft als ausgefallen erkannt wird.

PFD2: *Starke Genauigkeit*: Kein Prozess wird bevor er ausgefallen ist von einem anderen als ausgefallen erkannt.

- Realisierung
  - Im synchronen Modell und unter der Annahme von Ausfällen: z.B. durch *Alive*-Nachrichten und Timeouts (siehe nächste Folie)
  - Im asynchronen Modell: ???

Ausfallerkennung

- Realisierung eines „Perfect Failure Detetor“ ($\mathcal{P}$) unter Verwendung von Timeouts

Implements:
PerfectFailureDetector , **instance** $\mathcal{P}$.

Uses:
PrefectPointToPointLinks, **instance** *pl*.

upon event $\langle\ \mathcal{P},\ Init\ \rangle$ **do**
  $alive := \Pi$; $detected := \emptyset$; *starttimer*(Δ);

upon event $\langle\ Timeout\ \rangle$ **do**
  forall $p \in \Pi$ **do**
    if$(p \notin alive) \wedge (p \notin detected)$ **then**
      $detected := detected \cup \{p\}$;
      trigger $\langle\ \mathcal{P},\ Crash\ |\ p\ \rangle$
    trigger $\langle\ pl,\ Send\ |\ p,\ [\text{HeartbeatRequest}]\ \rangle$;
  $alive := \emptyset$; *starttimer*(Δ);

upon event $\langle\ pl,\ Deliver\ |\ p,\ [\text{HeartbeatRequest}]\ \rangle$ **do**
  trigger $\langle\ pl,\ Send\ |\ p,\ [\text{HeartbeatReply}]\ \rangle$;

upon event $\langle\ pl,\ Deliver\ |\ p,\ [\text{HeartbeatReply}]\ \rangle$ **do**
  $alive := alive \cup \{\ p\}$;

Ausfallerkennung

- Korrektheit
  - Letztendliche Vollständigkeit
    - Bei Ausfall eines Prozess p werden durch ihn keine Nachrichten mehr versendet
    - Durch die Eigenschaften des Kommunikationssystems wird gewährleistet, dass keine Nachrichten zugestellt werden, die nicht auch versendet wurden

    $\Rightarrow$ Alle Prozesse erkennen den Ausfall von p
  - Starke Genauigkeit
    - Der Ausfall eines Prozesses p wird nur dann erkannt, wenn ein Prozess q keine Antwort auf seine Nachricht an p vor dem Timeout erhält.

    $\Rightarrow$ Dies kann nur dann eintreten, wenn Prozess p tatsächlich ausgefallen ist, da eine rechtzeitige Zustellung das synchrone System erzwingt.
- Performanz
  - Der vorgestellte Algorithmus kann optimiert werden, indem keine Nachrichten an ausgefallene Prozesse zugestellt werden
  - Anpassung des Timeouts in Abhängigkeit der Anforderungen

Letztendlich perfekte Ausfallerkennung

- Einfacher in der Realität zu realisierende Variante
  - Definition des „Eventually Perfect Failure Detector“ ($\Diamond\mathcal{P}$):

Module:
Name: EventuallyPerfectFailureDetector, **instance** $\Diamond\mathcal{P}$.

Events:
Indication: $\langle\ \Diamond\mathcal{P},\ Suspect\ |\ \text{p}\ \rangle$: Notifies that process p is suspected to have crashed.

Indication: $\langle\ \Diamond\mathcal{P},\ Restore\ |\ \text{p}\ \rangle$: Notifies that process p is not suspected anymore.

Properties:
EPFD1: *Letztendliche Vollständigkeit:* Es gibt einen (unbekannten) Zeitpunkt, ab dem jeder fehlerhafte Prozess von jedem korrekten Prozess dauerhaft als ausgefallen verdächtigt wird

EPFD2: *Letztendliche Genauigkeit:* Es gibt einen Zeitpunkt, ab dem kein korrekter Prozess von einem anderen korrekten Prozess als ausgefallen betrachtet wird

Letztendliche perfekte Ausfallerkennung

- Realisierung einer *letztendliche* perfekte Ausfallerkennung durch Verwendung eines adaptiven Timeouts
  - Da die Latenz des Netzwerkes unbekannt ist aber dennoch eine möglichst rasche Erkennung von Fehlern erfolgen soll wird das Timeout initial gering angesetzt und dann bei Erkennungsfehlern sukzessive erhöht.

Implements:
EventuallyPerfectFailureDetector , **instance** $\diamond\mathcal{P}$.
Uses:
PrefectPointToPointLinks, **instance** *pl*.

upon event $\langle\ \mathcal{P}$, *Init* $\rangle$ **do**
alive $:= \Pi$; *suspected* $:= \emptyset$; *delay* $:= \Delta$; *starttimer(delay)*;

upon event $\langle$ *pl*, *Deliver* | *q*, [HeartbeatRequest] $\rangle$ **do**
trigger $\langle$ *pl*, *Send* | *q*, [HeartbeatReply] $\rangle$;

upon event $\langle$ *pl*, *Deliver* | *p*, [HeartbeatReply] $\rangle$ **do**
alive $:=$ *alive* $\cup$ { *p*};

Letztendliche perfekte Ausfallerkennung (Forts.)

upon event $\langle$ *Timeout* $\rangle$ **do**
  if *alive* $\cap$ *suspected* $\neq \emptyset$ **then**
    delay $:=$ *delay* $+ \Delta$;
  forall $p \in \Pi$ **do**
    if($p \notin$ *alive*) $\wedge$ ($p \notin$ *suspected*) **then**
      suspected $:=$ *suspected* $\cup \{p\}$;
      trigger $\langle\ \diamond\mathcal{P}$, *suspected* | *p* $\rangle$
    else if ($p \in$ *alive*) $\wedge$ ($p \in$ *suspected*) **then**
      suspected $:=$ *suspected* $\setminus \{p\}$;
      trigger $\langle\ \diamond\mathcal{P}$, *Restore* | *p* $\rangle$;
    trigger $\langle$ *pl*, *Send* | *p*, [HeartbeatRequest] $\rangle$;
  alive $:= \emptyset$;
  starttimer(delay);

Wahl eines Anführers

- Problemstellung
  - In verteilten Systemen kann es viele Situationen geben, die einen ausgewählten Knoten erfordern („Anführer")
    - Koordinator von verteilten Aktionen
    - Erzeugung von eindeutigen „Token" im System
    - ...
  - Man möchte diesen Knoten automatisch bestimmen lassen
    - Initial beim Start des Systems
    - Zur Neukonfiguration des Systems nach Fehlern
- Realisierung
  - Kann unter Verwendung von Ausfallerkennung etabliert werden

Wahl eines Anführers

- Definition eines Moduls zur Anführerwahl (unter der Annahme einer perfekten Ausfallerkennung)

Module:
Name: LeaderElection , **instance** *le*.

Events:
Indication: $\langle$ *le*, *Leader* | *p* $\rangle$: Indicates that process *p* is elected as leader.

Properties:
LE1: *Letztendliche Festlegung*: Entweder es gibt keinen korrekten Prozess oder es wir in endlicher Zeit ein Anführer gewählt.
LE2: *Genauigkeit*: Ist ein Prozess Anführer so sind alle bisher gewählten Anführer ausgefallen.

- Korrektheit
  - Durch LE2 wird implizit erreicht, dass sich zu keinem Zeitpunkt zwei Knoten als Anführer (p,q) betrachten
  - Ein Prozess bleibt so lange Anführer bis er ausfällt, erst dann wird ein neuer Anführer bestimmt

Wahl eines Anführers

- Realisierung einer *stammbaumbasierten* Anführerwahl

Implements:
LeaderElection, **instance** *le*.

Uses:
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon $\langle$ *le*, *Init* $\rangle$ **do**
detected $:= \emptyset$;
leader $:= \perp$;

upon $\langle$ $\mathcal{P}$, *Crash* | *p* $\rangle$ **do**
detected $:=$ *detected* $\cup \{p\}$;

upon *leader* $\neq$ *maxrank(* $\Pi \setminus$ *detected)* **do**
leader $:=$ *maxrank(* $\Pi \setminus$ *detected)*;
trigger $\langle$ *le*, *Leader* | leader $\rangle$;

- (Zusatz)annahmen
  - Jeder Prozess hat einen eindeutigen Identifikator (z.B. $i \in \mathbb{N}$)
  - *maxrank* ermittelt aus einer Menge von Prozessen, den Prozess mit dem größten Identifikator

Wahl eines Anführers

- Definition eines Moduls zur Anführerwahl unter der Annahme einer letztendlich perfekten Ausfallerkennung

Module:
Name: EventualLeaderDetector, **instance** Ω.

Events:
Indication: $\langle$ Ω, *Trust* | *p* $\rangle$: Indicates that process *p* is *trusted* to be leader.

Properties:
ELD1: *Eventual accuracy:* There is a time after which every correct process trusts some correct process

ELD2: *Eventual agreement*: There is a time after which no two correct processes trust different correct processes.

Wahl eines Anführers

- Stammbaumbasierte Anführerwahl unter Verwendung einer letztendlich perfekten Ausfallerkennung

Implements:
EventualLeaderDetector, **instance** Ω.

Uses:
EventuallyPerfectFailureDetector, **instance** $\Diamond\mathcal{P}$.

upon $\langle$ Ω, *Init* $\rangle$ **do**
suspected $:= \emptyset$;
leader $:= \perp$;

upon $\langle$ $\Diamond\mathcal{P}$, *Suspect* | *p* $\rangle$ **do**
suspected $:=$ *suspected* $\cup \{p\}$;

upon $\langle$ $\Diamond\mathcal{P}$, *Restore* | *p* $\rangle$ **do**
suspected $:=$ *suspected* $\setminus \{p\}$;

upon *leader* $\neq$ *maxrank(* $\Pi \setminus$ *suspected)* **do**
leader $:=$ *maxrank(* $\Pi \setminus$ *suspected)*;
trigger $\langle$ Ω, *Trust* | leader $\rangle$;

Wahl eines Anführers

- Korrektheit
  - Die letztendliche Exaktheit des Algorithmus begründet sich in der letztendlichen Vollständigkeit des Fehlerdetektors ($\Diamond\mathcal{P}$), da nicht auf als ausgefallen vermutete Prozesse vertraut wird und der Fehlerdetektor ab einem bestimmten Zeitpunkt alle ausgefallenen Prozesse zuverlässig und dauerhaft identifiziert.
  - Die letztendliche Genauigkeit des Fehlerdetektors ($\Diamond\mathcal{P}$) bedingt, dass sich alle korrekten Prozesse einigen, da sie ab einem bestimmten Zeitpunkt alle die gleiche Menge von Prozessen als ausgefallen betrachten.
- Performanz
  - Alle Operation sind lokal und damit vernachlässigbar
  - Die Wahl eines neuen Anführers hängt von der Trägheit des Fehlerdetektors ab

Multicast-Kommunikation

- Multicast-Kommunikation bildet einen Basismechanismus zur Realisierung von komplexen verteilten Anwendungen
  - Z.B. für die Bereitstellung von fehlertoleranten verteilten Diensten wie im Rahmen von FT-CORBA skizziert
  - Nachrichten werden hierbei innerhalb einer Gruppe von Prozessen ausgetauscht
  - Dabei können in Abhängigkeit des Systemmodells verschiedene Zustellungsgarantien bezüglich
    - Zuverlässigkeit und
    - Ordnung (bzw. Nachrichtenreihenfolge)

    zugesichert werden

Multicast-Kommunikation

- Zuverlässigkeit
  - Best-Effort Multicast
  - Zuverlässiger Multicast
  - Uniformer Zuverlässiger Multicast
- Ordnung
  - ungeordnet
  - FIFO-Ordnung
  - Kausale Ordnung (wird nicht weiter betrachtet)
  - Totale Ordnung

Zuverlässigkeit

- Schwächste Form des Multicast: „Best-Effort"

Module:
Name: BestEffortBroadcast, **instance** *beb*.

Events:
Request: ⟨ *beb*, *Broadcast* | *m* ⟩: Broadcasts a message *m* to all processes.
Indication: ⟨ *beb*, *Deliver* | p,m ⟩: Delivers a message *m* broadcast by process *p*.

Properties:
BEB1: *Validity*: If a correct process broadcasts a message *m*, then every correct process eventually delivers *m*.

BEB2: No duplication: No message is delivered more than once.

BEB3: No creation: If a process delivers a message *m* with sender *s*, then *m* was previously broadcast by process *s*.

- BEB1 erzwingt Lebendigkeit, wohingegen BEB2 und BEB3 vor allem die Integrität absichern
- Eigenschaften sind ähnlich zu zuverlässigen Punkt-zu-Punkt Verbindungen

! Hinweis: Es wird davon ausgegangen, dass jede Nachricht verschieden ist

Zuverlässigkeit

- Realisierung des „Best-Effort"-Multicast

Implements:
BestEffortBroadcast, **instance** *beb*.

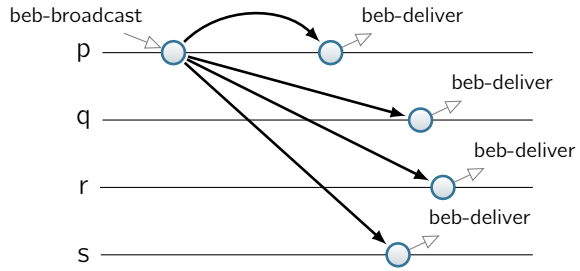
Uses:
PerfectPointToPointLinks, **instance** *pl*.

upon event ⟨ *beb*, *Broadcast* | m ⟩ **do**
  forall $q \in \Pi$ **do**
    trigger ⟨ *pl*, *Send* | q,m ⟩;

upon event ⟨ *pl*, *Deliver* | p, m ⟩ **do**
  trigger ⟨ *beb*, *Deliver* | p,m ⟩;

Zuverlässigkeit

- Ablauf einer „Best-Effort"-Zustellung von Nachrichten



Zuverlässigkeit

- Zuverlässiger Multicast
  - Sollte der Sender von Nachrichten während der Zustellung ausfallen so werden diese Nachrichten entweder an alle oder nicht zugestellt.

Module:
Name: ReliableBroadcast, **instance** *rb*.

Events:
Request: ⟨ *rb*, *Broadcast* | m ⟩: Broadcasts a message *m* to all processes.
Indication: ⟨ *rb*, *Deliver* | p,m ⟩: Delivers a message *m* broadcast by process *p*.

Properties:
RB1: *Validity*: If a correct process *p* broadcasts a message m, then *p* eventually delivers *m*.

RB2: *No duplication*: No message is delivered more than once.

RB3: *No creation*: If a process delivers a message *m* with sender *s*, then *m* was previously broadcast by process *s*.

RB4: *Agreement*: If a message *m* is delivered by some correct process, then *m* is eventually delivered by every correct process.

Zuverlässigkeit

- Realisierung eines zuverlässiger Multicast durch den „Lazy Reliable Broadcast"-Algorithmus
  - Erweiterung zu bisherigen Algorithmen:
    - *[Data,s,m]* – Implementierungsabhängiger Protokollkopf *(Data)* mit Ursprungsabsender *(s)*

Implements:
ReliableBroadcast, **instance** *rb*.

Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨ *rb*, *Init* ⟩ **do**
  delivered := *emptyset*;
  correct := Π;
  forall $p \in \Pi$ **do**
    from[*p*] := $[\emptyset]^N$;

Zuverlässigkeit (Forts.)

upon event ⟨ *rb*, *Broadcast* | m ⟩ **do**
  trigger ⟨ *beb*, *Broadcast* | *[Data,self,m]* ⟩;

upon event ⟨ *beb*, *Deliver* | p, *[Data,s,m]* ⟩ **do**
  if $m \notin$ *delivered* **then**
    delivered := *delivered* $\cup$ *m*;
    trigger ⟨ *rb*, *Deliver* | s,m ⟩;
    from[p] := *from[p]* $\cup$ *{(s,m)}*;
    if $p \notin$ *correct* **then**
      trigger ⟨ *beb*, *Broadcast* | [Data,s,m] ⟩;

upon event ⟨ $\mathcal{P}$, *Crash* | p ⟩ **do**
  correct := *correct* $\setminus$ *p*;
  forall (s,m) $\in$ *from[p]* **do**
    trigger ⟨ *beb*, *Broadcast* | [Data, s,m] ⟩;

- Es gibt zwei Möglichkeiten die zu einer erneuten Zustellung von Nachrichten führen können:
  - Unmittelbar bei Ausfallerkennung
  - Es treffen Nachrichten von einem bereits als ausgefallen erkannten Knoten ein

Zuverlässigkeit

- Korrektheit
  - Die Anforderung keine Nachrichten zu erfinden und Nachrichten zuzustellen wird durch das „Best-Effort Broadcast"-Modul bereits gewährleistet
  - Die Anforderung Nachrichten nicht mehrfach zuzustellen wird durch den Algorithmus selber erreicht
  - Die Einigung der Zustellung einer Nachricht wird durch die allgemeine Zustellungsgarantie den Algorithmus (Behandlung von Ausfällen) und den perfekten Fehlerdetektor erreicht.
- Performance
  - Im fehlerfreien Fall reichen $O(N)$ Nachrichten
  - Im Fehlerfall $O(N)$ Schritte und $O(N^2)$ Nachrichten

Zuverlässigkeit

- Realisierung eines zuverlässiger Multicast durch den „Eager Reliable Broadcast"-Algorithmus im Kontext von „fail silent"-Fehlern
  - „Lazy Reliable Broadcast"-Algorithmus benötigt einen Fehlerdetektor der zwar nicht unbedingt starke Genauigkeit aufweisen muss aber Vollständigkeit
  - Im Kontext von „fail silent"-Fehlern ist dies nicht möglich und entsprechend verzichtet der folgende Algorithmus auf einen Fehlerdetektor

Implements:
ReliableBroadcast, **instance** *rb*.

Uses:
BestEffortBroadcast, **instance** *beb*.

upon event ⟨ *rb*, *Init* ⟩ **do**
delivered := ∅;

upon event ⟨ *rb*, *Broadcast* | m ⟩ do
trigger ⟨ *beb*, *Broadcast* | [Data, self,m] ⟩;

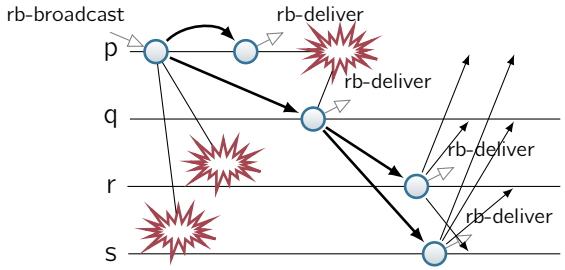
Zuverlässigkeit (Forts.)

upon event ⟨ *beb*, *Deliver* | *[Data, s,m]* ⟩ do
if *m* ∉ *delivered* **then**
delivered := *delivered* ∪ {*m*};
trigger ⟨ *rb*, *Deliver* | s,m ⟩;
trigger ⟨ *beb*, *Broadcast* | [Data, s,m] ⟩;

- Korrektheit
  - Die Einigung der Zustellung einer Nachricht wird durch das sofortige weiterleiten empfangener Nachrichten und die Eigenschaften des „Best-Effort Broadcast"-Moduls erreicht
- Performance
  - Im fehlerfreien Fall reichen $O(N^2)$ Nachrichten
  - Im Fehlerfall $O(N)$ Schritte und $O(N^2)$ Nachrichten

Zuverlässigkeit

- Ablauf einer „ReliableBroadcast"-Zustellung von Nachrichten wenn der Sender einer Nachricht ausfällt.



Zuverlässigkeit

- Uniformer Zuverlässiger Multicast
  - Bisher kann es sein das ein Prozess eine Nachricht lokal zustellt und sie durch einen Ausfall nicht weiter propagiert wird. Dies kann bei aktiver Replikation bereits zu Inkosistenzen führen.
  - Entsprechend fordert der uniforme, zuverlässige Multicast das die Menge der Nachrichten welche von fehlerhaften Prozessen zugestellt werden immer eine Untermenge der von korrekten Prozessen zugestellten Nachrichten ist.

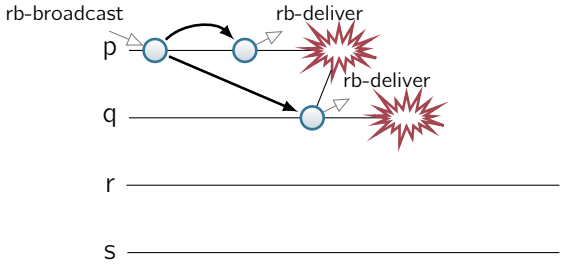
Module:
Name: UniformReliableBroadcast, **instance** *urb*.

Events:
⟨ *urb*, *Broadcast* | m ⟩ and ⟨ *urb*, *Deliver* | p, m ⟩, with the same meaning and interface as in (regular) reliable broadcast.

Properties:
URB1–URB3: Same as properties **RB1–RB3** in (regular) reliable broadcast.
URB4: *Uniform agreement*: If a message m is delivered by some process (whether correct or faulty), then *m* is eventually delivered by every correct process.

Zuverlässigkeit

- Ablauf einer Zustellung wenn eine (uniforme) Einigung der Prozesse nicht erfolgt



Zuverlässigkeit

- Realisierung des uniformen, zuverlässigen Multicast im Kontext von „fail stop"-Fehlern durch die Quittierung von Nachrichten aller fehlerfreien Knoten
  - Ziel ist hierbei die Vermeidung einer verführten lokalen Zustellung von Nachrichten
  - Im Algorithmus wird dies durch die Sammlung vermittelter aber nicht nicht ausreichend quittierter Nachrichten erreicht (*pending*).

Implements:
UniformReliableBroadcast, **instance** *urb*.

Uses:
BestEffortBroadcast, **instance** *beb*.
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨ *urb*, *Init* ⟩ **do**
delivered := ∅;
pending := ∅;
correct := Π;
forall *m* **do** *ack[m]* := ∅;

Zuverlässigkeit (Forts.)

upon event ⟨ *urb*, *Broadcast* | m ⟩ **do**
pending := *pending* ∪ { *(self,m)*};
trigger ⟨ *beb*, *Broadcast* | [Data, *self*,m] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [Data, *s,m*] ⟩ **do**
ack[m] := *ack[m]* ∪ { *p*};
if (*s,m*) ∉ *pending* **then**
pending := *pending* ∪ { *(s,m)*};
trigger ⟨ *beb*, *Broadcast* | [Data, *s*,m] ⟩;

upon event ⟨ $\mathcal{P}$, *Crash* | p ⟩ **do**
correct := *correct* \ {p};

function *candeliver*(*m*) **returns** Boolean **is**
return (*correct* ⊆ *ack[m]*);

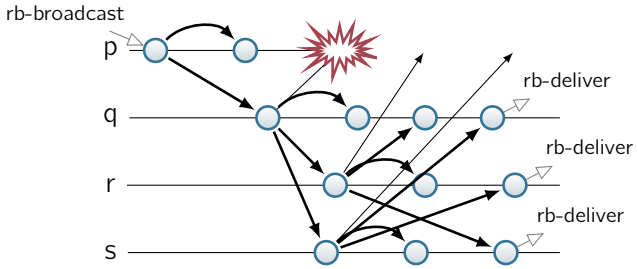
upon exists (*s,m*) ∈ pending such that *candeliver*(*m*) ∧ *m* ∉ *delivered* **do**
delivered := *delivered* ∪ {m};
trigger ⟨ *urb*, *Deliver* | s, m ⟩;

Zuverlässigkeit

- Korrektheit
  - Die Einigungseigenschaft des Algorithmus ist gegeben weil ein Prozess wartet bis er von allen korrekten Knoten eine Quittierung erhält bevor er lokal die Nachricht zustellt. Es kann auf alle korrekten Knoten gewartet werden da der Fehlerdetektor exakt ist.
- Performance
  - Im fehlerfreien Fall sind im ersten Schritt N und dann im zweiten Schritt $N(N-1)$, dies führt zu N^2 Nachrichten.
  - Im Fehlerfall werden bis zu $N+1$ Schritte benötigt.

Zuverlässigkeit

- Ablauf einer uniformen und zuverlässigen Zustellung von Nachrichten wenn alle Knoten den Erhalt von Nachrichten quittieren



Zuverlässigkeit

- Realisierung des uniformen zuverlässigen Multicast im Kontext von fail silent-Fehlern durch die Quittierung einer Mehrheit ($> N/2$) fehlerfreien Knoten

Implements:
UniformReliableBroadcast, **instance** *urb*.

Uses:
BestEffortBroadcast, **instance** *beb*.

// Except for the function *candeliver()* below and for the absence of ⟨ *Crash* ⟩
// events triggered by the perfect failure detector, it is the same as the
// previous algorithm.

function *candeliver*(*m*) **returns** Boolean **is**
return *#(ack[m])* $> N/2$;

Ordnung von Nachrichten

- FIFO-Ordnung
  - Nachrichten, die von einem Knoten in einer bestimmten Reihenfolge erzeugt wurden, werden von anderen Knoten in genau dieser Reihenfolge empfangen/bearbeitet

Module:
Name: FIFOReliableBroadcast, **instance** *frb*.

Events:
⟨ *frb*, *Broadcast* | m ⟩ and ⟨ *frb*, *Deliver* | p,m ⟩, with the same meaning and interface as in (regular) reliable broadcast.

Properties:
FRB1–FRB4: Same as properties **RB1–RB4** in (regular) reliable broadcast.

FRB5: *FIFO delivery*: If some process broadcasts message m_1 before it broadcasts message m_2, then no correct process delivers m_2 unless it has already delivered m_1.

Ordnung von Nachrichten

Implements:
FIFOReliableBroadcast, **instance** *frb*.

Uses:
ReliableBroadcast, **instance** *rb*.

upon event ⟨ *frb*, *Init* ⟩ **do**
lsn := 0;
pending := ∅;
forall $p \in \Pi$ **do** *next[p]* := 1;

upon event ⟨ *frb*, *Broadcast* | m ⟩ **do**
lsn := *lsn* + 1;
trigger ⟨ *rb*, *Broadcast* | [Data, self, m, lsn] ⟩;

upon event ⟨ *rb*, *Deliver* | p,[DATA, s, m, sn] ⟩ **do**
pending := *pending* ∪ $\{(s, m, sn)\}$;
while exists $(s, m', sn') \in$ pending such that $sn' = next[s]$ **do**
$next[s] := next[s] + 1$;
$pending := pending \setminus \{(s, m'; sn')\}$;
trigger ⟨ *frb*, *Deliver* | s, m' ⟩;

Ordnung von Nachrichten

- Einigung
  - Bestimmung eines Wertes aus einer Menge von Vorschlägen
  - Basis zum Erhalt eines gemeinsamen Zustandes innerhalb einer Gruppe von Knoten

Module:
Name: Consensus, **instance** *c*.

Events:
Request: ⟨ *c*, *Propose* | v ⟩: Proposes value *v* for consensus.
Indication: ⟨ *c*, *Decide* | v ⟩: Outputs a decided value *v* of consensus.

Properties:
C1: Termination: Every correct process eventually decides some value.

C2: Validity: If a process decides *v*, then *v* was proposed by some process.

C3: Integrity: No process decides twice.

C4: Agreement: No two correct processes decide differently.

Ordnung von Nachrichten

- „Flooding Consensus"-Algorithmus

Implements:
Consensus, **instance** *c*.

Uses:
BestEffortBroadcast, **instance** beb;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨ *c*, *Init* ⟩ **do**
correct := Π;
round := 1;
decision := ⊥;
receivedfrom := $[\emptyset]^N$; *receivedfrom[0]* := Π;
proposals := $[\emptyset]^N$;

upon event ⟨ $\mathcal{P}$, *Crash* | p ⟩ **do**
correct := correct \ {*p*};

upon event ⟨ *c*, *Propose* | v ⟩ **do**
proposals[1] := proposals[1] ∪ {*v*};
trigger ⟨ *beb*, *Broadcast* | [Proposal,1,proposals[1]] ⟩;

Ordnung von Nachrichten (Forts.)

upon event ⟨ *beb*, *Deliver* | p, [Proposal,r,ps] ⟩ **do**
receivedfrom[r] := receivedfrom[r] ∪ {*p*};
proposals[r] := proposals[r] ∪ ps;

upon correct ⊆ receivedfrom[round] ∧ decision = ⊥ **do**
if receivedfrom[round] := receivedfrom[round-1] **then**
decision := min(proposals[round]);
trigger ⟨ *beb*, *Broadcast* | [Decided, decision] ⟩;
trigger ⟨ *c*, *Decide* | decision ⟩;
else
round := round + 1;
trigger ⟨ *beb*, *Broadcast* ⟩[Proposal, round; proposals[round -1]];

upon event ⟨ *beb*, *Deliver* | p, [Decided, v] ⟩ **such that** $p \in$ correct ∧ decision = ⊥ do
decision := v;
trigger ⟨ *beb*, *Broadcast* | [Decided, decision] ⟩;
trigger ⟨ *c*, *Decide* | decision ⟩;

Ordnung von Nachrichten

- Anmerkungen zum Algorithmus
  - Jeder Knoten macht inital nur einen Vorschlag
  - Ablauf in Runden
  - Ende einer Runde ist erreicht wenn von allen korrekten Knoten eine Nachricht bzgl. dieser Runde erhalten haben
  - Entscheidung erfolgt wenn Menge der korrekten Knoten der aktuellen Runde mit der vorhergehenden übereinstimmt

Ordnung von Nachrichten

- Globale Ordnung von Nachrichten für all Knoten

Module:
Name: TotalOrderBroadcast, **instance** *tob*.

Events:
Request: ⟨ *tob*, *Broadcast* | m ⟩: Broadcasts a message *m* to all processes.
Indication: EventttobDeliver(p,m): Delivers a message *m* broadcast by process *p*.

Properties:
TOB1: Validity: If a correct process p broadcasts a message *m*, then *p* eventually delivers *m*.

TOB2: No duplication: No message is delivered more than once.

TOB3: No creation: If a process delivers a message *m* with sender *s*, then *m* was previously broadcast by process *s*.

TOB4: Agreement: If a message *m* is delivered by some correct process, then *m* is eventually delivered by every correct process.

TOB5: Total order: Let m_1 and m_2 be any two messages and suppose *p* and *q* are any two correct processes that deliver m_1 and m_2. If *p* delivers m_1 before m_2, then *q* delivers m_1 before m_2.

Ordnung von Nachrichten: Totale Ordnung

- „Consensus-based Total Order Multicast"-Algorithmus

Implements:
TotalOrderBroadcast, **instance** *tob*.

Uses:
ReliableBroadcast, **instance** *rb*;
Consensus (multiple instances).

upon event ⟨ *tob*, *Init* ⟩ **do**
  $unordered := \emptyset$;
  $delivered := \emptyset$;
  $round := 1$;
  $wait :=$ False;

upon event ⟨ *tob*, *Broadcast* | *m* ⟩ **do**
  trigger ⟨ *rb*, *Broadcast* | *m* ⟩;

Ordnung von Nachrichten: Totale Ordnung (Forts.)

upon event ⟨ *rb*, *Deliver* | *p*, *m* ⟩ **do**
  if $m \notin delivered$ **then**
    $unordered := unordered \cup \{(p, m)\}$;

upon $unordered \neq \emptyset \wedge wait =$ False **do**
  $wait :=$ True;
  Initialize a new instance *c.round* of consensus;
  trigger ⟨ *c.round*, *Propose* | *unordered* ⟩;

upon event ⟨ *c.r*, *Decide* | *decided* ⟩ **such that** $r = round$ **do**
  forall $(s, m) \in sort(decided)$ **such that** $m \notin delivered$ **do** // by the sorted order
    trigger ⟨ *tob*, *Deliver* | *s*, *m* ⟩;
    $delivered := delivered \cup \{m\}$;
    $unordered := unordered \setminus \{(s, m)\}$;
  $round := round + 1$;
  $wait :=$ False;

Zusammenfassung

- Pseudocode und Modularisierung als Basis zu Erfassung von verteilten Algorithmen
- Ausgehend von Punkt-zu-Punkt Verbindung zu Multicast
  - Zuverlässigkeit
  - Ordnung von Nachrichten

- In Summe aber nur Einzelbeispiele und weiter Vertiefung nötig um Anwendungen in der Praxis fehlertolerant auszugestalten .

- Referenz

C. Cachin, R. Guerraoui, and L. Rodrigues.
Introduction to Reliable and Secure Distributed Programming (2. ed.).
Springer, 2011.