Verteilte Sensornetzwerke

Mit Datenaggregation und Sensorfusion

PD Stefan Bosse

Universität Bremen - FB Mathematik und Informatik

Gruppenkommunikation in Sensornetzwerken

Es sollen grundlegende Prinzipien der höheren Gruppenkommunikation betrachtet werden.

Tupelräume

Einzelne Sensorknoten können organisatorische Strukturen bilden, z.B., einen Konsens über eine sensorische Information (Entscheidung) finden

Verhalten und Entscheidungsfindung bei Fehlern (Ausfall von Knoten, Nachrichtenverlust, Verfälschung)

Übergang von passiven zu aktiven Nachrichten (Agenten)

Tupelräume

- Tupel-Räume stellen ein **assoziiertes Shared-Memory-Modell** dar, wobei die gemeinsam genutzten Daten als **Objekte** mit einer Reihe von **Operationen** betrachtet werden, die den Zugriff der Datenobjekte unterstützen
- Tupel sind in **Räumen** organisiert, die als abstrakte Berechnungsumgebungen betrachtet werden können.
- Ein Tupelraum verbindet verschiedene Programme, die **verteilt** werden können, wenn der Tupel-Space oder zumindest sein operativer Zugriff verteilt ist.
    - Oder: **Mobile Sensorknoten** als Tupel Verteiler!
- Das Tupelraum Organisations- und Zugangsmodell bietet **generative Kommunikation**, d.h. Datenobjekte können in einem Raum durch Prozesse mit einer Lebensdauer über das Ende des Erzeugungsprozesses hinaus gespeichert werden.
- Ein bekanntes Tupelraum-Organisations- und Koordinationsparadigma ist **Linda** [GEL85].

Tupelräume

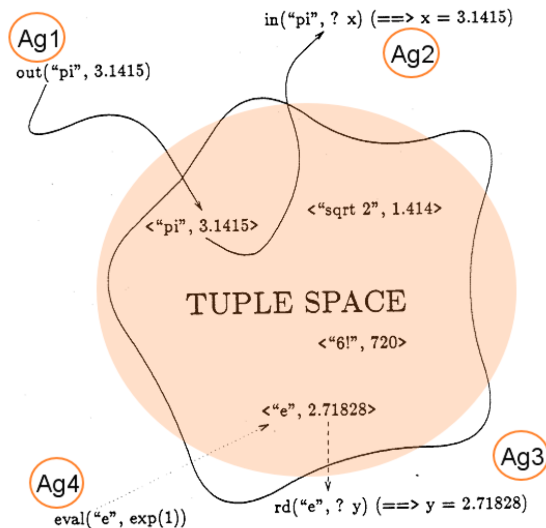


Abb. 1. Ein Schnappschuss eines Tupelraumes mit Tupeln und Tupeloperationen [11]

Tupelräume

- Kommunikation von Sensorknoten über Tupelräume ist eine **Koordionierungssprache**.

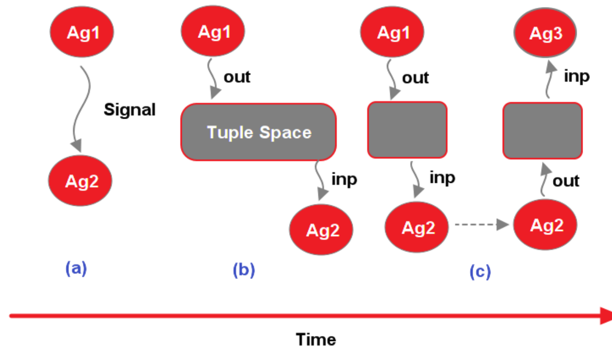


Abb. 2. Direkter Nachrichtenaustausch (a), z.B. durch Signale, im Vergleich zu generativer Kommunikation (b) und virtuelle verteilte Räume (c) durch mobile Prozesse (Sensorknoten)

Tupelräume - Datenmodell

- Die Daten sind mit Tupeln organisiert.
- Ein Tupel ist eine lose gekoppelte Verbindung einer beliebigen Anzahl von Werten beliebiger Art /Typ/
- Ein Tupel ist ein Wert und sobald es in einem Tupelraum gespeichert ist, ist es persistent.
- Tupeltypen ähneln den Datenstrukturtypen, sie sind jedoch dynamisch und können zur Laufzeit ohne statische Beschränkungen erstellt werden.
- Auf die *Elemente von Tupeln* kann nicht direkt zugegriffen werden, was üblicherweise Mustererkennung und *musterbasierte Dekomposition* erfordert, im Gegensatz zu Datenstrukturtypen, die einen benannten Zugriff auf Feldelemente bieten, obwohl die Behandlung von Tupeln als Arrays oder Listen diese Beschränkung lösen kann.
- Ein Tupel mit n Feldern heißt n-stellig und wird in der Notation $<v_1, v_2, ..>$ angegeben.

Tupelräume - Datenmodell

Beispiele

```
<'SENSOR',1000>
<'SENSOR2',[10,100,2]>
<1,3,5,7,11>
<'SLEEPMODE',True,2500.34>
<0,'OFF'>
<1,'ON'>
```

- Formal werden Tupel als **Vektoren** durch die folgende Generierungsregel mit *Werten* v, *Ausdrücken* ε und *Variablen* x definiert, die als tatsächliche Parameter betrachtet werden (d.h. Variablen x, die mit Wertsemantik verwendet werden):

$$t = \left\langle \overrightarrow{d} \right\rangle, \text{with } \overrightarrow{d} ::= d | d, \overrightarrow{d} \text{ and } d ::= v | \varepsilon | x$$

Tupelräume - Datenmodell

- Tupelwerte erfordern einen **Mustervergleich** basierend auf dem *Vorlagenmuster* mit der folgenden Generierungsregel, bestehend aus tatsächlichen (v, ε, x) und formalen Parametern (x?, Variablen, die mit Referenzsemantik verwendet werden):

$$p = \left\langle \overrightarrow{dt} \right\rangle, \text{with } \overrightarrow{dt} ::= dt|dt, \overrightarrow{dt} \text{ and } dt ::= v|\varepsilon|x|x?|\perp$$

- Ein Suchmuster kann ein Wildcard (⊥) anstelle von formalen Parametern verwenden.

- Jedes Tupel t hat eine Typsignatur Sig (t) = S_t = <T_1; T_2; ...; T_n>, ein Tupel mit der gleichen Stelligkeit wie t, das den Typ jedes Tupelfeldes angibt.

- Ein Tupel kann nur durch seine Verknüpfung mit Templates p angesprochen werden.

Tupel Räume - Datenmodell

- Üblicherweise wird das **erste Feld** eines Tupels als symbolischer **Schlüssel** behandelt, der eine Tupelunterklasse identifiziert, indem Textzeichenfolgen oder abzählbare symbolische Werte (numerisch) verwendet werden.

Mustersuche

Sei $t = <d_1, d_2, .., d_n>$ ein Tupel, $p = <dt_1, dt_2, .., dt_m>$ eine Vorlage; dann wird t durch p abgedeckt (bezeichnet durch match (t, p) = true), wenn die folgenden Bedingungen gelten: (i) m = n. (ii) $\forall\ dt_i = d_i$ oder $dt_i = \perp$, $1 \leq i \leq n$. Bedingung (1) prüft, ob t und p die gleiche Stelligkeit haben, während (2) prüft, ob jedes Nicht-Wildcard-Feld von p gleich ist dem entsprechenden Feld von t.

Def. 1. Mustersuche

Tupelräume - Operationale Semantik

- Es gibt eine Reihe von Operationen, die von Prozessen angewendet werden können, bestehend aus
  - einer Reihe reiner Datenzugriffsoperationen, die Tupel als passive Datenobjekte behandeln,
  - und Operationen, die Tupel als eine Art von aktiven Rechenobjekten behandeln (genauer gesagt, zu berechnende Daten).
  - RPC-Semantik (Remote Procedure Call).

out(*t*)
: Die Ausführung der Ausgabeoperation fügt das Tupel t in den Tupelraum ein. Mehrere Kopien desselben Tupelwerts können eingefügt werden, indem die Ausgabeoperation iterativ angewendet wird. Die gleichen Tupel können nach dem Einfügen in den Tupelraum nicht unterschieden werden.

Beispiel: `out("Sensor",1,100); out("Sensor",2,121);`

Tupelräume - Operationale Semantik

inp(*p*)

Die Ausführung der Eingabeoperation entfernt ein Tupel *t* aus dem Tupelraum, der der Mustervorlage *p* entspricht. Wenn kein passendes Tupel gefunden wird führt das zu einer Blockierung des aufrufenden Prozesses bis ein passendes Tupel eingestellt wird.

Beispiel: `inp("Sensor",1,s1?); inp("Sensor",i?,s?);`

rd(*p*)

Die Ausführung der Leseoperation gibt eine Kopie eines Tupels *t* zurück, dass der Vorlage *p* entspricht, entfernt sie jedoch nicht. Wenn kein passendes Tupel gefunden wird führt das zu einer Blockierung des aufrufenden Prozesses bis ein passendes Tupel eingestellt wird.

Beispiele: `rd("Sensor",1,s1?); rd("Sensor",i?,s?);`

Tupelräume - Operationale Semantik

inp?(*p*), rd?(*p*)
: Nichtblockierende Version von *inp*/*rd*. Wird kein passendes Tupel gefunden wird die Operation ergebnislos terminiert.

Beispiel: `res:=inp?('SENSOR',a?,b?);`

inpw?(*tmo*,*p*), rdw?(*tmo*,*p*)
: Teilblockierende Version von *inp*/*rd*, Wird einer Zeit von *tmo* kein passendes Tupel gefunden wird die Operation abgebrochen.

Beispiel: `res:=inpw?(1000,'SENSOR',a?,b?);`

- Die Verwendung von zeitlich unbegrenzt blockierenden Operationen kann unter Betrachtung der Lebendigkeit von Agenten nachteilig sein. Daher sollte immer eine zeitliche Begrenzung und anschließende Abfrage des Operationsstatus erfolgen (abgebrochen?)

Tupelräume - Operationale Semantik

test(t), testandset(*p*,function (*t*)→*t*)

Nicht blockierender Test eines Tupels und atomare Veränderung eines Tupels, dass der Vorlage *p* entspricht. Das zweite Argument ist eine Abbildungsfunktion. Das Ergebnistupel ersetzt das ursprüngliche.

Markierungen

- Tupel sind persistent und können für immer in einem Tupelraum verbleiben!
- Daher ist die Verwendung von *Markierungen* häufig sinnvoll.
- Eine Markierung ist ein Tupel mit einer Lebenszeit τ
- Nach Ablauf der Lebenszeit wird das Tupel - sofern es nicht entfernt wurde - durch einen Garbagecollector entfernt.

$$m = \left\langle \tau, \overrightarrow{d} \right\rangle, \text{with } \overrightarrow{d} ::= d|d, \overrightarrow{d} \text{ and } d ::= v|\varepsilon|x,\ \tau : \text{timeout}$$

mark(*tmo*,*t*)

Ausgabe eines Tupels *t* mit einer Lebenszeit τ (im Tupelraum).

Tupelräume - Operationale Semantik

eval(*p*)

Diese Operation ermöglicht die Injektion von **aktiven Tupeln**, die derzeit nicht vollständig ausgewertet sind, indem ein erweitertes funktionales Tupel t verwendet wird (mit erweitertem dt $::= v \mid \varepsilon \mid x \mid f(x)$ mit einem Funktionsargument). *Das Tupel wird erst bei Bedarf in einem eigenen Prozess ausgewertet (i.A. durch inp oder rd Operation initiiert)*

Diese Operation nimmt eine Funktion $f(x)$ an, die in den Prozessen vorhanden ist, die am Tupleraum teilnehmen und die für die vollständige Berechnung dieser Tupel verwendet werden kann.

Tupelräume - Operationale Semantik

Die *eval* Semantik kann kompliziert sein (Seiteneffekte im Ausführungsprozess, Funktionsschnittstellen, erwartete Datentypen usw.)

Alternative Implementierung mit *eval* (Klientenseite) und *listen* und *reply* Operationen (Serverseite):

```
P1: eval("square",2,y?)
P2: def sq = fun x -> x*x;
    listen("square",x?,?);
    y=sq(x);
    reply("square",x,y);
```

Tupelräume - Synchronisationsmodell

- Es gibt **Produzenten- (Generator) und Verbraucherprozesse**.

1. Ein *Produzent* erzeugt ein Tupel, das von einem Konsumentenprozess entfernt werden kann.
    - Die Tupelausgabeoperation endet unmittelbar (asynchron), alternativ nachdem das Tupel im Tupelraum gespeichert wurde (synchron).
2. Ein Verbraucher-Prozess wird blockiert, wenn die Anfrage nicht bearbeitet werden kann, wenn im Tupel-Bereich tatsächlich kein passendes Tupel vorhanden ist.
3. Nachdem ein übereinstimmendes Tupel im Tupelraum gespeichert wurde, wird es sofort einen der wartenden Verbraucherprozesse zugewiesen.

Tupelräume - Synchronisationsmodell

- Es gibt **Produzenten- (Generator) und Verbraucherprozesse**.

1. Ein *Produzent* erzeugt ein Tupel, das von einem Konsumentenprozess entfernt werden kann.
    - Die Tupelausgabeoperation endet unmittelbar (asynchron), alternativ nachdem das Tupel im Tupelraum gespeichert wurde (synchron).
2. Ein Verbraucher-Prozess wird blockiert, wenn die Anfrage nicht bearbeitet werden kann, wenn im Tupel-Bereich tatsächlich kein passendes Tupel vorhanden ist.
3. Nachdem ein übereinstimmendes Tupel im Tupelraum gespeichert wurde, wird es sofort einen der wartenden Verbraucherprozesse zugewiesen.

- Daher ist die Eingabeoperation immer synchron. Einzige Ausnahme sind die nicht permanent blockierenden Versionen, die das Warten auf eine obere Zeitgrenze begrenzen (Timeout).

- Es gibt keine anfängliche zeitliche Anordnung von Erzeuger- und Verbraucheroperationen.

Tupelräume - Beispiele

```
out(['SENSOR',10,20]);
out(['SENSOR',9,23]);
out(['PI',3,14]);
out(['DATA',[1,2,3,4]]);
inp(['SENSOR',_,_]);
>> [ 'SENSOR', 9, 23 ]
inp(['SENSOR',_,_]);
>> [ 'SENSOR', 10, 20 ]
inp(['SENSOR',_,_]);
>> null
rd([_,_])
>>[ 'DATA', [ 1, 2, 3, 4 ] ]
```

Verteilte Tupelräume

- Die *Verteilung* von Tupel-Räumen auf verschiedenen Rechnerknoten impliziert *Synchronisationsprobleme* und erfordert normalerweise eine zuverlässige *Gruppenkommunikation*, die in Rechnernetzwerken nicht erwartet werden kann.

- Die Verteilung von Tupel-Räumen bedeutet die Verteilung und asynchrone Ausführung einer Menge von Tupelraum-Servern anstelle eines einzelnen Servers.

- Ein Tupelraum-Server bietet die notwendige Koordination für gleichzeitige oder verschachtelte In / Out-Anfragen.
    - Die Verteilung der Server führt zu einer Verteilung der Koordination.
    - Dieses Problem kann jedoch gelöst werden, indem der Tupelraum in **Unterräume** partitioniert wird und jeder Unterraum auf einem anderen Knoten von einem Server bedient wird.
    - Problem: Tupel sind nicht gleichmäßig verteilt → schlechtes Load Balancing!

Verteilte Tupelräume

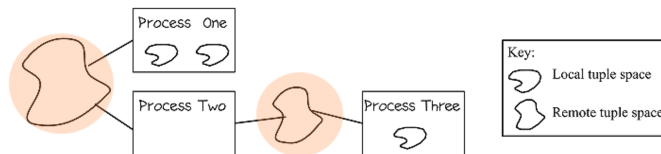


Abb. 3. Zusammenhang von lokalen und entfernten (verteilten) Tupelräumen

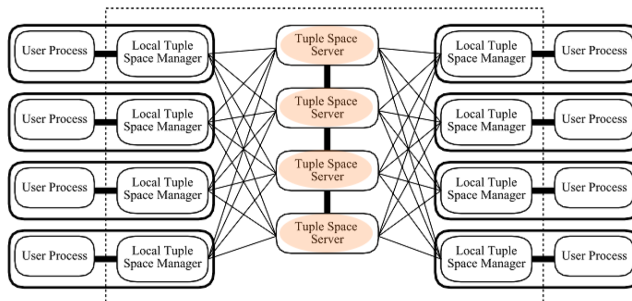


Abb. 4. Lokale und multiple globale Tupelraumserver

Gruppenentscheidung und Verhandlung

- In Sensornetzwerken gibt es in der Regel **Organisationsstrukturen**
- In diesen Strukturen sollen **gemeinsame Ziele** entweder vorgegeben und umgesetzt oder verhandelt werden.

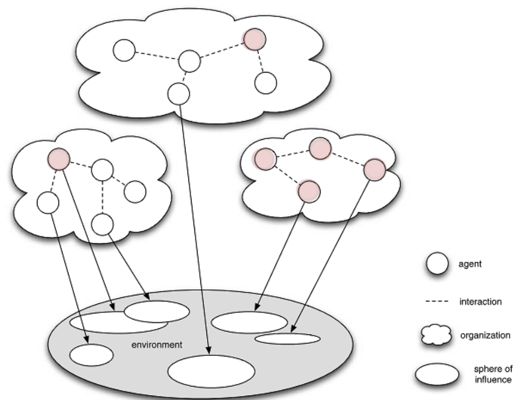


Abb. 5. Typische Gruppenstrukturen in Sensornetzwerken

Gruppenentscheidung und Verhandlung

- Dabei können Gruppenentscheidungen und Verhandlungen auf Basis von **Nützlichkeitsfunktionen** geführt werden
- Es sei $\boldsymbol{G}$ eine Gruppe Prozessen (Sensorknoten) $\boldsymbol{G}=\{P_1,P_2,..,P_m\}$
- Es gibt nun verschiedene Stufen der Verhandlung:
  - Wahl eines Gruppenführers (Leader) oder Mediators
  - Abgabe von Stimmen / Absichten
  - Bestimmung eines gemeinsamen Ergebnisses mittels Konsensalgorithmus bzw.
  - Wahl: Über Pluralität, z.B. mit einer Mehrheitsentscheidung
  - Benachrichtigung der Gruppenteilnehmer über Ergebnis
- **Nachteil des Mehrheitsentscheids**: Das Volk ist dumm! D.h. es könnte eine besseres Ergebnis für ein MAS erzielt werden, wenn Fraktionen in den Stimmabgaben berücksichtigt werden würden (differenziertes und gewichtetes Ergebnis) ...

Verhandlung und Abstimmung

- Verhandlung in Gruppen und Abstimmung über ein gemeinsames Ergebnis (Konsens) ist ein weiteres wichtiges Beispiel für verteiltes Gruppenverhalten → Kommunikation!
- Ein Verhandlungsproblem ist ein Problem, bei dem mehrere Prozesse versuchen, zu einer Vereinbarung oder einem Deal zu kommen.
- Es wird angenommen, dass jeder Prozess gegenüber allen möglichen Deals eine Präferenz hat.
- Die Prozesse senden sich Nachrichten in der Hoffnung, einen Deal zu finden, auf den sich alle Agenten einigen können.

- Diese Prozesse stehen vor dem Problem:
  - Sie wollen ihren eigenen Nutzen maximieren, sehen sich aber auch der Gefahr eines Zusammenbruchs der Verhandlungen oder des Ablaufs einer Frist für die Vereinbarung gegenüber.
  - Daher muss jeder Prozess sorgfältig verhandeln und jeden Nutzen abwägen, den er aus einem Versuch gegen einen möglicherweise besseren Abschluss oder das Risiko eines Ausfalls bei den Verhandlungen zieht.

Verhandlung und Abstimmung

Automatisierte Aushandlung kann in Sensornetzwerken sehr nützlich sein, da sie eine verteilte Methode zur Aggregation von verteiltem Wissen bietet.

- Verschiedene Protokolle existieren, z.B.,
  - Monotone Konzession
  - Monotone Konzession mit zusätzlicher Risikobewertung
  - Schrittweise Verhandlung

- Häufig in der Form (Monotone Konzession, Vidal,2010)

```
0. δi <- arg maxδ ui(δ) // Maxmimize utility
1. Propose δi
2. Receive δj proposal
3. if ui(δj) > ui(δi)
4.   then Accept δj
5.   else δi <- δi' such that uj(δi') >= e+uj(δi)
6. loop 2.
```

- mit $u_i(\delta)$: Nützlichkeitsfunktion eines Deals δ des i-ten Knoten/Prozesses

Verteilter Konsens

- Ein verteilter Konsensalgorithmus hat das Ziel in einer Gruppe von Prozessen oder Sensorknoten eine gemeinsame Entscheidung zu treffen
- Zentrale Eigenschaften:
  - Zustimmung/Übereinstimmung
  - Terminierung; Lebendigkeit und Deadlockfreiheit
  - Gültigkeit; Robustheit gegenüber Störungen wie fehlerhaften Nachrichten oder Ausfälle von Gruppenteilnehmern

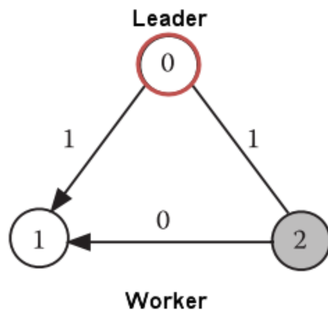
- Beim Konsens kann ein Master-Slave Konzept oder ein Gruppenkonzept mit Leader/Commander und Workern verwendet werden.
  - Beim Master-Slave Konzept kommunizieren nur Slaves mit dem Master
  - Bei Gruppenkonzept (i.A. mit einem Leader) kommunizieren auch alle Gruppenteilnehmer untereinander
- Durch Störung (Fehler oder Absicht) kann es zu fehlerhaften bis hin zu fehlgeschlagenen Konsens kommen.

Verteilter Konsens

- Zwei Bedingungen für Interaktive Konsistenz (zu erfüllen):
  - IC1: Jeder Worker empfängt die *gleiche* Anweisung vom Leader!
  - IC2: Wenn der Leader *fehlerfrei* arbeitet, dann empfängt jeder *fehlerfreie* Worker die Anweisung die der Leader sendete!

Byzantinisches Generalproblem

- Beispiel: In einer Gruppe aus drei Prozessen/Agenten ist einer fehlerhaft bzw. versendet fehlerhafte Nachrichten (durch Störung oder Absicht) mit Anweisungen 0/1 (schließlich dann ein Konsensergebnis) [12]



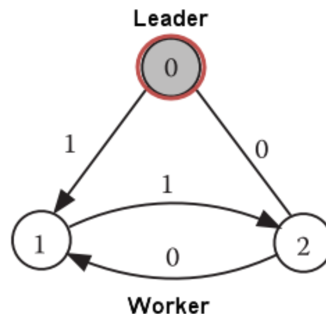


Abb. 6. Byzantinisches Generalproblem: (a) Leader 0 ist fehlerfrei, Worker 2 ist fehlerhaft (b) Leader 0 ist fehlerhaft, Worker 1 und 2 sind fehlerfrei [12]

Verteilter Konsens

- Jeder Worker der Nachrichten empfängt ordnet diese nach direkten und indirekten (von Nachbarn)
- **Fall (a)**: Prozess 2 versendet fehlerhafte Nachricht mit Anweisung 0, Prozess 1 empfängt eine direkte Nachricht mit Anweisung 1 und eine indirekte mit (falschen) Inhalt Anweisung 0
  - Bedingung IC1 ist erfüllt. Um Bedingung IC2 zu erfüllen wird Worker 1 die direkte Anweisung 1 von Prozess 0 (Leader) auswählen → Konsens wurde gefunden
- **Fall (b)**: Prozess 0 (Leader) versendet an Prozess 1 richtige Nachricht mit Anweisung 1 und falsche Nachricht mit Anweisung 0 an Prozess 1
  - Würde Prozess 1 wieder zur Erfüllung von IC2 eine Entscheidung treffen (Anweisung 1 auswählen), dann wäre IC1 verletzt. Wie auch immer Prozess 1 entscheidet ist entweder IC1 oder IC2 verletzt → **Unentscheidbarkeit** → Kein Konsens möglich

Verteilter Konsens

Das nicht-signierte Nachrichtenmodell erfüllt die Bedingungen:

1. Nachrichten werden während der Übertragung nicht verändert (aber keine harte Bedingung).
2. Nachrichten können verloren gehen, aber die Abwesenheit von Nachrichten kann erkannt werden.
3. Wenn eine Nachricht empfangen wird (oder ihre Abwesenheit erkannt wird), kennt der Empfänger die Identität des Absenders (oder des vermeintlichen Absenders bei Verlust).

- Algorithmen zur Lösung des Konsensproblems müssen m fehlerhafte Prozesse annehmen (bzw. fehlerhafte Nachrichten)

Der OM(m) Algorithmus

Entscheidungsfindung in mehreren Phasen (m)

- Ein Algorithmus der einen Konsens erreicht bei Erfüllung der Bedingungen IC1 und IC2 mit bis zu m fehlerhaften Prozesse bei insgesamt n ≥ $3m+1$ Prozessen mit nicht signierten ("mündlichen") Nachrichten. Grundlegend:
  1. Leader i sendet einen Wert $v \in \{0, 1\}$ an jeden Worker $j \neq$ i.
  2. Jeder Worker j akzeptiert den Wert von i als Befehl vom Leader i.

Verteilter Konsens

1. Leader i sendet einen Wert $v \in \{0, 1\}$ an jeden Worker $j \neq$ i.
2. Wenn $m > 0$, dann beginnt jeder Worker j, der einen Wert vom Leader erhält, eine neue Phase (m-1), indem er ihn mit OM(m-1) an die verbleibenden Worker sendet.
    - In dieser Phase fungiert j als Leader.
    - Jeder Worker erhält somit (n-1) Werte: (a) einen Wert, der direkt von dem Leader i von OM (m) empfangen wird und (b) (n-2) Werte, die indirekt von den (n-2) Workern erhalten werden, die aus ihrem Broadcast OM(m-1) resultieren.
    - Wird ein Wert nicht empfangen wird er durch einen Standardwert ersetzt.
3. Jeder Worker wählt die **Mehrheit** der (n-1) Werte, die er erhält, als Anweisung vom Leader i.

Def. 2. Algorithmus OM(m)

Verteilter Konsens

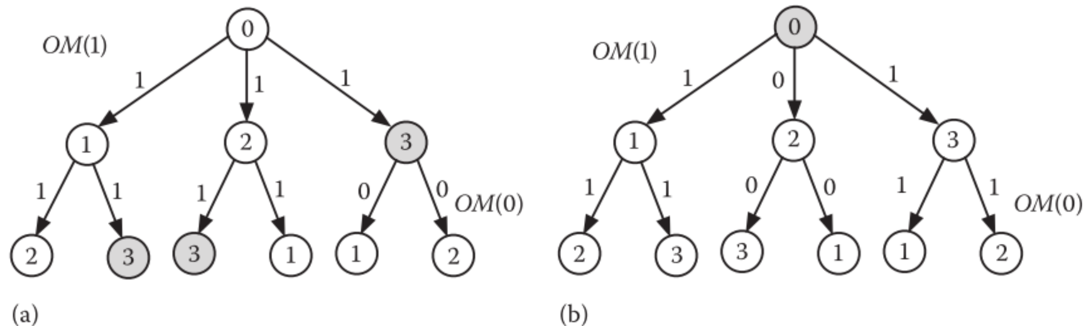


Abb. 7. Eine Illustration von OM(1) mit vier Prozessen und einem fehlerhaften Prozess: die Nachrichten auf der oberen Ebene spiegeln die Eröffnungsnachrichten von OM(1) wider und die auf der unteren Ebene spiegeln die OM(0)-Meldungen wider, die von den Mitteilungen der oberen Ebene ausgelöst werden. (a) Prozess 3 ist fehlerhaft. (b) Prozess 0 (Leader) ist fehlerhaft.[12]

Kommunikationsarchitektur

```
DistributedAlg:
  while not terminated
    send (message)
    container = receive (N messages)
    make decision :- Next round? Succes? Failure? Termination?
  end
end
```

Wo liegen die Probleme in send-receive Paar Schleifen? Was ist die grundlegende Annahme, und wie sieht die Realität aus?

Problem: Ein Prozess/Knoten sendet bevor ein anderer Knoten aus Nachrichten wartet.

- Prinzipiell kann vor der Sendung ein Nachrichtenkontainer mit Pufferung Q(ueue) eingerichtet werden.

Aber: Auch dieser Zeitpunkt kann nach der Sendung von Nachrichten von anderen Teilnehmern liegen

Problem: Ein Prozess/Knoten sendet bevor ein anderer Knoten aus Nachrichten wartet.

- Prinzipiell kann vor der Sendung ein Nachrichtenkontainer mit Pufferung Q(ueue) eingerichtet werden.

Aber: Auch dieser Zeitpunkt kann nach der Sendung von Nachrichten von anderen Teilnehmern liegen

Verteilte Algorithmen basieren häufig auf der Annahme synchroner Systeme und synchronen Nachrichtenaustausch. Realität ist strikt asynchron.

Verteilter Konsens

Der Paxos Algorithmus

- Paxos ist ein Algorithmus zur Implementierung von fehlertoleranten Konsensfindungen.
- Er läuft auf einem *vollständig verbundenen Netzwerk* von n Prozessen und toleriert bis zu m Ausfälle, wobei $n \geq 2m+1$ ist.
- Prozesse können abstürzen und Nachrichten können verloren gehen, byzantinische Ausfälle (absichtliche Verfälschung) sind jedoch zumindest in der aktuellen Version ausgeschlossen.
- Der Algorithmus löst das Konsensproblem bei Vorhandensein dieser Fehler auf einem *asynchronen System von Prozessen*.

- Obwohl die Konsensbedingungen Zustimmung, Gültigkeit und Terminierung sind, garantiert Paxos in erster Linie die Übereinstimmung und Gültigkeit und nicht die Beendigung - es ermöglicht die Möglichkeit der Beendigung nur dann, wenn es ein ausreichend langes Intervall gibt, in dem kein Prozess das Protokoll neu startet.

Verteilter Konsens

- Ein Prozess kann drei verschiedene Rollen wahrnehmen (neben Initiator):
    - Antragsteller (proposer),
    - Akzeptor (aceptor) und
    - Lerner (learner).

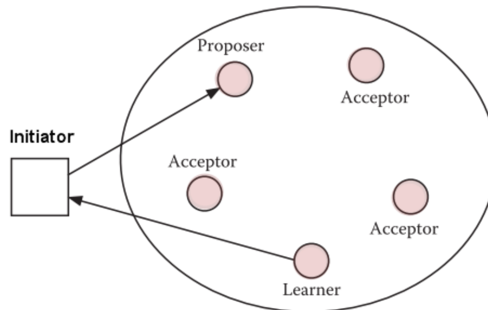


Abb. 8. Typische Rollenverteilung beim Paxos Algorithmus

Verteilter Konsens

- Die **Antragsteller** reichen die vorgeschlagenen Werte im Namen eines Initiators ein,
- die **Akzeptoren** entscheiden über die Kandidatenwerte für die endgültige Entscheidung und
- die **Lernenden** sammeln diese Informationen von den Akzeptoren und melden die endgültige Entscheidung dem Initiator zurück.

- Ein Vorschlag, der von einem Antragsteller gesendet wird, ist ein Tupel (v, n), wobei v ein Wert und n eine Sequenznummer ist.

- Wenn es nur einen Akzeptor gibt, der entscheidet, welcher Wert als Konsenswert gewählt wird, dann wäre diese Situation zu einfach. Was passiert, wenn der Akzeptor abstürzt? Um damit umzugehen, gibt es mehrere Akzeptoren.

- Ein Vorschlag muss von mindestens einem Akzeptor bestätigt werden, bevor er für die endgültige Entscheidung in Frage kommt.

Verteilter Konsens

Die Sequenznummer wird verwendet, um zwischen aufeinander folgenden Versuchen der Protokollanwendung zu unterscheiden.

- Nach Empfang eines Vorschlags mit einer größeren Sequenznummer von einem gegebenen Prozess, verwerfen Akzeptoren die Vorschläge mit niedrigeren Sequenznummern.
- Schließlich akzeptiert ein Akzeptor die Entscheidung der Mehrheit.

Phasen des Paxos Algorithmus

1. **Die Vorbereitungsphase**
    - Jeder Antragsteller sendet einen Vorschlag (v, n) an jeden Akzeptor
    - Wenn n die größte Sequenznummer eines von einem Akzeptor empfangenen Vorschlags ist, dann sendet er ein *ack* $(n, \bot, \bot)$ an seinen Antragsteller
    - Hat der Akzeptor einen Vorschlag mit einer Sequenznummer $n' < n$ und einem vorgeschlagenen Wert v akzeptiert, antwortet er mit $ack(n, v, n')$.

2. **Aufforderung zur Annahme eines Eingabewertes**
   - Wenn ein Antragsteller $ack(n, \perp, \perp)$ von einer Mehrheit von Akzeptoren empfängt, sendet er an alle Akzeptoren $accept(v, n)$ und fordert sie auf, diesen Wert zu akzeptieren.
   - Wenn ein Akzeptor in Phase 1 einen $ack(n, v, n')$ an den Antragsteller zurücksendet, muss der Antragsteller den Wert v mit der höchsten Sequenznummer in seiner Anfrage an die Akzeptoren einbeziehen.
   - Ein Akzeptor akzeptiert einen Vorschlag (v, n), sofern er nicht bereits zugesagt hat, Vorschläge mit einer Sequenznummer größer als n zu berücksichtigen.

3. **Die endgültige Entscheidung**
    - Wenn eine Mehrheit der Akzeptoren einen vorgeschlagenen Wert akzeptiert, wird dies der endgültige Entscheidungswert. Die Akzeptoren senden den akzeptierten Wert an die Lernenden, wodurch sie feststellen können, ob ein Vorschlag von einer Mehrheit von Akzeptoren akzeptiert wurde.

Divide-and-Conquer

Replikation

- Replikation dient:
    - Der Gruppenbildung mit Gemeinsamkeiten,
    - Der Vererbung von Verhalten und Spezialisierung
    - Der räumlichen Exploration
    - ..

Beim Divide-and-Conquer" (Teile und herrsche) Ansatz wird versucht ein komplexes Problem soweit rekursiv zu zerlegen dass am Ende nur noch triviale Probeleme übrig bleiben!

- Beispiel: Verteiltes Sensornetzwerk und Bestimmung von geometrisch ausgedehnter Sensoraktivität und Sensorfusion

Verteilter Informationsaustausch

Problem: Wie können Informationen von der Informationsquelle zu Informationssenken, d.h. Knoten die an den Informationen interessiert sind, zugestellt werden?

- Die Replikation kann verwendet werden, um die Zustellungswahrscheinlichkeit zu erhöhen und die Latenz zu verringern.
- Aktive Nachrichten mit Agenten und Lernende Agenten können die Pfadsuche verbessern, indem sie ihre Reisegeschichte berücksichtigen.
- Datenzentrierte gerichtete Diffusionsalgorithmen können leicht mit autonomen mobilen Agenten implementiert werden.
- *Problem: Flutung des Netzwerks mit Agenten!*

Ereignisbasierte Verteilung

Ein Ereignisbenachrichtigungsalgorithmus kann verwendet werden um effizient eine Kommunikation zwischen Informationsquellen und Senken herzustellen.

- Dazu können Benachrichtigungsagenten verwendet werden die entlang eines Pfades Ereignisse markieren, z.B. dass ein Sensorwert über einer Schwelle liegt.

Agenten sind im einfachsten Fall Nachrichten die Aktionen auslösen, ihren Inhalt verändern können, und weitergesendet werden können.

Verteilter Informationsaustausch

- Suchagenten werden dann benutzt die Ereignisse zu finden die von den Benachrichtigungsagenten hinterlassen wurden.
- Der Ursprung eines Ereignisses kann durch Rückverfolgung des Pfades gefunden werden.

Random Walk

- Eine einfache Möglichkeit besteht darin, dass der Weg zum Empfänger (Datensenke) durch zufällige Richtungswechsel und Migration von datentragenden Agenten erfolgt.
  - Es wird kein geometrisches Modell und Kenntnis des Netzwerkes benötigt

Gerichtete Diffusion

- Durch Replikation der Information bzw. der datentragenden Agenten und grob richtungsorienterte Zustellung mit überlagerten Random Walk und/oder ggf. Backtracking um tote Netzwerkenden (Seitenarme) zu entkommen.
  - Dazu wird partielles geometrisches Modell des Netzwerkes benötigt

Verteilter Informationsaustausch

Potentialfeldansatz

- Eine weitere Möglichkeit besteht darin dass dateninteressierte Knoten "farbige" Markierungen in ihrer Umgebung mit Gradienten verteilen
  - Die "Farbe" charakterisiert die Informations/Datenart, z,B. Temperaturssensor, Ortsinformation, usw.
- Ein datentragendee Nachricht oder ein Agent wird entlang des Gradienten der Markierungen einen Weg zur Datensenke finden

Verteilte Mustererkennung in Sensornetzwerken

Ausgangssituation: Verteiltes Sensornetzwerk mit Knoten in einem Maschengitternetzwerk und Verwendung von aktiven Nachrichten (Agenten).

- Fehlerhafte oder verrauschte Sensoren können Datenverarbeitungsalgorithmen erheblich stören.
- Es ist notwendig, fehlerhafte Sensoren von gut arbeitenden Sensoren zu isolieren.
- Üblicherweise werden Sensorwerte innerhalb eines räumlich nahen Bereichs korreliert, beispielsweise in einem räumlich verteilten mechanischen Lastmonitoringnetzwerks unter Verwendung von Dehnungssensoren.

Verteilte Mustererkennung in Sensornetzwerken

- Es wird ein verteiltes gerichtetes Diffusionsverhalten und eine Selbstorganisation verwendet, die von einem Bildmerkmalsextraktionsansatz abgeleitet sind.
- Es handelt sich hierbei um einem selbstadaptiven Kantendetektor.
- Eine einzelne sporadische Sensoraktivität, die nicht mit der umgebenden Nachbarschaft korreliert ist, sollte von einer erweiterten korrelierten Region unterschieden werden, die das zu erfassende Merkmal darstellt.

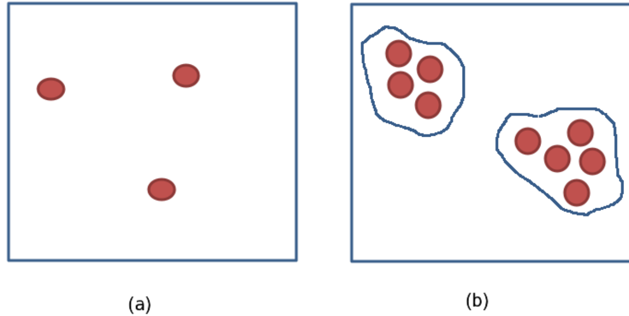


Abb. 9. (a) Nichtkorellierte Sensorstimuli (b) Korrelierte Sensorstimulibereiche

Verteilte Mustererkennung in Sensornetzwerken

Der Algorithmus

Die Merkmals-Erkennung wird vom mobilen Explorationsagenten durchgeführt, der zwei verschiedene Verhaltensweisen unterstützt: Diffusion und Reproduktion.

- Das Diffusionsverhalten wird verwendet, um sich in einem ausgedehnten Bereich zu bewegen, der hauptsächlich durch die Lebensdauer des Agenten begrenzt ist, und um das Merkmal zu detektieren;
    - hier den Bereich mit erhöhter mechanischer Verzerrung (genauer gesagt die Kante eines solchen Bereichs).
- Die Erkennung des Merkmals wird durch das Reproduktionsverhalten verstärkt, das den Agenten veranlasst, am aktuellen Knoten zu bleiben, eine Merkmalsmarkierung zu setzen und mehr Explorationsagenten in der Nachbarschaft auszusenden.

Verteilte Mustererkennung in Sensornetzwerken

- Der lokale Stimulus *H(i,j)* für einen Explorationsagenten, der sich an einem spezifischen Knoten mit der Koordinate (*i*,*j*) befindet, ist gegeben durch:

$$H(i,j) = \sum_{s=-R}^{R} \sum_{t=-R}^{R} \{|S(i+s, j+t) - S(i,j)| \leqslant \delta\}$$

$$S: \text{ Sensor Signal Strength}$$

$$R: \text{ Square Region around (i,j)}$$

- Die Berechnung von *H* an der aktuellen Position (*i*,*j*) des Agenten erfordert die Sensorwerte innerhalb des quadratischen Bereichs (der Region von Interesse ROI) *R* um diesen Ort herum.
- Wenn ein Sensorwert *S(i+s,j+t)* mit *i,j* ∈ {-*R*, .., +*R*} ähnlich dem Wert *S* an der aktuellen Position ist (Differenz ist kleiner als der Parameter *d*), wird *H* um eins erhöht.

Verteilte Mustererkennung in Sensornetzwerken

- Wenn der H-Wert innerhalb eines parametrierten Intervalls $D = [e_0, e_1]$ liegt, hat der Explorationsagent das Feature erkannt und verbleibt am aktuellen Knoten, um neue Explorationsagenten zu reproduzieren, die an die Umgebung gesendet werden.
- Wenn H außerhalb dieses Intervalls liegt, wird der Agent zu einem anderen Knoten wechseln und die Exploration (Diffusion) neu starten.
- Die Berechnung von H erfolgt durch eine verteilte Berechnung von Teilsummenausdrücken durch Aussenden von Kind-Agenten an die Nachbarschaft, die selbst mehr Agenten aussenden können, bis die Grenze der Region R erreicht ist.
- Jeder untergeordnete Agent kehrt zu seinem Ursprungsknoten zurück und übergibt den Teilsummenbegriff an seinen übergeordneten Agenten.

Verteilte Mustererkennung in Sensornetzwerken

- Da ein Knoten in der Region R von mehr als einem Kind-Agenten besucht werden kann, setzt der erste Agent, der einen Knoten erreicht, eine Markierung `MARK`.
  - Wenn ein anderer Agent diese Markierung findet, kehrt er sofort zum übergeordneten Agenten zurück.
- Dieser Mehrwegebesuch hat den Vorteil einer erhöhten Wahrscheinlichkeit, Knoten mit fehlenden (nicht arbeitenden) Kommunikationsverbindungen zu erreichen.
- Ein Eventagent, der von einem Sensingagenten erzeugt wird, liefert schließlich Sensorwerte an Rechenknoten, was hier nicht berücksichtigt wird.

Das Agentenverhalten

- Definition von Typen, Körpervariablen, und Hauptklasse Explorer mit den Aktivitäten *init* und *percept*

```
1   κ: { SENSORVALUE,FEATURE, H, MARK }    set of key symbols
2   ξ: { TIMEOUT, WAKEUP }                 set of signals
3   δ: { NORTH,SOUTH, WEST, EAST, ORIGIN }  set of directions
4   ε1 =3; ε2 = 6; MAXLIVE = 1;           some constant parameters
5
6   Ψ Explorer: (dir,radius) → {
7     * Body Variables *
8     Σ: { dx, dy, live, h, s0, backdir, group }   global persistent variables
9     σ: { enoughinput, again, die, back, s, v }   local  temporary  variables
10
11    Activities
12    α init: {
13      dx ← 0; dy ← 0; h ← 0; die ← false; group ← ℜ{0..10000};
14      if dir ≠ ORIGIN then
15        ⇔dir; backdir ← ϖ(dir)
16      else
17        live ← MAXLIVE; backdir ← ORIGIN
18      ∇⁺(H,$self,0);
19      ∇%(SENSORVALUE,s0?)
20    }
21    α percept: {
22      enoughinput ← 0;
23      ∀{nextdir∈δ | nextdir ≠ backdir ∧ ?Λ(nextdir)} do
24        enoughinput++;
25        Θ→Explorer.child(nextdir,radius)
26      τ⁺(ATMO,TIMEOUT)
27    }
```

- Aktivitäten *reproduce* und *diffuse*

```
28    α reproduce: {
29      live--;
30      ∇×(H,$self,?);
31      if ?∇(FEATURE,?) then ∇⁻(FEATURE,n?) else n ← 0;
32      ∇⁺(FEATURE,n+1);
33      if live > 0 then
34       π*(reproduce → init)
35       ∀{nextdir∈δ | nextdir ≠ backdir ∧ ?Λ(nextdir)} do
36         Θ→(nextdir,radius)
37       π*(reproduce → exit)
38    }
39    α diffuse: {
40      live--;
41      ∇×(H,$self,?);
42      if live > 0 then
43       dir ← ℜ{nextdir∈δ | nextdir ≠ backdir ∧ ?Λ(nextdir)}
44      else
45       die ← true
46    }
47    α exit: { ⊗($self) }
48
49    inbound: (nextdir) → {
50      case nextdir of
51      | NORTH →  dy > -radius
52      | SOUTH →  dy < radius
53      | WEST  →  dx > -radius
54      | EAST  →  dx < radius
55    }
56
```

- Signalhandler und Hauptübergangsnetzwerk

```
57    Signal handler
58    ξ TIMEOUT: {
59      enoughinput ← 0
60    }
61    ξ WAKEUP: {
62      enoughinput--;
63      if ?∇(H,$self,?) then ∇⁻(H,$self,h?);
64      if enoughinput < 1 then τ⁻(TIMEOUT);
65    }
66
67    Main Transitions
68    Π: {
69      entry → init
70      init → percept
71      percept → reproduce | (h ≥ ε1 ∧ h ≤ ε2) ∧ (enoughinput < 1)
72      percept → diffuse   | (h < ε1 ∨ h > ε2) ∧ (enoughinput < 1)
73      reproduce → exit
74      diffuse → init | die = false
75      diffuse → exit | die = true
76    }
```

Verteilte Mustererkennung in Sensornetzwerken { - }

- Subklasse Nachbarschaftsexplorer

```
77    Explorer child subclass
78    φ child: {
79      α exit      imported from root class
80      ξ TIMEOUT
81      ξ WAKEUP
82      α percept_neighbour {
83        if not ?∇(MARK,group) then
84          back ← false; enoughinput ← 0; ∇τ(MTMO,MARK,group);  ∇%(SENSORVALUE,s?);
85          h ← (if |s-s0| ≤ DELTA then 1 else 0);
86          ∇+(H,$self,h);
87          π*(percept_neighbour → move)
88          ∀{nextdir∈δ | nextdir ≠ backdir ∧ ?Λ(nextdir) ∧ inbound(nextdir)} do
89            Θ→(nextdir,radius)
90          π*(percept_neighbour → goback | enoughinput < 1)
91          τ+(ATMO,TIMEOUT)
92      }
93      α move: {
94        backdir ← ϖ(dir); (dx,dy) ← (dx,dy) + ∂(dir);
95        ⇔dir;
96      }
97      α goback: {
98        if ?∇(H,$self,?) then ∇-(MARK,$self,h?) else h ← 0;
99        ⇔backdir;
100     }
101     α deliver: {
102       ∇-(H,$parent,v?); ∇+(H,$parent,v+h);
103       ξWAKEUP ⇒ $parent;
104     }
105     π: {
106       entry → move
107       move → percept_neighbour
108       deliver → exit
109       goback → deliver
110     }
```

Verteilte Mustererkennung und Sensordistribution

- Sensordistribution in verteilten Sensornetzwerken kann strombasiert oder ereignisbasiert erfolgen.

Strombasierte Sensordistribution
Es gibt einen zentralen oder mehrere dezentrale Netzwerkknoten die in periodischen Intervallen die Sensorwerte aller Knoten abfragen - unabhängig davon ob diese sich zu der letzten Anfrage geändert haben

Ereignisbasierte Sensordistribution
Die Sensorknoten liefern Sensordaten zu einem zentralen oder mehreren dezentralen Knoten wenn sich (1) Der Sensorwert geändert hat und (2) es sich um ein ausgedehntes korreliertes Ereignis handelt → Verteilte Mustererkennung

- Sensorwerte können dann per Randomwalk oder gerichteter Diffusion verteilt werden.

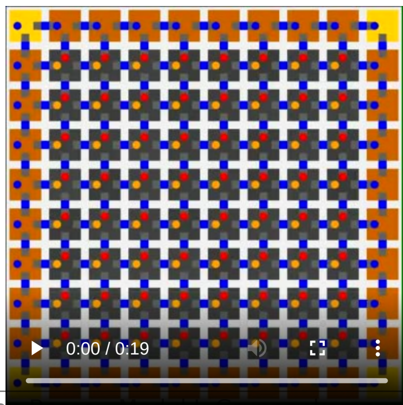
Gerichtete Diffusion

Von einem Quellknoten werden Duplikate von Verteilungsagenten in alle Richtungen ausgesendet, und an den Rändern des Netzwerks zu den Senkeknoten (Rechnern) geleitet.

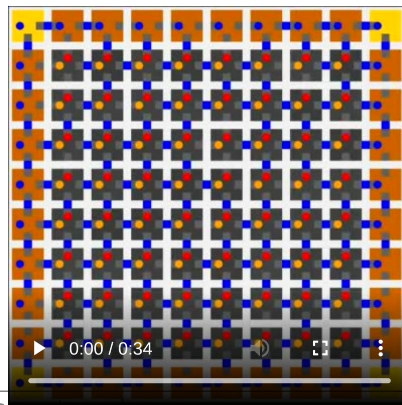
Verteilte Mustererkennung und Sensordistribution

Beispiel in Aktion: Positive Ereigniserkennung

Cluster mit 100% intakten Netzwerkverbindungen



Cluster mit 60% intakten Netzwerkverbindungen



Verteilte Mustererkennung und Sensordistribution

Beispiel in Aktion: Gemischte Situation

Cluster und Störungen