# Multiagentensysteme

*Technologien, Architekturen, Plattformen*

Prof. Dr. Stefan Bosse

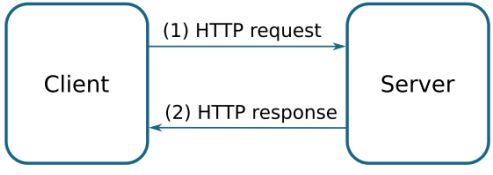
Universität Koblenz - FB Informatik - Praktische Informatik

# Verteiltes Rechnen mit JAM Agenten (ABC)

Zusammenfassung der JAM Plattform und Anwendung im Mobilen Crowdsensing

# Motivation

- Crowdsensing ist verteilte Datenverarbeitung
- **Verteilte Datenverarbeitung** bedeutet **verteilte Kommunikation zwischen Prozessen**
- Traditionelle verteilte Netzwerke wie das Internet der Dinge (IoT) müssen mit einer Vielzahl von Kommunikationsprotokollen und Netzwerkstrukturen umgehen können → **Stark heterogene Systeme**
- **Datenrepräsentation** ist eine weitere Hürde in solchen verteilten Systemen
- Häufig HTTP basierte Klienten-Server Kommunikation (zentrale Serverinstanz)



# Datengetriebene Kommunikation

[Bellifemine, Developing multi-agent systems with a FIPA-compliant agent framework]

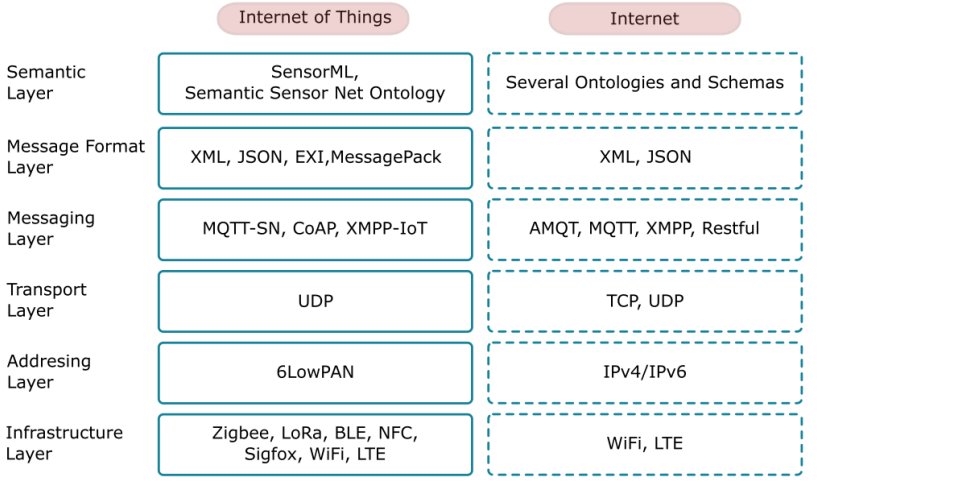



Abb. 1. Große Diversität an IoT Protokollen und Datenrepräsentation für die Kommunikation → nicht einheitlich

# Agenten

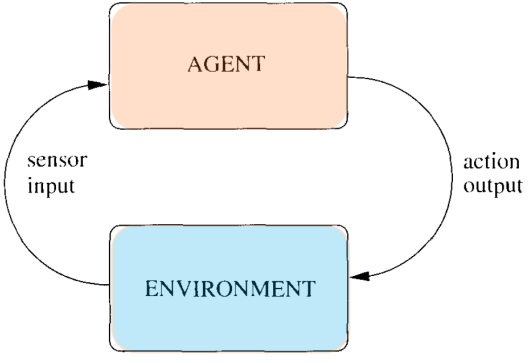
Agenten besitzen eine Vielzahl von Fähigkeiten, die sie von klassischen Programmen unterscheiden - obwohl Agenten auch Programme sein können!

- Fähigkeit zu eigenständiger Aktivität (**Nicht Nutzeraktiviert**)
- Autonomes, "selbstbestimmtes" Verhalten (**Nicht durch zentrale Instanz gesteuert**)
- Fähigkeit zum selbstständigen Schlussfolgern (**Umgang mit unsicheren Wissen**)
- Flexibles und rationales Verhalten (**Adaptivität an veränderliche Weltbedingungen**)
- Fähigkeit zu Kommunikation und Interaktion (**Synchronisation**)
- Kooperatives oder konkurrierendes Verhalten (**Lösung von Wettbewerbskonflikten**)
- Fähigkeit zur ziel- und aufgabenorientierten Koordination (**Kooperation**)

# Agentenmodell

- Ein Agent in seiner Umgebung nimmt sensorische Eingaben aus der Umgebung auf und produziert Ausgabe-Aktionen, die ihn *und* die Umgebung beeinflussen.
- Die Interaktion ist gewöhnlich fortlaufend und nicht-terminierend!

## Einfachstes Agentenmodell




## Beispiel

```
Umgebung: Raum in Gebäude
Sensor: Temperatur T
Aktor:  Heizung

Verhalten:
(1) Temperatur zu niedrig  ⟼ Heizung an
(2) Temperatur angenehem   ⟼ Heizung aus
```

# Agenten und Weltumgebung

## Reaktive und Zustandsbasierte Agenten

- Reaktive und zustandsbasierte Agenten führen einen Zyklus durch:
  - Perzeption ⇒ Verabeitung ⇒ Zustandsänderung ⇒ Entscheidung ⇒ Aktion

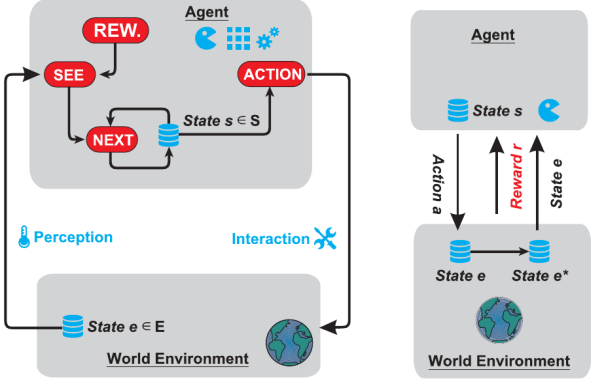



Abb. 2. (Links) Interaktion eines Agenten mit der Umwelt: Perzeption, Aktion (Rechts) Lernender Agent (Reinforcement Agent)

# ATG Modell

## Zustandsbasierter Agent

- Besteht aus: (1) Körpervariablen (2) Kontrollzustand und Verhalten

## Aktivität und Zustand

- Das Verhalten eines aktivitätsbasierten Agenten ist durch einen Agentenzustand gekennzeichnet, der durch Aktivitäten verändert wird.
- Aktivitäten verarbeiten Wahrnehmungen, planen Aktionen und führen Aktionen aus, die den Steuerungs- und Datenzustand des Agenten ändern.
- Aktivitäten und Übergänge zwischen Aktivitäten werden durch einen Aktivitätsübergangsgraphen (Activity Transition Graph, ATG) dargestellt.
- Die Übergänge starten Aktivitäten in der Regel abhängig von der Auswertung von Agentendaten (Körpervariablen), die den Datenzustand des Agenten repräsentieren.

# ATG Modell

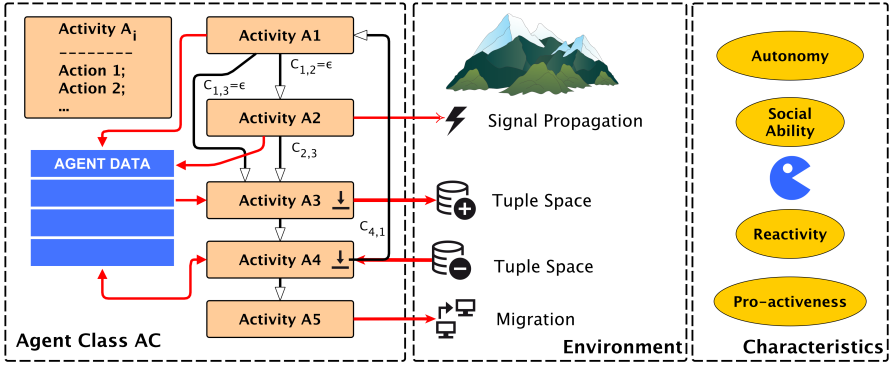



Abb. 3. (Links) Agentenverhalten, das von einem Aktivitätsübergangsgraphen vorgegeben ist, und die Interaktion mit der Umgebung, die durch Aktionen erfolgt, die in Aktivitäten ausgeführt werden (Rechts) Agentenmerkmale

# ATG Modell

- Ein Aktivitätsübergangsgraph, der mit Agentenklassen assoziiert ist, besteht aus einer Menge von Aktivitäten $\mathbb{A} = \{A_1, A_2, ..\}$ und einer Menge von Übergängen $\mathbb{T} = \{T_1 (C_1), T_2 (C_2), ..\}$, die die Kanten des gerichteten Graphen darstellen.

- Die Ausführung einer Aktivität, die selbst aus einer Folge von Aktionen und Berechnungen besteht, ist mit dem Erreichen eines Unterzieles oder das Erfüllen einer Voraussetzung verknüpft, um ein bestimmtes Ziel zu erreichen, z. B. Sensordatenverarbeitung und Verteilung von Informationen.

- Normalerweise werden Agenten verwendet, um komplexe Aufgaben zu zerlegen basierend auf der Zerlegung durch MAS.

- Agenten können ihr Verhalten basierend auf Lern- und Umgebungsänderungen oder durch Ausführen einer bestimmten Unteraufgabe mit nur einer *Untermenge* des ursprünglichen Agentenverhaltens ändern → **Dynamische ATG**.

# ATG Modell

- Das ATG-Verhaltensmodell ist eng mit der Interaktion von Agenten mit deren Umgebung verbunden, hier hauptsächlich durch
  - Den Austausch von Daten unter Verwendung einer Tupelraum-Datenbank;
  - Durch Migration; und durch
  - Die Weitergabe von Nachrichten zwischen Agenten mittels Signalen.
  - Replikation und Instantiierung von Agenten

# DATG Modell

- Ein ATG beschreibt das vollständige Agentenverhalten.
- Jedes Unterdiagramm und jeder Teil des ATG kann einem Unterklasseverhalten eines Agenten zugeordnet werden.
- Daher führt das Modifizieren der Menge von Aktivitäten $\mathbb{A}$ und Übergängen $\mathbb{T}$ des ursprünglichen ATG zu mehreren Unter- und Oberverhaltensweisen, die Algorithmen implementieren, um verschiedene unterschiedliche Ziele zu erfüllen.
- Die Rekonfiguration der Aktivitäten führt zu einer Menge von Aktivitätsmengen $\mathbb{A}^* = \{\mathbb{A}_i \subset \mathbb{A}, \mathbb{A}_j \subset \mathbb{A}, \mathbb{A}_k \supset \mathbb{A}, ..\}$, die von der ursprünglichen Menge $\mathbb{A}$ abgeleitet sind, und die Modifikation oder Rekonfiguration von Übergängen $\mathbb{T}^* = \{\mathbb{T}_1 \subset \mathbb{T}, \mathbb{T}_2 \subset \mathbb{T}, ..\}$ ermöglicht die dynamische ATG-Zusammensetzung (Komposition) und Agentenunterklassifizierung zur Laufzeit,

# DATG Modell

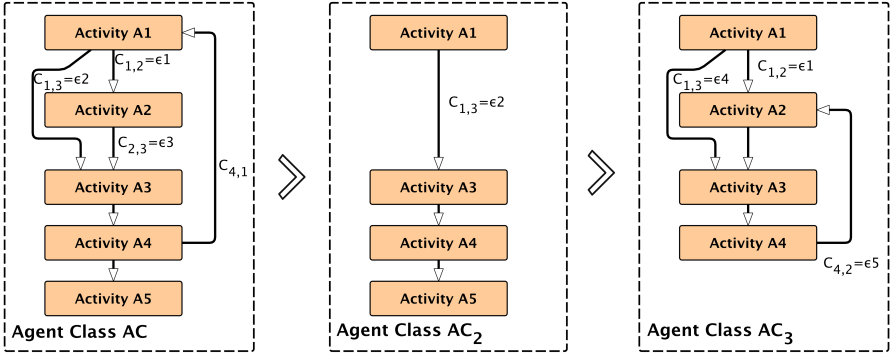



Abb. 4. Dynamischer ATG: Transformation und Komposition

# Beispiel eines Agenten

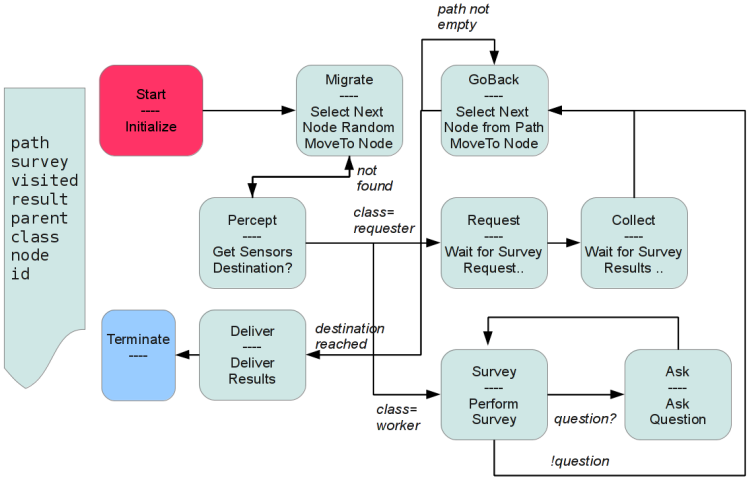



Abb. 5. ATG eines einfachen Umfrageagenten mit Unterklassen (*master*, *requester*, *worker*)

# Agentenklassen

Eine Agentenklasse beschreibt ein bestimmtes Verhalten eines Agenten mittels eines ATG und einer Menge von Körpervariablen.

- Von einer Klasse können zur Laufzeit Agenten instantiiert (erzeugt) werden.

**Verhalten**

Eine bestimmte Agentenklasse $AC_{\mathrm{i}}$ bezieht sich auf das zuvor eingeführte ATG Modell, dass das Laufzeitverhalten und die von Agenten ausgeführten Aktion definiert.

**Wahrnehmung**

Ein Agent interagiert mit seiner Umgebung, indem er eine Datenübertragung unter Verwendung eines einheitlichen Tupelraums mit einer koordinierten datenbankähnlichen Schnittstelle durchführt.
Daten aus der Umgebung beeinflussen das folgende Verhalten und die Aktion eines Agenten. Daten, die an die Umgebung (z.B. die Datenbank) weitergegeben werden, beeinflussen das Verhalten anderer Agenten.

# Agentenklassen

## Speicher

Zustandsbasierte Agenten führen Berechnungen durch, indem sie Daten ändern. Da Agenten als autonome Datenverarbeitungseinheiten betrachtet werden können, werden sie hauptsächlich private Daten modifizieren, und ein Berechnungsergebnis, dass diese Daten verwendet, in die Umgebung übertragen.
Daher enthält jeder Agent und jede Agentenklasse eine Menge von Körpervariablen $\mathbb{V} = \{v_1 : typ_1, v_2 : typ_2, ..\}$, die durch Aktionen in Aktivitäten modifiziert und in Aktivitäten und Übergangsausdrücken gelesen werden.

# Agentenklassen

## Parameter

Agenten können zur Laufzeit von einer bestimmten Agentenklasse instantiiert werden, die Agenten mit gleichen anfänglichen Steuerungs- und Datenzuständen erstellt.
Um einzelne Agenten zu unterscheiden (Individuen zu erzeugen), wird eine externe sichtbare Parametermenge $\mathbb{P} = \{p_1 : typ_1, p_2 : typ_2, ..\}$ hinzugefügt, die die Erstellung verschiedener Agenten bezüglich des Datenzustands ermöglicht. Innerhalb einer Agentenklasse werden Parameter wie Variablen behandelt.

# Agentenklassen

Eine Agentenklasse $AC_\mathrm{i}$ ist daher zunächst definiert durch das folgende Mengentupel:

$$AC_i = \langle \mathbb{A}_i, \mathbb{T}_i, \mathbb{V}_i, \mathbb{P}_i \rangle$$
$$\mathbb{A} = \{a_1, a_2, .., a_n\}$$
$$\mathbb{T} = \{t_{ij} = t_{ij}(a_i, a_j, cond) | a_i \xrightarrow{cond} a_j; i, j \in \{1, 2, .., n\}\}$$
$$a_i = \{i_1, i_2, .. | i_u \in ST\}$$
$$\mathbb{V} = \{v_1, v_2, .., v_m\}$$
$$\mathbb{P} = \{p_1, p_2, .., p_i\}$$

# Agentenklassen

## Multiagentensysteme

Es gibt ein Multiagentensystem (MAS), das aus einer Reihe einzelner Agenten besteht ($ag_1$, $ag_2$, ..). Es gibt verschiedene Verhaltensweisen für Agenten, die als Klassen **AC** = $\{AC_1, AC_2, ..\}$ bezeichnet werden. Ein Agent gehört zu einer dieser Klassen.

Jede Agentenklasse wird dann durch das erweiterte Tupel $AC$ = <$\mathbb{A}$, $\mathbb{T}$, $\mathbb{F}$, $\mathbb{S}$, $\mathbb{H}$, $\mathbb{V}$, $\mathbb{P}$> angegeben.

- $\mathbb{A}$ ist der Satz von Aktivitäten (Graphenknoten), $\mathbb{T}$ ist der Satz von Übergängen, die Aktivitäten (Beziehungen, Graphenkanten) verbinden,
- $\mathbb{F}$ ist der Satz von Rechenfunktionen,
- $\mathbb{S}$ ist der Satz von Signalen, $\mathbb{H}$ ist der Satz von Signalhandlern,
- $\mathbb{V}$ ist die Menge der Körpervariablen und $\mathbb{P}$ die Menge der Parameter, die von der Agentenklasse verwendet werden.

Def. 1. Agentenklasse

# Agentenklassen

In einer spezifischen Situation ist ein Agent $ag_i$ an einen Netzwerkknoten $N_{m,n,o,..}$ (z.B. Mikrochip, Computer, virtueller Simulationsknoten) an einem eindeutigen räumlichen Ort (m, n, o, ..) gebunden und wird dort verarbeitet.

Es gibt eine Menge verschiedener Knoten $\mathbf{N} = \{N_1, N_2, ..\}$, die z.B. in einem maschenartigen Netzwerk mit einer Nachbarverbindung (z.B. vier in einem zweidimensionalen Gitter) angeordnet sind. Die Knotenverbindung kann dynamisch sein und sich im Laufe der Zeit ändern. Die Knotennachbarn sind unterscheidbar.

Jeder Knoten ist in der Lage, eine Anzahl von Agenten $n_i(AC_i)$ zu verarbeiten, die zu einer Agentenverhaltensklasse $AC_i$ gehören, und mindestens eine Teilmenge von $\mathbf{AC'} \subset \mathbf{AC}$ zu unterstützen.

Ein Agent (oder zumindest sein Zustand) kann zu einem Nachbarknoten migrieren wo er weiter ausgeführt wird.

Def. 2. Agentenplattformen

# Agenteninterkation

Multiagentensystem interagieren miteinander, z.B. durch

1. Nachrichten
2. Signale
3. Semantischer (oder assoziativer) Geteilter Speicher, und synchronisiert ⇒ Tupelräume

Die Interaktion beeinflusst das Verhaltebn und somit den Aktivitätenfluss der Agente (Programmfluss)

# 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 Agenten** 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

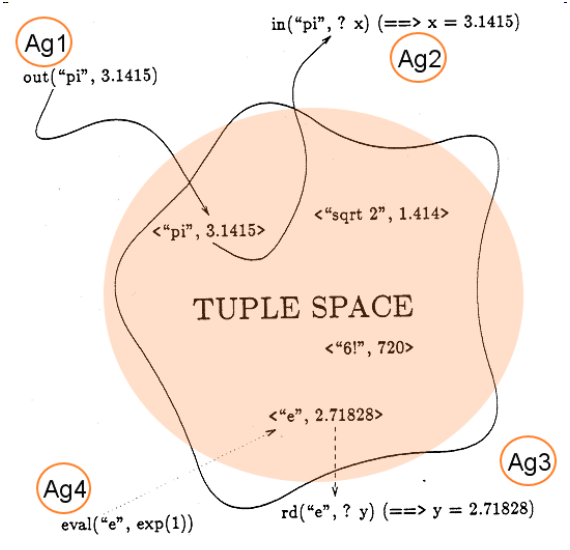



Abb. 6. Ein Schnappschuss eines Tupelraumes mit Tupeln und Tupeloperationen

# Tupelräume

- Kommunikation von Agenten über Tupelräume ist eine **Koordinationssprache**.

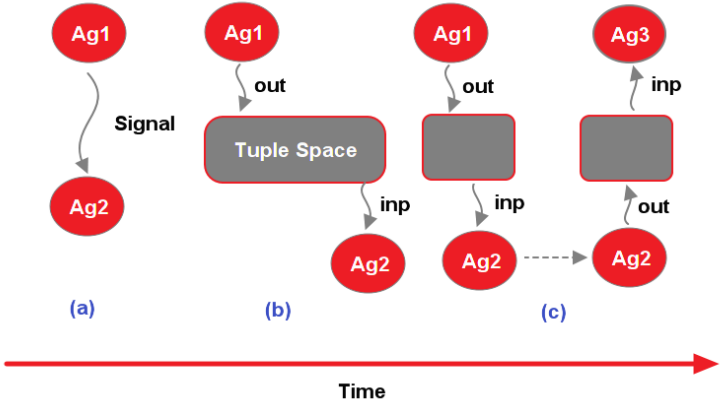



Abb. 7. Direkter Nachrichtenaustausch (a), z.B. durch Signale, im Vergleich zu generativer Kommunikation (b) und virtuelle verteilte Räume (c) durch mobile Prozesse (Agenten)

# 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 - 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**

```
function (t:tuple)
```

- 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.

# Tupelräume - Operationale Semantik

**inp**

```
function (p:pattern, callback: function
(tuple|tuple[]|null),all?:boolean)
```

- 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.

**rd**

```
function (p:pattern, cb:function (tuple|tuple[]|null),all?:boolean)
```

- 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.

# Tupelräume - Operationale Semantik

**Beispiele**

```
out(["Sensor",1,100]);
out(["Sensor",2,121]);
inp(["Sensor",1,_], function (t) { if (t) v=t[2]; });
inp(["Sensor",_,_], function (t) { if (t) n=t[1],v=t[2] });`
rd(["Sensor",_,_], function (t) { if (t) n=t[1],v=t[2] });`
```

# Tupelräume - Operationale Semantik

**try_inp, try_rd**

```
function (0,p:pattern,callback?:function,all?:boolean) ⟼
tuple|tuple[]|null
```

- Nichtblockierende Version von *inp*/*rd*. Wird kein passendes Tupel gefunden wird die Operation ergebnislos terminiert.

**try_inp, try_rd**

```
function (tmo:number,p:pattern,callback:function,all?:boolean)
```

- Teilblockierende Version von *try_inp*/*try_rd*, Wird innerhalb einer Zeit von *tmo* kein passendes Tupel gefunden wird die Operation ergebnislos abgebrochen.

- 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**

```
function (pattern) ⟼ boolean
```

- Nicht blockierender Test eines Tupels.

**ts (testandset)**

```
function (pattern,function (tuple) ⟼ tuple)
```

- 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.

**rm**

```
function (p:pattern,all?:boolean)
```

- Entfernung eines oder aller passenden Tupel.

# Tupelräume - Operationale Semantik

## 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**

```
function (tmo:number,t:tuple)
```

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

# Tupelräume - Operationale Semantik

**alt**

```
function (pattern [], function (tuple|tuple[]|null),all?:boolean)
```

- Gleichzeitige Mehrfachabfrage (mit *inp* Semantik) von einer Menge von Mustern. Das erste Tupel welches einem der Muster entspricht wird an die Rückruffunktion übergeben. Es gibt auch eine nicht- oder partiell blockierende Variante *try_alt*.

# Tupelräume - Produzenten und Konsumenten

## Produzent

```
this.act = {
 percept : function () {
   out(["SENSORNODE"]);
   mark(1000,["SENSOR","CLOCK",time()])
   mark(1000,["SENSOR","GPS",
              {lati:x,long:y}])
 }
}
```

## Konsument

```
this.act = {
 process : function () {
  if (test(["SENSORNODE"])
   inp(["SENSOR","CLOCK",_], function (t) {
    if (t) log('It is time '+t[2]);
   })
 }
}
```

# Agenteninstantiierung

Folgende Möglichkeiten gibt es:

1. Ein Agent wird von der Plattform erzeugt.
2. Ein Agent wird als Kopie von einem Agenten abgespalten (Eltern-Kind Gruppen).
3. Ein Agent erzeugt einen neuen Agenten (seiner oder einer anderen Klasse).

# Agenteninstantiierung

```
// 1. Plattform
function ac() { .. }
var ag = create(ac,{msg:'with pace'},2);
// 2. Agent forking
function ac() {
  act : function () {
    this.child = fork({
      v1:ε1,
      v2:ε2,
      ...
    })
  }
}
// 3. Agent-in-Agent creation
function ac () {
  act : function () {
    this.agent = create('agclassname',{
      v1:ε1,
      v2:ε2,
      ...
    },level)
  }
}
```

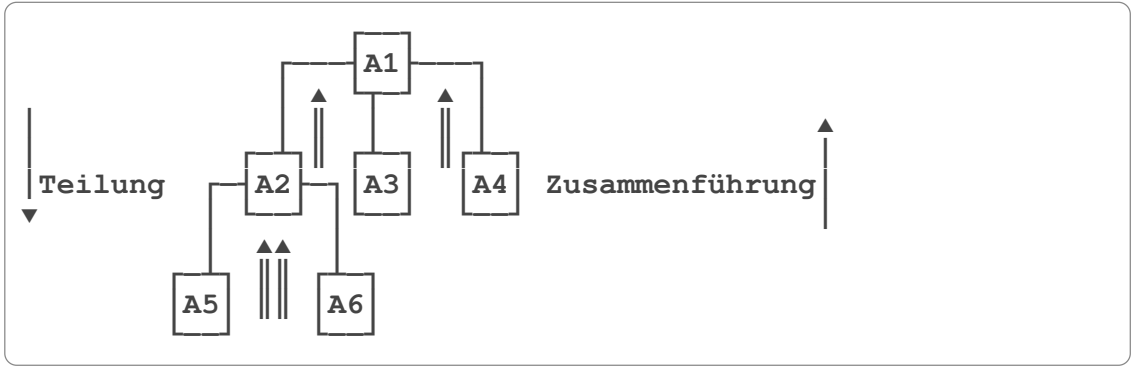
# Multiagentensysteme: Dynamische Erzeugung und Kommunikation

- Ein einzige Agent kann schon alles nötige Verhalten aufweisen um Probleme reaktiv, deliberativ, oder proaktiv zu lösen.
- Das Grundprinzip ist hier aber Divide&Conquer: Eine komplexe Aufgabe wird in kleinere Teilaufgaben zerlegt.
  - Dazu werden Gruppen von Agenten benötigt die sich situationsabhängig erzeugen und wieder vereinen.
  - Diese Agenten können entweder über Tupelräume (anonym) oder gerichtet mit Signalen kommunizieren

1. Initiale Menge von Agenten wird erzeugt
2. Abhängig von Perzeption und Historie werden neue Auftragsagenten dynamisch erzeugt
3. Die neuen Agenten übernehmen Teilaufgaben, die sie ggfs. an weitere erzeugte Agenten weitergeben
4. Ergebnisse der Auftragsagenten werden zusammengetragen und wieder beim Ursprungsagenten vereint.

Def. 3. Grundprinzip MAS

# Multiagentensysteme: Dynamische Erzeugung und Kommunikation




Alg. 1. Teilung und Zusammenführung von Agenten bzw. deren Daten

# Kommunikationssignale

- Agenten können direkt über Signale miteinander kommunizieren
- Es muss nur die ID des Agenten bekannt an den ein Signal gesendet werden soll
- Zunächst müssen Agenten auf der gleichen Plattform ausgeführt werden um Signale auszutauschen

Aber: Wenn Agenten A und B zu einem Zeitpunkt $t_0$ auf der gleichen Plattform ($N_1$) ausgeführt wurden, und dann die Agenten auf verschiedenen Plattformen ($N_1, N_2$) ausgeführt werden können diese dennoch Signale austauschen ($t_0 < t < t_1$). Es werden in den Plattformen Spuren der Agenten vermerkt, aber nur für eine bestimmte Zeit (hier bis $t_1$).

# Kommunikationssignale

- **Signale** sind einfache Nachrichten die
  - An einen bestimmten Prozess oder Agenten gerichtet sein kann → **Unicast**;
  - An eine bestimmte Gruppe von Prozessen oder Agenten → **Multicast**;
  - Oder an unbestimmte Gruppe von Prozessen oder Agenten in der Umgebung → **Broadcast**
- Ein Signal besteht aus einem Signaltyp (Nummer, Zeichenkette usw.) und einem (optionalen) Datenteil (Signalargument)
- Ein Signal kann gesendet und empfangen werden.
- Häufig wird auf den Empfang eines Signals nicht aktiv gewartet sondern mittels **Signalhandlern**, die eingehende Signale *asynchron* verarbeiten.
- Ein Agent kann verschiedene Signale aus einer Menge $\boldsymbol{S}=[Sig_1,Sig_2,..\}$ verarbeiten
- Für jedes Signal muss ein Signalhandler installiert werden!

# Kommunikationssignale

## Verarbeitung von Signalen

### Prinzip

```
signal SIGNAL1,SIGNAl2;
Ag1:
  on(SIGNAL1, function (arg,from) {
   do process signal SIGNAL1 from sender });
  on(SIGNAL2, function (arg,from) {
   do process signal SIGNAL2 from sender });
Ag2:
  send(Ag1,SIGNAL1,ε);
```

### AgentJS

```
signal SIGNAL1,SIGNAl2;
Ag1:
  on : {
    SIGNAL1 : function (arg,from) {
    }
  }
Ag2:
  send(Ag1,"SIGNAL1",ε);
```

# Kommunikationssignale

- Signale sind gekennzeichnet durch das Tupel
  *<Absender,Empfänger,Name,Wert>*
- Nachteil gegenüber Tupeln: Der Agent kann die eingehenden Nachrichten nicht filtern bezüglich
  - Inhalt
  - Relevanz und Interesse
  - Absender

# Kommunikationssignale

# Kommunikationssignale

- Anders als bei Tupeln muss der Empfänger (Agent) dem Absender (Agent) bekannt sein (Agentenreferenz)
  - Eltern-Kind Beziehungen werden daher häufig für Unicast Signale verwendet
  - Gruppenbeziehungen können durch Agentenklassen und räumliche Bereiche entstehen

## Routing

- Signale adressieren mobile Agenten die ihren Standort, d.h. die Plattform, wechseln können
- Der Vermittlung (Routing) der Signalnachrichten kommt daher besondere Bedeutung zu.
- Eine Möglichkeit eine Signalnachricht zwischen zwei Agenten *A* und *B* zu vermitteln ist die Vermittlung entlang des Pfades von *A* und *B* (*Spuren*)
  - Wenn die Spuren von *A* und *B* sich irgendwo kreuzen (Kreuzungspunkte sind die Plattformen) so kann die Nachricht über den Kreuzungspunkt zugestellt werden

# Kommunikationssignale

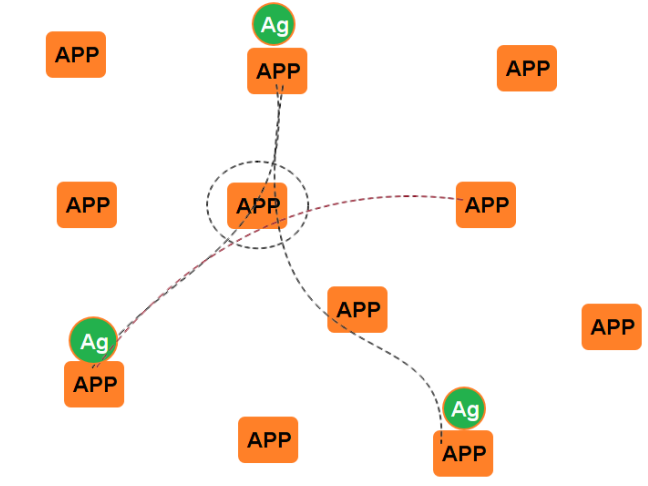



Abb. 8. Mobile Agenten in einem Netzwerk aus Plattformen (APP) und ihrer Spuren; Routing von Nachrichten über Kreuzungspunkte von Spuren

# Kommunikationssignale

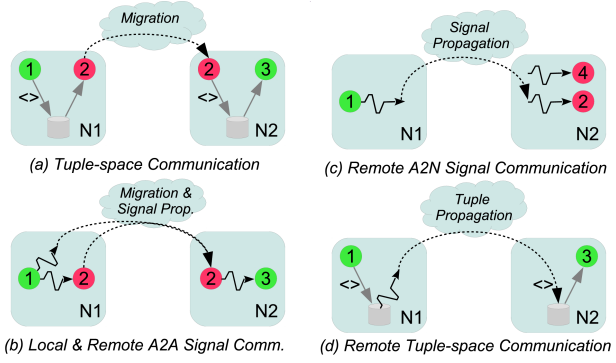



Abb. 9. Agentenkommunikation (a) Lokale Tupleräume (b) Agent-Agent Signale (c) Entfernte Agent-Knoten Signale (d) Entfernte Tupleraumoperationen