Klassische Kommunikations- und Koordinationsmodelle

Martin Buchheit

Deutsches Forschungszentrum für Künstliche Intelligenz
Stuhlsatzenhausweg 3
6600 Saarbrücken
e-mail: buchheit@dfki.uni-sb.de

Zusammenfassung

Bei der Betrachtung von Multi–Agenten–Systemen (MAS) als Teilbereich der Verteilten Künstlichen Intelligenz treten Probleme zutage, die der klassischen Informatik nicht unbekannt sind. Im folgenden werden bekannte klassische Verfahren zur Kommunikation und Koordination in Verteilten Systemen, wie sie in (Verteilten) Betriebssystemen, in (Verteilten) Programmiersprachen und in (Verteilten) Datenbanksystemen zur Anwendung kommen, vorgestellt. Die Verwendbarkeit dieser Methoden im Rahmen der Aufgabenstellung des Projektes AKA–Mod wird allgemein untersucht und am Beispiel eines dort verwendeten Szenarios von Transportunternehmen, des sog. Speditions-Szenarios, veranschaulicht.

Inhaltsverzeichnis

1 Problematik und Motivation **3**

2 Klassische Verfahren zur Kommunikation und Koordination **5**

2.1 Kommunikation 5

2.1.1 Kommunikation mittels gemeinsamer Datenbereiche 5

2.1.2 Kommunikation mittels Nachrichten 5

2.2 Koordination 11

2.2.1 Speicherbasierte Koordinations-Mechanismen 12

2.2.2 Nachrichtenbasierte Koordinations-Mechanismen 14

3 Zeit in verteilten Systemen **21**

4 Verwendbarkeit der Methoden für Multi-Agenten-Systeme **26**

4.1 Zielsetzungen der Problemlösungsmethoden 26

4.2 Integration klassischer Modelle 27

5 Fazit **29**

1 Problematik und Motivation

In der KI-Forschung läßt sich eine verstärkte Tendenz hin zur Beschäftigung mit Verteilungs-Aspekten beobachten. In den letzten Jahren ist ein eigenständiger Teilbereich der KI, die *verteilte künstliche Intelligenz* (VKI) entstanden. Vor allem das Modell autonomer kooperierender Agenten (AKA), das den Multi-Agenten-Systemen (MAS) zugrunde liegt, steht im Mittelpunkt des Interesses (vgl.[24]). Koordination und Kommunikation sind Voraussetzungen für intelligente Kooperation.

Koordination von und Kommunikation zwischen "Agenten" spielen aber auch im Bereich der klassischen Informatik eine wichtige Rolle.

Das wohl am besten untersuchte Beispiel sind die *Prozesse* innerhalb von *Betriebssystemen*. Ein Prozeß bezeichnet ein Programm zusammen mit einem "Pseudo-Prozessor" (vgl. [22]). Die Abstraktion Prozeß wird dabei vorgenommen, um die potentielle Parallelität, die das Vorhandensein mehrerer Ressourcen wie Drucker, Kartenleser, Prozessor etc. bietet, zu nutzen und zu strukturieren. Das Multiplexen des realen Prozessors zwischen den Prozessen ist Aufgabe des Betriebssystems.

Verteilte Betriebssysteme sind zur Organisation von Diensten in Clustern von Rechnern konzipiert (vgl. [23, 25]). Typischerweise lassen sich die Prozesse in verteilten Betriebssystemen in zwei Klassen aufteilen: in solche, die Dienstleistungen erbringen, die sog. *Server* und solche, die Dienstleistungen in Anspruch nehmen, die sog. *Clients*.

Zur Unterstützung der Programmierer bei der Nutzung der Parallelität, die der Zusammenschluß von Rechnern bietet, wurden seit Ende der siebziger Jahre *Sprachen zur Programmierung verteilter Systeme* entwickelt (vgl. [3]). Diese enthalten neben den Sprachkonstrukten sequentieller Programmiersprachen Mechanismen zur Kommunikation, Synchronisation, Ausnutzung der Parallelität und evt. zur Behandlung von Kommunikationsfehlern und Knotenausfällen.

Verteilte Datenbanksysteme sollen für geographisch verteilt gespeicherte Daten die logische Gesamtsicht einer einzigen Datenbank vermitteln (vgl. [4, 9]).

Die Verteilung bietet eine Reihe von Vorteilen. So sind als klassische Vorteile etwa zu nennen (vgl. [19]):

- Leistungsgewinn durch Parallelarbeit
- Fehlertoleranz durch Redundanz
- Erhöhung der Funktionalität und Flexibilität durch inkrementelle Erweiterbarkeit
- Bereitstellung von Rechenleistung an dem Ort, wo sie benötigt wird

Die KI ist mehr interessiert an den engen Beziehungen des Modells zu in der Welt vorkommenden intelligenten Gemeinschaften. Eines der faszinierendsten Phänomene der Natur ist der *Synergie–Effekt*, der durch folgende Aussage charakterisiert wird: "Das Ganze ist mehr als die Summe seiner Teile". So ist z.B. ein DNA-Molekül sicherlich mehr als die Summe seiner Atome, ein Organismus mehr als die Summe seiner Zellen (vgl. [24]).

Doch all diese Vorteile sind nicht umsonst zu haben, sie bringen eine Reihe von Problemen und Aufgaben mit sich. Die Zusammenarbeit muß koordiniert werden. Es müssen Informationen ausgetauscht werden. Die Agenten können und sollen sich gegenseitig beeinflußen. Im Gegensatz zum Ein-Agenten-Modell ändert sich die Welt ohne eigenes Zutun.

Koordination läßt sich in zwei große Klassen einteilen: in *positive* Koordination und *negative* Koordination (vgl. [18]). Die klassische Informatik beschäftigt sich vorwiegend mit Fragen der negativen Koordination, d. h. mit dem Auflösen und Verhindern gegenseitiger Behinderung z. B. bei der Vergabe von Ressourcen. Das Interesse der KI liegt mehr auf Seiten der positiven Koordination, d. h. der Entwicklung von Mechanismen zur gegenseitigen Hilfe zwecks Ausnutzung des Synergie-Effektes.

Koordination und Kommunikation hängen eng zusammen. Indem z. B. Prozesse sich synchronisieren, erfahren sie etwas voneinander – etwa über den aktuellen Kontrollfluß des anderen. Auf der anderen Seite werden Prozesse beim Nachrichtenaustausch zumindest in dem Sinne synchronisiert, daß der Empfänger einer Nachricht sicher sein kann, daß sie vom Sender zuvor abgeschickt wurde.

Koordination kann auch durch eine gemeinsame Zeit-Koordinate realisiert werden. Wir Menschen z.B. synchronisieren den größten Teil unserer Aktivitäten über eine gemeinsame Zeitbasis.

Sowohl Koordination mittels Kommunikation als auch Koordination mittels Zeit beinhalten jedoch schon für sich eine Reihe von (interessanten) Problemen, die hauptsächlich aus dem Fehlen einer konsistenten Sicht auf den globalen Zustand bzw. auf eine globale Zeit resultieren.

Der Rest dieses Artikels ist wie folgt aufgebaut. In Kapitel 2 werden klassische Kommunikations- und Koordinationsverfahren vorgestellt. Abschnitt 2.1 beschäftigt sich mit speicher- bzw. nachrichtenbasierten Kommunikationsmodellen, Abschnitt 2.2 mit speicher- bzw. nachrichtenbasierten Koordinationssmodellen. In Kapitel 3 werden Zeit-Aspekte in verteilten Systemen behandelt. In Kapitel 4 wird die Eignung der Modelle für MAS'e untersucht und am Beispiel des Speditions-Szenarios veranschaulicht.

2 Klassische Verfahren zur Kommunikation und Koordination

2.1 Kommunikation

Kommunikation dient den Agenten eines verteilten Systems dazu, Information über den Zustand der anderen Teile des Systems — sprich der "Welt" — zu erhalten. Diese Information ist in zweifacher Hinsicht notwendig für das System. Zum einen dient sie dazu, die Handlungen der einzelnen aufeinander abzustimmen. Dabei kann die Koordinations-Information explizit sein ("Ich tue gerade dies"), sie ist aber immer auch implizit vorhanden in dem Sinne, daß das Bereitstellen der Information vor dem Aufnehmen kommen muß. So können etwa Nachrichten erst ankommen, nachdem sie weggeschickt wurden. Zum anderen dient sie dem Datenaustausch als Grundlage von Kooperation zwischen den Agenten.

Grundsätzlich lassen sich zwei Methoden der Kommunikation unterscheiden: Kommunikation über gemeinsam zugängliche Datenbereiche (shared variables) und über Nachrichten (message passing).

2.1.1 Kommunikation mittels gemeinsamer Datenbereiche

Bei dieser Art der Kommunikation wird Information auf gemeinsam zugänglichen Speicher geschrieben bzw. von dort gelesen. Bei der Realisierung dieses Konzeptes sind zwei Aspekte wesentlich. Zum einen muß der Zugriff auf gemeinsame Variablen koordiniert werden, um konsistente Information zu erhalten. Es handelt sich also um ein Koordinationsproblem. In Abschnitt 2.2 wird dieses Problem näher untersucht. Zum anderen muß für eine effiziente Implementierung gemeinsamer Speicher zur Verfügung stehen. Ist dies nicht der Fall, muß er mittels Nachrichtenaustausch simuliert werden. Damit gehen jedoch die Hauptvorteile dieses Konzepts, nämlich einfache Realisierung und Effizienz, verloren.

Die Einsatzgebiete dieses Konzeptes reichen von so einfachen Anwendungen wie Realisierung von Puffern in Betriebssystemen bis hin zu so komplexen Bereichen wie Datenbank-Systemen und den Blackboard-Systemen der KI (vgl. [11]).

2.1.2 Kommunikation mittels Nachrichten

Hierbei verkehren Agenten durch das Versenden und Empfangen von Nachrichten. Abbildung 1 beschreibt die Situation. Die Agenten übergeben ihre Nachrichten an ein sog. *Nachrichten-Transport-System* (NTS) bzw. übernehmen sie von dort. Das NTS ist für das Weiterleiten der Nachrichten verantwortlich. Man unterscheidet verschiedene *Kommunika-*

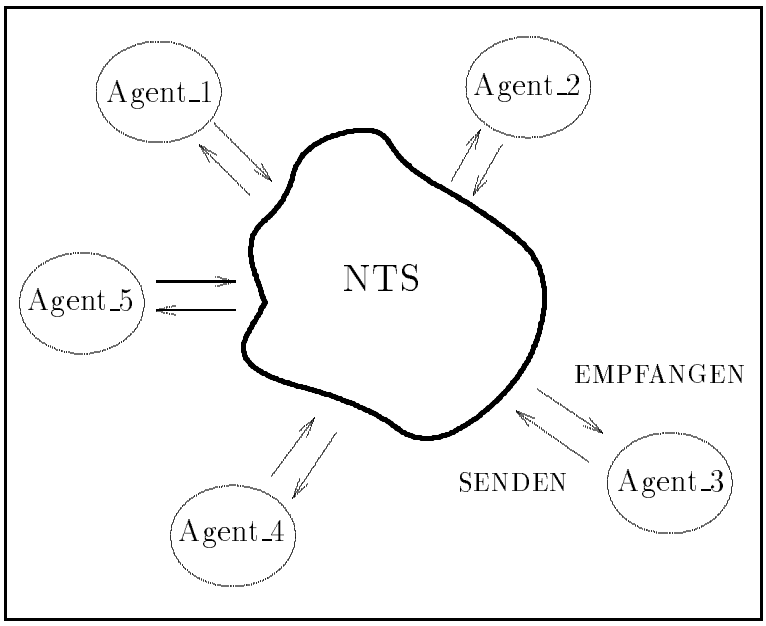

Abbildung 1: Prinzipielle Struktur eines nachrichtenbasierten Kommunikationsmodells

tions-Modelle.

Direkt kommunizierende Agenten

Dieses Modell geht von der direkten Bekanntschaft von Agenten aus. Das Senden von Nachrichten hat die Form

`send`(Empfänger,Botschaft)

und das Empfangen

`receive`(Sender,Botschaft)

`receive`(Botschaft)

für selektives resp. nicht-selektives Empfangen einer Nachricht.

Auf diesem Konzept basieren Sprachen wie CSP und OCCAM (vgl. [3]). Im Kontext von Betriebssystemen stehen hinter Nachrichten meist Anforderungen von Diensten. Oft interessiert weniger die Adresse desjenigen, der den Dienst erbringt, sondern vielmehr der Name des Dienstes. Dies führt zu dem Port-Konzept.

Portbezogene Kommunikation

Ports sind Übergabestelle von Nachrichten bei den Agenten. Agenten besitzen i. allg. mehrere Ports. Damit kann ein Agent verschiedene Typen von Nachrichten, etwa Kontroll- und Basisnachrichten unterscheiden. Außerdem kann dieses Modell so gestaltet sein, daß der Sender nicht den Empfänger, sondern nur den Portnamen kennen muß. Dies führt vor

allem in Betriebssystemen, wo hinter Portnamen Dienste stehen, zu einem hohen Maß an *Strukturtransparenz*.

Teams und Ports

In diesem Modell werden speicherbezogene und portbezogene Kommunikations-Mechanismen kombiniert. Gruppen von Agenten werden zu sog. *Teams* zusammengefaßt. Innerhalb dieser Teams erfolgt Kommunikation mittels gemeinsamer Speicherbereiche (vgl. Abschnitt 2.1). Einem Team wird eine Menge von Ports zugeordnet. Kommunikation zwischen den Teams erfolgt über diese Ports. Dieses Modell unterstützt insbesondere die effiziente Nutzung von Mehrprozessor-Konfigurationen. Ein Team läßt sich dabei in idealer Weise auf einen Mehrprozessor-Rechner abbilden. Diese Modellvorstellung wurde in den Systemen bzw. Sprachen SR, Eden, Argus und LADY realisiert (vgl. [3]).

Mailboxes

Bei diesem Modell verkehren die Agenten über von ihnen unabhängige Objekte, die sog. Mailboxes. Mailboxes können Nachrichten speichern. Sie sind prinzipiell nicht an bestimmte Agenten gebunden. Die Entnahme der Nachrichten aus den Mailboxes geschieht unter der Regie der Agenten. Ein Beispiel für die Realisierung des Modells ist die Sprache CSSA (vgl. [3]).

Neben der Einteilung in Kommunikations-Modelle unterscheidet man Kommunikation bzgl. des *Synchronisations-Grades* (vgl. [22, 6]). Wollen zwei Agenten kommunizieren, so werden sie i. allg. nicht gleichzeitig das `send`- bzw. `receive`-statement erreichen. Wird das `receive` vor dem `send` erreicht, so wartet der Empfänger sinnvollerweise, wird aber `send` vor `receive` aufgerufen, so gibt es zwei sinnvolle Alternativen:

1. Der Sender wartet, bis der Empfänger das `receive`-statement erreicht hat. Man spricht in diesem Fall von *synchroner* Kommunikation.
2. Der Sender schickt die Nachricht ab und führt das nächste statement aus. Die Nachricht wird vom NTS gepuffert bis der Empfänger das `receive`-statement erreicht. Man spricht in diesem Fall von *asynchroner* Kommunikation bzw. *freilaufendem* `send`.

Vorteile der synchronen Kommunikation sind eine implizite *Flußkontrolle*, d. h. das Verhindern des Anwachsens von Nachrichtenpuffern und eine hohe *Zuverlässigkeit*. Das Ende eines Sende-Vorganges kann als Bestätigung für die Ankunft der Nachricht dienen. Allerdings schränkt die synchrone Kommunikation potentielle Parallelarbeit durch das evt. Verzögern des Senders ein.

	mitteilungsorientiert	auftragsorientiert
synchron	Rendezvous-Technik	Remote-Procedure-Call
asynchron	No-Wait-Send	Remote-Service-Invocation

Tabelle 1: Kommunikations-Arten

Dies ist bei asynchroner Kommunikation nicht der Fall. Dafür fehlen dort die Flußkontrolle und die Zuverlässigkeit.

Beide Verfahren können durch das jeweils andere simuliert werden (vgl. [6]).

Als ein Beispiel für synchrone Kommunikation beim Menschen kann man das Telefonieren sehen, als ein Beispiel für asynchrone Kommunikation das Telefaxen.

Da bestimmte Folgen von `send/receive`-Operationen in der Praxis häufig vorkommen, faßt man diese zu sog. *Transaktionen* zusammen(vgl. [22]). Man kann diese in zwei *Transaktionsmuster* untergliedern:

1. *mitteilungsorientierte* Transaktionen. Hier endet eine Transaktion mit dem Senden einer Nachricht.

2. *auftragsorientierte* Transaktionen. Hier erwartet der Sender auf jeden Fall ein Ergebnis zurück.

Synchronisations-Grad und Transaktionsmuster führen zu der in Tabelle 1 dargestellten Klassifikation von Kommunikations-Arten:

Bei der *Rendezvous-Technik* werden Sender und/oder Empfänger so lange verzögert, bis sie ihren "Rendezvous-Punkt" erreicht haben. Die Transaktion endet dann mit Abschluß der `send`-Operation. Dieses Konzept wurde z.B. in CSP (vgl. [3]) realisiert.

Beim *No-Wait-Send* findet eine Verzögerung nur beim Empfänger, nicht beim Sender statt. Realisiert wurde dieses Konzept z.B. in der Sprache PLITS (vgl. [3]).

Beim *Remote-Procedure-Call* (RPC) wird der Sender mit dem Erreichen des `send`-statements solange verzögert, bis der Empfänger den Auftrag erhalten, ausgeführt und die Antwort zurückgeschickt hat. Aus seiner Sicht ist dieser Aufruf identisch mit einem Prozeduraufruf. Realisiert ist dieses Konzept in den Sprachen ADA, MESA, ARGUS, EPL und SR (vgl. [3]). RPC ist die zur Zeit populärste Kommunikations-Art.

Bei *Remote-Service-Invocation* muß der Sender eines Auftrags im Gegensatz zum RPC das Ergebnis explizit durch ein `receive`-statement erwarten.

Ein weiteres Merkmal ist die Struktur der Nachrichten aus der Sicht von Sender und Empfänger. Bei der von dem Betriebssystem UNIX unterstützten Inter-Prozeß-Kommunikation (IPK) (s. u.) werden Nachrichten als ASCII-Strings aufgefaßt. Bei den meisten verteilten Programmiersprachen haben Nachrichten die Form

$$\text{name}(\text{var}_1, \text{var}_2, \ldots, \text{var}_n)$$

z. B.

insert(5,TRUE)

`send`- und `receive`-statements haben dann die Form

`send` insert(5,TRUE) to empf
`receive` insert(I,FLAG).

Dies erinnert sehr stark an Prozeduren. Nachrichten werden in den meisten imperativen verteilten Programmiersprachen als Fernoperationen aufgefaßt. Bei Empfang einer Nachricht werden die aktuellen an die formalen Parameter gebunden und dann eine unter dem Nachrichtennamen definierte Operation ausgeführt (vgl. ADA, CSSA).

Es sind aber auch andere Strukturen denkbar. So können z. B. Bitvektoren versendet werden, wobei die Bedeutung der einzelnen Bits vordefiniert ist und Aktionen gemäß dem Status der gesendeten Bits ausgeführt werden. Diese Idee kann weiter entwickelt werden. Auf einer höheren Stufe kann ein Wörterbuch mit Schlüsselwörtern verwendet werden. Kommunikation besteht dann aus dem Senden und Empfangen von einzelnen Worten. Auf der nächst höheren Stufe können einfache Sätze eines vordefinierten Teilbereiches der natürlichen Sprache verwendet werden. (vgl. [8])

Die Einführung von Ports bzw. Nachrichtennamen macht ein erweitertes `receive`-statement notwendig. Es ist sinnvoll, auf die Ankunft von Nachrichten an verschiedenen Ports bzw. mit verschiedenen Namen gleichzeitig zu warten. Die meisten Sprachen sehen dafür ein `select`-statement mit folgendem Aussehen vor (vgl. CSSA, ADA):

`select`
  `receive` $\text{op}_1(\ldots)$
  `receive` $\text{op}_2(\ldots)$
    $\vdots$
  `receive` $\text{op}_n(\ldots)$
`endselect`

Hier wird auf das Ankommen einer der Nachrichten op_1, op_2, ..., op_n gewartet. Sind bei Erreichen von `select` mehrere passende Nachrichten da, wird nichtdeterministisch eine ausgewählt. Oft kann die Annahme einer Nachricht noch an eine Bedingung an den Status des Agenten oder den Inhalt der Nachricht geknüpft sein, etwa

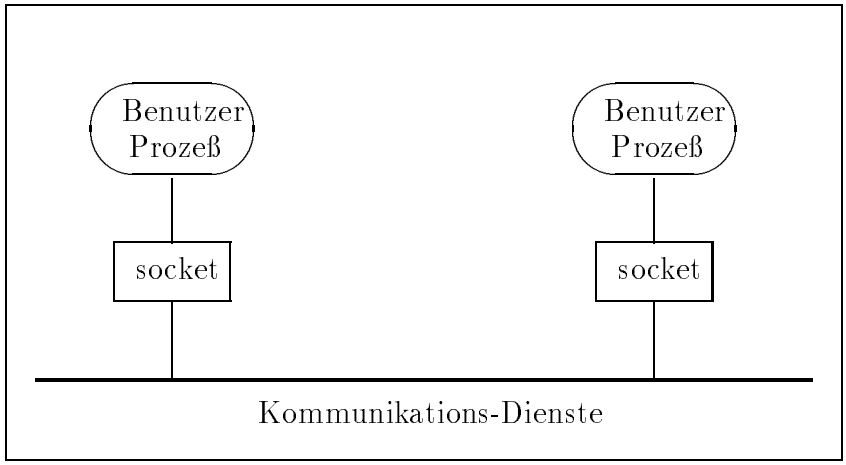

Abbildung 2: Kommunikation via sockets

`receive` fac(I) assert $I \geq 0$.

Im folgenden werden die Kommunikationsmechanismen des Betriebssystems UNIX und der Programmiersprache CSSA kurz vorgestellt. Zunächst zur IPK in UNIX.[1] Die Kommunikations-Endpunkte in den Prozessen werden *sockets* genannt. Sie entsprechen in etwa den zuvor genannten Ports, stellen jedoch immer die Verbindung zum NTS dar, also auch beim Senden. Vergleiche dazu Abbildung 2.

Es gibt zwei Arten von sockets, die *stream-sockets* und die *datagram-sockets*. Wollen zwei Prozesse über stream-sockets kommunizieren, müssen diese zuerst verbunden werden. Dies ist bei datagram-sockets nicht notwendig. Dafür haben die stream-sockets einige Vorteile gegenüber datagram-sockets was Komfort und Sicherheit angeht: Die Reihenfolge der Nachrichten bleibt erhalten, es können keine Duplikate entstehen und Nachrichten können nicht verloren gehen.

Für stream-sockets stehen neben Primitiven zum Verbindungsaufbau die Operationen

```
send(<socket>, <daten>)
recv(<socket>, var <daten>)
```

zur Verfügung. Für datagram-sockets stehen die Operationen

```
sendto(<socket>, <daten>, <empfsocket>)
recv(<socket>, var <daten>)
```

zur Verfügung.

Der Benutzer trägt selbst dafür die Verantwortung, daß dem Betriebssystem die sockets bekannt sind bzw. die notwendigen Verbindungen bestehen.

[1] Diese Darstellung läßt einige technische Details außer Betracht.

Neben diesen gibt es noch eine Reihe weiterer Kommunikations-Operationen. Vergleiche dazu etwa [13].

Nun zu der verteilten Programmiersprache CSSA (**C**omputing **S**ystem for the **S**ociety of **A**gents). Das zugrunde liegende Kommunikations-Modell ist das *Mailbox-Modell*. Allerdings wird jedem Agenten fest eine Mailbox zugeordnet. Die Kommunikation erfolgt *asynchron* und *mitteilungsorientiert*. Allerdings wird die Realisierung eines synchronen, auftragsorientierten Transaktionsmechanismus unterstützt. Dazu besteht die Möglichkeit, einer Nachricht eine Antwortverpflichtung mitzugeben. Der Sender kann dann die Antwort mittels eines `receive`-statements abwarten. Nachrichten haben die Form:

$$\text{OpBezeichner}(\text{par}_1, \text{par}_2, \ldots, \text{par}_n)$$

Nachrichten können nur an bekannte Agenten verschickt werden. Die Entnahme der Nachrichten aus der Mailbox geschieht implizit durch das Ausführen einer Operation mit passenden Namen und Parameter-Listen. Zusätzlich kann eine Bedingung an den Inhalt der Nachricht formuliert werden.

Für ein ausführliche Beschreibung und Beispiele sei auf [19] verwiesen.

2.2 Koordination

Wollen Agenten kooperieren, so müssen sie ihre Handlungen aufeinander abstimmen. Der Koordinations-Overhead sollte dabei allerdings möglichst gering sein. Die Synchronisation verteilter Prozesse ist zentrales Thema der Betriebssystem-Theorie und wichtiges Thema der Datenbank-Theorie. Hier wurden eine Vielzahl von Konzepten von *busy-waiting* über *Semaphore* bis hin zu *Monitoren* (s. u.) entwickelt. All diese Konzepte lassen sich unter dem Attribut *speicherbasiert* zusammenfassen. Da man bei verteilten Systemen nicht mehr von gemeinsamem Speicher ausgehen kann, wurden und werden in der Theorie der verteilten Betriebssysteme und Datenbanksysteme neue, allein auf Nachrichtenaustausch beruhende Synchronisationsmechanismen entwickelt. Die Synchronisations-Aufgaben bei Prozessen lassen sich in drei typische Muster einteilen (vgl [19]): Zwei oder mehrere Prozesse

- dürfen gewisse Aktionen *nicht gleichzeitig* ausführen,
- müssen gewisse Aktionen *gleichzeitig* ausführen,
- müssen bzgl. der Ausführung gewisser Aktionen eine bestimmte *Reihenfolge* einhalten.

In die erste Klasse fällt z. B. die verteilte Vergabe von exklusiv nutzbaren Betriebsmitteln. In der Theorie der verteilten Systeme faßt man die Lösungsverfahren zu diesem Problem unter dem Begriff *Election-Verfahren* zusammen. In die zweite Klasse fallen u. a. konsistente

Bestimmungen globaler Eigenschaften wie *Terminierung* oder *Verklemmung*. Der Begriff der *Gleichzeitigkeit* wird hier noch näher zu erläutern sein.

Im folgenden werden zuerst die speicherbasierten Koordinations-Mechanismen in ihrer zeitlichen und logischen Entwicklung vorgestellt. Danach werden die Grundideen der nachrichtenbasierten Koordination vorgestellt.

2.2.1 Speicherbasierte Koordinations-Mechanismen

Hier sind im wesentlichen zwei Synchronisations-Typen zu unterscheiden: *wechselseitiger Ausschluß* und *Bedingungs-Synchronisation* (vgl. [2]). Beim wechselseitigen Ausschluß geht es darum, einem Agenten eine Ressource für eine Folge von Operationen exklusiv zur Verfügung zu stellen. Dies ist immer dann notwendig, wenn paralleles Nutzen der Ressource zu Inkonsistenzen im Ablauf führen würde – etwa das parallele Benutzen eines Druckers.

Bei der Bedingungs-Synchronisation geht es darum, einen Agenten solange zu verzögern, bis eine bestimmte Bedingung eingetreten ist. So sollte das Schreiben in einen vollen Puffer solange verzögert werden, bis im Puffer wieder Platz ist.

Eine sehr einfache Art, Synchronisation zwischen Agenten zu realisieren, besteht im *Setzen und Testen* von gemeinsamen Variablen. Hardwaremäßig muß dazu eine (unteilbare) Operation zum Lesen und Schreiben einer sog. *Lock-Variablen* zur Verfügung stehen. Bei der Bedingungs-Synchronisation signalisiert ein Prozeß durch Setzen einer Lock-Variablen das Zutreffen einer Bedingung. Ein Prozeß, der auf das Eintreten dieser Bedingung wartet, muß beständig die Variable lesen (und setzen). Man nennt diese Methode deshalb auch *busy waiting*.

Neben der unnötigen Belastung des Prozessors beim Warten ist die Unstrukturiertheit und Unübersichtlichkeit von Realisierungen des wechselseitigen Ausschlusses eine Hauptschwäche der set-and-test Methode (vgl. [2]).

Diese Schwächen versuchte Dijkstra mit der Einführung von *Semaphoren* zu überkommen. Busy waiting wird dadurch vermieden, daß Agenten, die auf das Freiwerden einer Ressource oder das Eintreten einer Bedingung warten, von einer zentralen Instanz – etwa dem Betriebssystem – "eingeschläfert" bzw. bei Eintreten eines entsprechenden Ereignisses "geweckt" werden. Sie belegen während dieser Zeit nicht den Prozessor.

Ein Semaphor ist eine Variable, die die Werte FREI und BELEGT annehmen kann. Auf Semaphoren sind zwei Operationen definiert: **P** und **V**. Eine **P**-Operation auf ein belegtes Semaphor führt dazu, daß der Agent "an dem Semaphor wartet". Erst durch eine **V**-Operation auf das Semaphor wird der Agent wieder reaktiviert. Damit lassen sich sowohl wechselseitiger Ausschluß als auch Bedindungs-Synchronisation realisieren. (vgl. [2]).

Beim wechselseitigen Ausschluß wird einem kritischen Bereich ein Semaphor zugeordnet. Dieser Bereich wird dann von einer **P**-Operation und einer **V**-Operation eingeschlossen:

P(mutex)
⋮
kritischer Bereich
⋮
V(mutex)

Semaphore haben zwei wesentliche Schwächen. Zum einen gibt es auf der sprachlichen Ebene keine Möglichkeit, die Zugehörigkeit von Semaphoren zu kritischen Abschnitten oder Bedingungen auszudrücken. Zugriffsfunktionen stehen losgelöst von gemeinsam benutzten Daten. Zum anderen läßt die sprachliche Unterstützung der Bedingungs-Synchronisation zu wünschen übrig. Lösungen für komplexere Aufgaben werden sehr unübersichtlich (vgl. [2]).

Angfang der siebziger Jahre entwickelten Brinch Hansen und Hoare unabhängig voneinander das *Monitor-Konzept* (vgl. [2]). Ein Monitor faßt gemeinsam von mehreren Prozessen benutzte Datenstrukturen und die darauf definierten Zugriffsfunktionen in Form von Abstrakten Datentypen zusammen. Die Funktionen eines Monitors schließen sich wechselseitig aus, sodaß immer nur ein Prozeß die Monitordaten manipulieren kann. Sprachkonstrukte zur Bedingungs-Synchronisation kommen hinzu.

Somit ergeben sich gegenüber Semaphoren wesentlich strukturiertere Problemlösungen. Außerdem werden die Implementierungsdetails der Zugriffsfunktionen in der Monitorimplementierung verborgen. Dies unterstützt das Prinzip des *Information Hiding.*

Probleme mit Monitoren treten im wesentlichen bei geschachtelten Monitor-Aufrufen auf (vgl. [17]).

Die weitere Entwicklung von speicherbasierten Mechanismen ging und geht in zwei Richtungen. Zum einen wurde versucht, die Problemlösungen noch besser zu strukturieren. Eine Möglichkeit dazu bieten die *path expressions.* Bei diesen werden alle Constraints für einen kritischen Bereich – z. B. auch Einschränkungen an die Reihenfolge von Monitoraufrufen – an zentraler Stelle deklarativ beschrieben (vgl. [2]).

Die andere Richtung betrifft die Granularität der Sperren. Sind die gemeinsam benutzten Datenbereiche sehr groß – wie etwa bei Datenbanken – dann ist es nicht sinnvoll, jeweils die komplette Datenbank zu sperren. Hier müssen Konzepte entwickelt werden, die zum einen möglichst kleine Bereiche sperren und zum anderen effizient zu implementieren sind.

Zum Abschluß dieses Abschnitts soll noch auf ein wesentliches Problem bei all diesen Konzepten eingegangen werden: *Die Deadlocks.* Deadlocks oder Verklemmungen entstehen durch zirkuläre Wartebedingungen. Typisches Beispiel ist: A und B wollen beide die Res-

sourcen R_1 und R_2 nutzen, A hat bereits R_1, B hat R_2. Dann warten A auf B und B auf A – endlos. Deadlocks können nur entstehen, falls alle vier der folgenden Bedingungen erfüllt sind (vgl. [22]):

1. Die Ressourcen können nur exklusiv genutzt werden.
2. Agenten sind bereits in Besitz von Ressourcen, wenn sie weitere anfordern.
3. Belegte Ressourcen können nicht zwangsweise entzogen werden.
4. Es existiert eine zirkuläre Kette von Agenten derart, daß jeder Agent ein oder mehrere Exemplare von Ressourcen besitzt, die vom nächsten in der Kette gefordert werden.

Es gibt prinzipiell zwei Arten, wie man gegen Deadlocks vorgeht. Man kann zum einen ihre Entstehung verhindern, indem man bei jeder Ressourcen-Anforderung testet, ob diese zu einem Deadlock führen kann. Oder man tut nichts gegen die Entstehung, testet aber periodisch, ob ein Deadlock vorliegt und löst ihn gegebenenfalls auf.

Die Algorithmen für Vermeidung und Erkennung von Deadlocks sind jedoch sehr aufwendig. Noch schwieriger wird das Problem bei Systemen ohne gemeinsamen Speicher. Hier ist es schwierig, überhaupt eine konsistente Sicht auf den Betriebsmittel-Zustand zu erhalten (s. u.).

2.2.2 Nachrichtenbasierte Koordinations-Mechanismen

In Systemen, die keinen gemeinsamen Speicher besitzen, muß die Koordination über Nachrichten erfolgen. Zentrales Problem ist es, eine konsistente Sicht auf den globalen System-Zustand zu erhalten. Information von anderen ist nur über Nachrichten zu bekommen, die Laufzeit der Nachrichten ist aber nicht vernachlässigbar. Information kann bei ihrer Ankunft schon wieder veraltet sein.

In [19] werden Basisprobleme für die Koordination in verteilten Systemen herausgearbeitet. Aus den dafür entwickelten Basisalgorithmen lassen sich Algorithmen für komplexere Aufgaben aufbauen. Im wesentlichen sind diese Basisprobleme: *Election, Schnappschuß* und *verteilte Terminierung.*

Im folgenden werden diese Probleme und ihre Lösungsansätze kurz vorgestellt. Es zeigt sich, daß bei all diesen Problemen der Begriff der Zeit in verteilten Systemen eine wichtige Rolle spielt. Zeit wird in Abschnitt 3 behandelt.

Election

Ein Election-Problem tritt immer dann auf, wenn mehrere gleichwertige Agenten sich auf einen eindeutigen Vertreter (leader) einigen müssen. Solche Probleme treten in der Praxis z. B. bei Token-Protokollen auf, wo bei Verlust des Tokens genau ein neues Token generiert werden muß. (In der Tat scheint der große Aufwand, der zur Lösung des Election-Problems getrieben wird, direkt mit der großen Popularität von z. B. Token-Ring und Token-Bus für LAN's zusammenzuhängen.)

Election-Algorithmen dienen als *Basisalgorithmen* für verwandte Probleme. So eignen sie sich z. B. zum *Symmetrisieren* von verteilten Algorithmen. Unter einem symmetrischen Algorithmus versteht man einen Algorithmus, der parallel von mehreren Agenten angestoßen werden kann. Ein unsymmetrischer Algorithmus muß von genau einem Agenten angestoßen werden. Election-Algorithmen sind inhärent symmetrisch. Durch Vorschalten einer Election-Phase erhält man aus einem unsymmetrischen Algorithmus einen symmetrischen.

Ein dem Election-Problem verwandtes Problem ist das *Maximums-Problem*. I. allg. kann man davon ausgehen, daß jedem Agenten eine eindeutige Identifikation – etwa in Form einer ganzen Zahl – zugeordnet werden kann. Unter dem Maximums-Problem versteht man die Bestimmung des Agenten mit der größten Identifikation. Da beim Election-Problem keine Fairness gefordert wird, ist jede Lösung des Maximums-Problems eine Lösung des Election-Problems.

Für Aufgaben wie das Generieren eines Token ist eine solche Lösung befriedigend. Für Aufgaben wie die Vergabe von Betriebsmitteln müssen jedoch andere Maximalitäts-Kriterien gefunden werden. Eines ist z. B. die "früheste" Anforderung. Hier kommt ein gemeinsamer Zeitbegriff ins Spiel. Dazu mehr in Kapitel 3.

Ein Lösungsansatz für das Maximums-Problem stellt die schrittweise *Approximation* des Maximums dar. Dazu initialisiert jeder Agent eine Variable mit der kleinstmöglichen Identität und wartet, bis er von einer größeren Identität erfährt. Als Start erfährt jeder Agent von seiner eigenen Identität. Sobald ein Agent von einer größeren Identität als der aktuell gemerkten erfährt, teilt er diese den anderen Agenten mit. Es ist leicht einzusehen, daß irgendwann jeder die größte Identität als aktuelle speichert, vorausgesetzt kein Agent ist isoliert. Damit blockiert der Algorithmus. Aber kein Agent weiß, daß er das Ergebnis weiß.

Es ist ein nicht-triviales Problem, die verteilte Terminierung festzustellen. Mehr dazu weiter unten.

Es existieren eine Vielzahl von Varianten an Election-Algorithmen. Insbesondere für

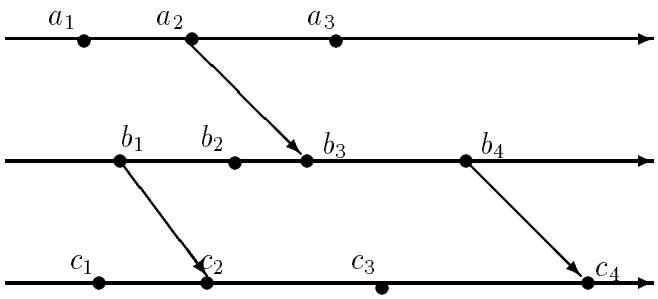

Abbildung 3: Ein Zeitdiagramm

spezielle Topologien[2] wie *unidirektionale Ringe, bidirektionale Ringe, Bäume* etc. wurden spezielle Algorithmen entwickelt und bezüglich ihrer Nachrichtenkomplexität untersucht. Für einen umfassenden Überblick siehe [19].

Schnappschuß

Beim Schnappschuß geht es darum, eine in einem zu präzisierenden Sinne optimale Approximation des "momentanen" globalen Zustandes zu ermitteln. Der Zustand, in dem sich alle Agenten eines Systems "gleichzeitig" befinden, ist nur unter einer relativistischen Sicht der Gleichzeitigkeit zu ermitteln. Darunter ist folgendes zu verstehen: Wenn ein Ereignis E_1 ein Ereignis E_2 beeinflußen kann, so muß E_1 früher als E_2 stattfinden. Man nimmt diese Beobachtung als Grundlage und definiert zwei Ereignisse, die sich nicht beeinflußen können, als gleichzeitig. In verteilten Systemen kann ein Ereignis E_1 ein Ereignis E_2 beeinflußen, falls:

- E_1 und E_2 im selben Agenten stattfinden und E_1 zeitlich vor E_2 liegt.
- E_1 und E_2 in unterschiedlichen Agenten A_1 und A_2 stattfinden und A_2 von A_1 – direkt oder indirekt – zwischen E_1 und E_2 eine Nachricht erhält.

Um diese Idee zu präzisieren, werden im folgenden einige Begriffe eingeführt. Die Aktivitäten eines Agenten werden als Folge von atomaren *Ereignissen* angesehen. e, e', e_i bezeichnen Ereignisse. e heißt *direkter Vorgänger* von e' ($e \ll e'$), falls entweder e und e' Ereignisse eines Agenten sind und e (zeitlich lokal) direkt vor e' stattfindet oder e das Senden und e' das Empfangen derselben Nachricht bezeichnen.

Unter der *globalen Ereignisordnung* oder *Kausalordnung* $<$ versteht man die transitive Hülle von $\ll$. $<$ ist eine Halbordnung auf allen Ereignissen.

[2] Unter einer Topologie versteht man die Struktur der Bekanntschaftrelation der Agenten

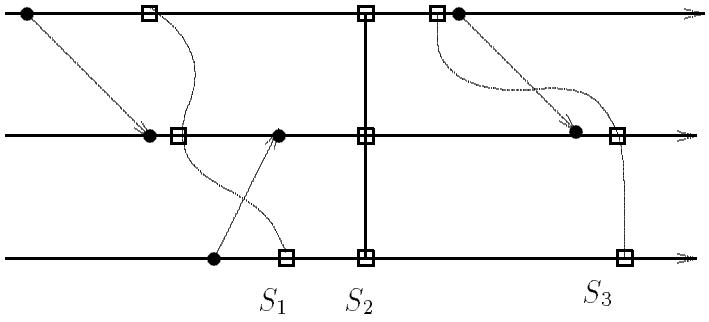

Abbildung 4: Schnitte im Zeitdiagramm

Zwei Ereignisse e_1 und e_2 gelten als kausal unabhängig oder gleichzeitig ($e_1 \| e_2$), falls weder $e_1 < e_2$ noch $e_2 < e_1$ gilt. Die Begriffe sollen im sog. *Zeitdiagramm* veranschaulicht werden. Die Zeit läuft dabei von links nach rechts. Für jeden Agenten wird eine Zeitachse eingetragen. Ereignisse werden durch Punkte auf den Zeitachsen dargestellt, Nachrichten durch diagonal von links nach rechts verlaufende Pfeile. Betrachte dazu Abbildung 3. Es gilt z. B. $a_1 \ll a_2 \quad a_1 < b_4 \quad a_1 \parallel b_2$.

Zeitliche Begriffe wie *später* kommen durch den abstrakten Begriff *Schnitt* ins Spiel. Doch zunächst der Begriff der *lokalen Ereignisordnung* $<_1$. Für zwei Ereignisse e_1 und e_2 gilt $e_1 <_1 e_2$, falls $e_1 < e_2$ gilt und beide Ereignisse im gleichen Prozeß stattfinden. Zu einer Menge von Ereignissen E heißt $S \subseteq E$ *Schnitt* von E, falls mit $e \in S$ und $e' <_1 e$ auch $e' \in S$ gilt.

Graphisch kann man den Schnitt im Zeitdiagramm als von oben nach unten verlaufende Linie darstellen. Zum Schnitt gehören dann alle Ereignisse links von dieser Linie. Siehe dazu Abbildung 4.

Wünschenswert wäre ein Schnitt zu einem (globalen) Zeitpunkt. Im Diagramm entspräche dies einer senkrechten Linie (vgl. S_2). Dieser wäre jedoch nur realisierbar, wenn es entweder einen globalen Beobachter gäbe, der alle Ereignisse "sofort" wahrnimmt, oder wenn Nachrichtenlaufzeiten vernachlässigbar wären. Da dies beides nicht der Fall ist, muß man sich mit weniger zufrieden geben. Wenn ein Schnitt keine *tatsächliche* Trennlinie zwischen Vergangenheit und Zukunft realisiert, dann sollte er wenigstens eine *mögliche* Trennlinie realisieren. D. h., er sollte keinen offensichtlichen Widerspruch enthalten. Ein offensichtlicher Widerspruch wäre z. B., wenn eine in der Zukunft abgeschickte Nachricht in der Vergangenheit ankäme (vgl. S_3). Dies wird durch den Begriff des *konsistenten Schnittes* präzisiert. Ein Schnitt S von E heißt konsistenter Schnitt von E, falls mit $e \in S$ und $e' < e$ auch $e' \in S$ gilt. Auf der Menge der konsistenten Schnitte wird eine Halbordnung *später* definiert, wobei ein Schnitt S_1 *später* heißt als ein Schnitt S_2, falls $S_2 \subseteq S_1$. Die

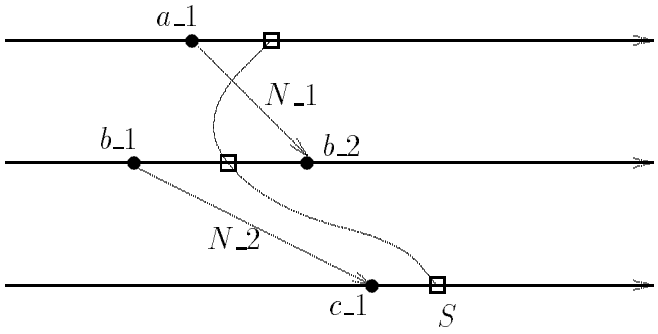

Abbildung 5: Zeitdiagramm

Menge der konsistenten Schnitte bildet zusammen mit der *später*-Relation einen Verband. Dieser bestimmt die Struktur einer virtuellen globalen Zeit, wie sie in Kapitel 3 betrachtet wird. Ausgehend von konsistenten Schnitten wird nun der Begriff des globalen Zustandes definiert(!). Er besteht im wesentlichen aus den "momentanen" Zuständen der Agenten und den "auf dem Weg befindlichen" Nachrichten.

$P_i(e)$ bezeichne den lokalen Zustand des Agenten P_i nach dem lokalen Ereignis e. Der *globale Zustand* eines konsistenten Schnittes S ist ein Paar $Z(S) \equiv (P(S), N(S))$ mit $P(S) \equiv \bigotimes P_i(e_i)$, wobei e_i das jeweils größte Ereignis bzgl. $<_1$ von P_i in S ist (falls ein solches existiert, ansonsten bezeichne $P_i(e_i)$ den Anfangszustand von P_i) und $N(S)$ die Menge der Nachrichten bezeichnet, deren jeweiliges Sendeereignis, nicht aber ihr Empfangsereignis, in S liegt.

Das *Schnappschuß-Problem* besteht nun darin, zu einem beliebigen Zeitpunkt, ausgehend von einem Initiator-Agenten, einen konsistenten Schnitt und den zugehörigen globalen Zustand zu ermitteln.

Man beachte, daß der so gelieferte globale Zustand nicht unbedingt die reale Situation widerspiegelt. Abbildung 5 möge eine reale Abfolge beschreiben. Mit dem konsistenten Schnitt S wird ein Zustand geliefert, bei dem die Nachricht N_1 noch unterwegs ist, Nachricht N_2 jedoch schon angekommen ist. Von außen gesehen kommt N_1 vor N_2 an, innerhalb des Systems ist das jedoch nicht "wahrnehmbar", c_1 und b_2 sind gleichzeitig.

Was nutzt ein solcher globaler Zustand? Er nutzt einem dann etwas, wenn man *stabile Zustände* des Systems erkennen oder sich *monoton* ändernde Zustände approximieren will. Dazu folgendes. Angenommen, zu einem Zeitpunkt r, dargestellt durch den (hypothetischen) Schnitt R, liefere ein Schnappschuß den globalen Zustand $Z(S)$. Zu dem Zeitpunkt r werde zusätzlich ein weiterer Schnappschuß initialisiert, der den Zustand $Z(S')$ liefere. Dann gilt sicherlich: $S \subseteq R \subseteq S'$. Eine Funktion f von globalen Zuständen heiße *isoton*, falls für $S_1 \subseteq S_2$ gilt $f(Z(S_1)) \leq f(Z(S_2))$, wobei $\leq$ eine Ordnung auf dem Wertebereich von

f ist. Beschreibt man eine Eigenschaft des Systems durch eine isotone Funktion f, so gilt: $f(Z(S)) \leq f(Z(R)) \leq f(Z(S'))$. Damit kann man durch eine Folge von Schnappschüssen die Eigenschaft f für den momentanen Zustand approximieren.

Eine Sonderrolle spielen Prädikate. Diese können als Funktionen mit dem Wertebereich {TRUE, FALSE} aufgefaßt werden.. Mit der Ordnung FALSE < TRUE gilt für ein isotones Prädikat Pr: Gilt es einmal, so gilt es danach immer, d. h. mit $Pr(Z(S)) = \text{TRUE}$ und $S \subseteq S'$ gilt $Pr(Z(S')) = \text{TRUE}$. Solche Prädikate heißen *stabil*. Stabile Prädikate haben zwei "günstige" Eigenschaften (vgl. [19]):

- Liefert ein Schnappschuß einen globalen Zustand $Z(S)$ mit $Pr(Z(S)) = \text{TRUE}$, so gilt Pr "momentan".
- Gilt Pr "momentan", so liefert ein jetzt oder später gestarteter Schnappschuß einen Zustand $Z(S)$ mit $Pr(Z(S)) = \text{TRUE}$.

Wesentlich ist, daß sich viele interessante Eigenschaften durch isotone Funktionen bzw. stabile Prädikate beschreiben lassen. So sind Deadlock und Terminierung stabil und die Anzahl der insgesamt verschickten oder empfangenen Nachrichten isoton. Ziel bei der Betrachtung verteilter Systeme sollte es also sein, Eigenschaften herauszukristallisieren, die sich durch isotone Funktionen bzw. stabile Prädikate beschreiben lassen.

Im folgenden wird das Prinzip eines einfachen (unsymmetrischen) Schnappschuß-Algorithmus vorgestellt. Grundidee dabei ist, daß Agenten und Nachrichten *färbbar* sind. Sie können die Farben schwarz und rot annehmen. Nachrichten tragen immer die Farbe des sendenden Agenten. Zu Beginn sind alle Agenten und Nachrichten schwarz. Ein Übergang von schwarz nach rot markiert einen Schnitt von Vergangenheit zur Zukunft. Nachrichten werden eingeteilt in *Kontrollnachrichten* und *Basisnachrichten*. Kontrollnachrichten sind diejenigen, die zur Ermittlung des Schnittes verschickt werden, Basisnachrichten alle anderen. Mit dem Start wird der Initiator des Schappschusses rot und sendet rote Kontrollnachrichten als Schnappschuß-Aufforderung an alle ihm bekannten Agenten. Sobald ein Agent von einem Schnappschuß erfährt, wird er ebenfalls rot (damit sendet er in Zukunft nur noch rote Nachrichten) und schickt Zustandsinformation an den Initiator.

Der kritische Fall ist derjenige, wenn ein schwarzer Agent eine rote Basisnachricht erhält. Es handelt sich dabei um eine Nachricht aus der Zukunft. Würde er dem Initiator diese Nachricht als empfangen melden, käme es zu Inkonsistenzen. Es bieten sich zwei Lösungsmöglichkeiten an. Entweder verzögert der Agent den Empfang der Nachricht, bis die entsprechende Kontrollnachricht auch ihn erreicht hat. Oder er betrachtet die rote Basisnachricht als Schnappschuß-Aufforderung und ermittelt seinen Zustand bevor er sie verarbeitet. Die zweite Möglichkeit verhindert Verzögerungen.

Zur Ermittlung der auf dem Weg befindlichen Nachrichten gibt es verschiedene Möglichkeiten. Zum einen kann sich jeder Agent alle gesendeten und empfangenen Nachrichten merken. Bei einem Schnappschuß sendet er diese Information an den Initiator. Dieser kann daraus die Differenz berechnen. Eine zweite Methode basiert auf der Beobachtung, daß die gesuchten Nachrichten gerade diejenigen sind, die im schwarzen Bereich abgeschickt wurden und im roten Bereich ankommen. Der Initiator wartet, bis ihm alle derartigen Nachrichten gemeldet sind. Das Problem ist es herauszufinden, wann er aufhören kann zu warten. Eine Lösung besteht darin, daß die Agenten die gesendeten und empfangenen Nachrichten zählen und diese Anzahlen beim Schnappschuß mitsenden. Der Vorteil gegenüber Variante 1 ist, daß nur die Anzahlen, nicht die Nachrichten selbst gespeichert werden müssen, der Nachteil ist eine u. U. lange Wartezeit.

Es ist leicht zu zeigen, daß die Menge der schwarzen Ereignisse einen konsistenten Schnitt bildet und daß der gemeldete Zustand der globale Zustand des durch die schwarzen Ereignisse definierten Schnittes ist.

In der Literatur werden eine große Anzahl von Schnappschuß-Algorithmen behandelt. Insbesondere werden auch symmetrische Algorithmen vorgestellt. Für eine umfassende Übersicht siehe [19].

Verteilte Terminierung

Hierbei geht es um das Problem, wie in einem verteilten System festgestellt werden kann, daß eine verteilte Berechnung beendet ist. Verteilte Terminierung kann auf das Schnappschuß-Problem zurückgeführt werden. Wie angegeben handelt es sich um ein stabiles Prädikat auf dem globalen Zustand. Dennoch wurde gerade dieses Problem sehr intensiv untersucht. 1980 zum ersten Mal formuliert, existierten 1987 bereits über 50 Lösungsvorschläge.

Für sich selbst kann jeder Agent sehr wohl entscheiden, ob er mit seinen Aufgaben fertig ist. Da er aber (jedenfalls nicht ohne weiteres) keine konsistente globale Sicht hat, weiß er nicht, wie weit die anderen sind bzw. ob er von den anderen noch Arbeit bekommen wird. "Einfaches" Nachfragen führt zu den gleichen Problemen wie beim Schnappschuß.

Prinzipiell läßt sich das Terminierungs-Problem mit einem Schnappschuß-Algorithmus lösen. Dieser wird solange iteriert, bis die Terminierung über dem globalen Zustand festgestellt wird. Die in der Literatur vorgestellten, auf das spezielle Problem zugeschnittenen Lösungen versuchen jedoch, Vorteile aus der speziellen Situation zu ziehen, um die Algorithmen nach verschiedenen Bewertungskriterien bzw. für bestimmte Topologien zu optimieren.

Für eine nähere Betrachtung muß der Begriff der Terminierung präzisiert werden. Man unterscheidet grundsätzlich zwei verschiedene Auffassungen von Terminierung: *Kommunikationsorientierte Terminierung* und *Ereignisorientierte Terminierung*. Bei der kommu-

nikationsorientierten Terminierung betrachtet man eine verteilte Berechnung als beendet, wenn keine Nachrichten mehr unterwegs sind und die Agenten in einem Zustand sind, in dem sie keine Nachrichten mehr senden werden. Bei der ereignisorientierten Terminierung sieht man die Terminierung als Prädikat über dem Gesamtzustand an. Das Ende der Berechnung ist erreicht, wenn das Prädikat zutrifft. Dabei können sehr wohl noch Nachrichten unterwegs sein. Die ereignisorientierte Terminierung ist in hohem Maße problemspezifisch. Oft ist es auch ein nicht-triviales Problem, die Korrektheit des Terminierungs-Prädikates zu zeigen.

Bei der kommunikationsorientierten Terminierung sucht man nach Algorithmen, die problemunabhängig arbeiten. Von den einzelnen Agenten muß man nur wissen, ob sie "zur Zeit" aktiv oder passiv sind, d. h. ob sie Nachrichten senden können oder nicht. Allerdings lassen sich damit problemabhängige Fälle nicht bearbeiten. Außerdem lassen sich beabsichtigte Terminierung und unbeabsichtigte Terminierung – z. B. ein Deadlock – nicht ohne weiteres auseinanderhalten.

Für eine weitergehende Betrachtung, insbesondere eine umfassende Übersicht über veröffentlichte kommunikationsorientierte Terminierungsalgorithmen, sei auf [19] verwiesen.

3 Zeit in verteilten Systemen

Wenn wir Vorgänge in der Zeit in Form von Zeitdiagrammen darstellen, gehen wir davon aus, daß wir jedem Ereignis einen Zeitpunkt zuordnen können, und daß diese Zeitpunkte linear geordnet sind, bzw., daß es einen idealen Beobachter gibt, der jedes Ereignis sofort wahrnimmt.

In der Realität approximieren wir diese ideale Zeitvorstellung durch hinreichend synchron laufende Uhren. Und obwohl damit Inkonsistenzen der Art, daß "spätere" Vorgänge "frühere" beeinflussen können, nicht ausgeschlossen sind, reicht dieses Modell erstaunlicherweise aus, um das Leben in der Welt zu koordinieren[3].

In Computern laufen Berechnungen jedoch in ganz anderen Zeiträumen ab. In einer millionstel Sekunde, was etwa der Zeit entspricht, die Quarzchronographen mindestens divergieren, führen Computer mehr als 10 Rechenschritte aus.

Dies ist aber nicht allein der Grund dafür, das Modell einer linearen Zeit in Frage zu stellen. Auch in unserer physikalischen Realität ist spätestens seit Entwicklung der Relativitäts-Theorie fraglich, ob unsere Vorstellung einer linear geordneten Zeit sinnvoll ist. Aufgrund der durch die Lichtgeschwindigkeit begrenzten Ausbreitungsgeschwindigkeit von Signalen ergeben sich im linearen Modell Inkonsistenzen. So schreibt Kurt Gödel in seinen

[3]Lamport nennt dies "das Mysterium des Universums"

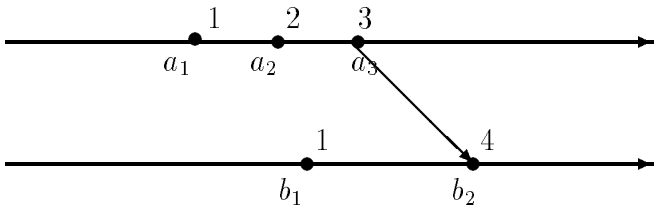

Abbildung 6: Zeitdiagramm

"Bemerkungen über die Beziehungen zwischen der Relativitätstheorie und der idealistischen Philosophie" [12]: *"Die Behauptung, die Ereignisse A und B hätten gleichzeitig stattgefunden, verliert ihren objektiven Sinn insofern, als ein anderer Beobachter mit dem gleichen Anspruch auf Richtigkeit behaupten kann, A und B hätten nicht gleichzeitig stattgefunden."*

Im folgenden werden zwei Modelle der Zeit als halbgeordnete Struktur – die *Lamport-Zeit* und die *Vektor-Zeit* – vorgestellt, die die in Kapitel 2.2.2 definierte Ereignis- bzw. Kausal-Struktur erhalten.

Es wird sich zeigen, daß die Vektorzeit eine bemerkenswerte Analogie zu der von H. Minkowski als ein geeignetes Modell für die relativistische Realität 1928 eingeführte *Raum-Zeit* aufweist.

L. Lamport geht in [16] aus von der Kausalordnung $<$ auf Ereignissen, die er *happened-before*-Relation nennt. Ziel ist es, diese Relation für Ereignisse zu berechnen. Dazu dienen die *logischen Uhren*. Technisch gesehen handelt es sich dabei um einen Mechanismus, um Ereignissen in einer sinnvollen Weise einen Zeitstempel zuzuordnen. Mathematisch gesehen ist eine logische Uhr C eine ordnungserhaltende Funktion von der Menge der Ereignisse E in eine geordnete Menge von Zeitpunkten T,d. h.: $C : E \to T$ mit der *Uhren-Bedingung*:

$$\forall e, e' \in E : e < e' \rightsquigarrow C(e) < C(e')$$

Lamport schlägt als Menge von Zeitpunkten T die Menge der natürlichen Zahlen $\mathbb{N}$ mit der üblichen Ordnung $<$ vor. Als Realisierung der logischen Uhren schlägt er vor, in jedem Agenten A_i einen Zähler C_i zu installieren, der zwischen je zwei Ereignissen inkrementiert wird. Jedem Ereignis wird als Zeitstempel der lokale Zählerstand bei seinem Eintritt zugeordnet. Nachrichten tragen als Zeitstempel die aktuellen Zählerstände ihrer Sender. Bei Empfang einer Nachricht mit Zeitstempel T_i wird der Zählerstand falls nötig so inkrementiert, daß er größer als T_i ist. Der Zeitstempel eines Ereignisses e_j ist dann $C_i(e_j)$, falls sich e_j im Agenten A_i ereignet: $C(e_j) = C_i(e_j)$.

Eine derart realisierte logische Uhr erfüllt die Uhrenbedingung, die Kausal-Ordnung wird homomorph übertragen. Allerdings gilt nicht die Umkehrung der Uhrenbedingung.

Aus $C(e) < C(e')$ folgt nicht $e < e'$. Kausal unabhängige Ereignisse werden eventuell "geordnet". Siehe dazu Abbildung 6. a_2 und b_1 sind kausal unabhängig, aber $C(b_1) = 1 < 2 = C(a_2)$. Eine isomorphe Einbettung der Ereignisstrukturen in $(\mathbb{N}, <)$ ist prinzipiell nicht möglich, da $(\mathbb{N}, <)$ total geordnet ist.

Für bestimmte Probleme ist dies kein Nachteil, sondern im Gegenteil ein Vorteil. So gibt Lamport in [16] einen auf der totalen Ordnung auf $\mathbb{N}$ basierenden Algorithmus für den *wechselseitigen Ausschluß* von Agenten an. Darauf aufbauend lassen sich *verteilte Semaphore* realisieren (vgl. [16, 23]).

Für andere Probleme aber wäre es wünschenswert, die Kausalstruktur isomorph zu übertragen. Dies leistet die *Vektor-Zeit*. Sie erfüllt die *strenge Uhren-Bedingung*:

$$\forall e, e' \in E : e < e' \text{ gdw } C(e) < C(e')$$

Die Idee ist dabei die folgende. Zeitstempel sind Vektoren aus $\mathbb{N}^n$, wobei n die Anzahl der Agenten ist. Jeder Agent A_i unterhält einen Vektor $C_i \in \mathbb{N}^n$, wobei er in der i-ten Komponente ($C_i[i]$) seine lokale Zeit vermerkt. Diese inkrementiert er jeweils zwischen zwei lokalen Ereignissen. In den restlichen Komponenten vermerkt er sein Wissen über die lokalen Zeiten der anderen Agenten. Idealerweise wären dies deren aktuelle Zählerstände. Dieser nicht erreichbare Zustand (eines idealen Beobachters) wird in dem Sinne optimal approximiert, daß jede Nachricht mit dem Zeitvektor des Senders versehen wird und der Empfänger die Zeitkomponenten der anderen eventuell nach oben korrigiert. Jeder Agent hat in seiner Komponente natürlich den höchsten Wert, d. h. $\forall i, j : C_i[i] \geq C_j[i]$ Der Zeitstempel eines Ereignisses ist analog zur Lamport-Zeit als aktueller Zeitvektor desjenigen Agenten, in dem das Ereignis stattfindet, definiert.

Auf $\mathbb{N}^n$ sind die Relationen $\leq$, $<$, $\parallel$ wie folgt definiert:

$$\begin{aligned} u \leq v \quad &\text{gdw} \quad u[i] \leq v[i], \quad i = 1..n \\ u < v \quad &\text{gdw} \quad u \leq v \text{ und } u \neq v \\ u \parallel v \quad &\text{gdw} \quad \neg(u < v) \text{ und } \neg(v < u) \end{aligned}$$

Es läßt sich zeigen, daß diese Art der Zuordnung die strenge Uhren-Bedingung erfüllt und damit gilt auch:

$$\forall e, e' \in E : e \parallel e' \text{ gdw } C(e) \parallel C(e').$$

Es bestehen interessante Zusammenhänge zwischen konsistenten Schnitten und der Vektor-Zeit. Bezeichnet X einen Schnitt und c_i das jeweils maximale Ereignis von Agent A_i in X, so heißt der durch $t_x = \sup(C(c_1), C(c_2), \ldots, C(c_n))$ definierte Wert *Zeit des Schnittes* X. Das Supremum wird dabei komponentenweise gebildet.

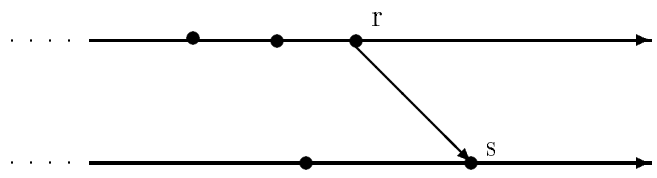

Abbildung 7: Ausschnitt aus einem Zeitdiagramm

Zwei verschiedenen Schnitten ein und der selben Ereignisstruktur kann der gleiche Zeitvektor zugeordnet werden. Dies gilt jedoch nicht bei konsistenten Schnitten. Dort ist die Zuordnung eindeutig. Diese Beobachtung führt zu folgendem Theorem, das sich als praktischer *Konsistenztest* eignet: Ein Schnitt X mit maximalen Ereignissen c_i ist genau dann konsistent, wenn $t_x = (C(c_1)[1], C(c_2)[2], \ldots, C(c_n)[n])$ gilt.

Dies ist intuitiv klar, denn hätte ein Agent A_i an der Stelle j einen größeren Eintrag als A_j an "seiner" Stelle j, dann hätte A_i von einem Ereignis erfahren, das in A_j nach dem Schnittzeitpunkt stattgefunden hat. Dies entspricht der Situation bei dem Schnappschuß-Algorithmus, in der ein schwarzer Agent eine rote Basisnachricht erhält.

Zu einem gegebenen Zeitdiagramm sind nicht beliebige Zeitstempel möglich. Betrachte dazu Abbildung 7 als Ausschnitt aus einem Zeitdiagramm. Zeitvektoren $\binom{o}{p}$ mit $o < r$ und $p > s$ sind nicht möglich. Der Begriff der *möglichen Zeitvektoren* läßt sich formalisieren. Damit ergibt sich dann das entscheidende Theorem: Für eine gegebene Zeitstruktur E sind der Verband der konsistenten Schnitte und der Verband der möglichen Zeitvektoren isomorph.

Anwendungen der Vektorzeit liegen u.a. in dem Testen verteilter Systeme (vgl. [14]) und in der verteilten Simulation (vgl. [20]).

Zum Abschluß des Kapitels über die Zeit soll noch kurz auf die Analogien zwischen Vektor-Zeit und Minkowski'scher Raum-Zeit eingegangen werden.

Minkowski führte zur Darstellung der relativistischen Realität sein sog. *Raum-Zeit-Modell* ein, Raum und Zeit werden miteinander verschmolzen. In diesem Modell wird ein n-1-dimensionaler Raum mit der eindimensionalen Zeit zu einem n-dimensionalen Bild der Welt verknüpft. Jedes Ereignis bekommt so seinen "Platz" in der Raum-Zeit.

Die mögliche Beeinflussung von Ereignissen wird durch ihre gegenseitige Raum-Zeit-Lage und die Lichtgeschwindigkeit als maximale Signalausbreitungs-Geschwindigkeit bestimmt.

Dies wird in dem Modell durch die sog. *Lichtkegel* zum Ausdruck gebracht (vgl. Abbildung 8). Ereignisse im *Vorkegel* von P können P beeinflussen, Ereignisse im *Nachkegel* von P können von P beeinflußt werden. Alle Ereignisse außerhalb des Vor- bzw. Nachkegels

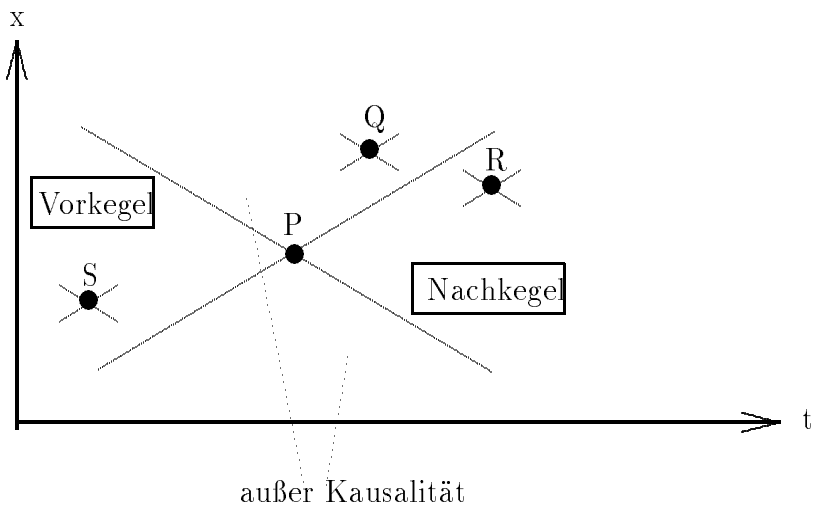

Abbildung 8: Minkowski'sche Raum-Zeit

bilden P's *Gegenwart*, sie geschehen *gleichzeitig* zu P[4]. Die so definierte *Kausalordnung* ist eine Halbordnung.

Das Minkowski'sche Konzept des Lichtkegels eines Ereignisses läßt sich auf Zeitdiagramme übertragen. Definiert man den Vorkegel $VK(e) = \{e' \mid e' < e\}$ und den Nachkegel als $NK(e) = \{e' \mid e < e'\}$, so bilden $VK(e)$ und $E \backslash NK(e)$ konsistente Schnitte. Es gilt $t_{VK(e)} = C(e)$. D. h. der Zeitvektor stellt eine kompakte Repräsentation des Wissens über die Ereignisse, von denen ein Ereignis abhängt, dar.

Soviel zur Zeit. Zum Schluß noch eine Aussage über die Zeit, mit der die meisten Abhandlungen über die Zeit beginnen.

Was also ist die Zeit ?
Wenn niemand mich fragt,
Weiß ich's.
Will ich's aber einem Fragenden erklären,
Weiß ich's nicht.

Augustinus: Bekenntnisse

[4]Beachte, daß diese Gleichzeitigkeit (wie $\|$) nicht transitiv ist!

4 Verwendbarkeit der Methoden für Multi-Agenten-Systeme

Betrachtet man Multi-Agenten-Systeme, so treten auf den ersten Blick sehr ähnliche Probleme wie bei der klassischen Informatik der verteilten Systeme auf. Um ein Problem gemeinsam zu lösen, müssen die Agenten ihre Handlungen koordinieren, sie müssen sich untereinander verständigen. Um zu adäquaten Lösungen zu kommen, müssen die Agenten über einen gemeinsamen Zeitbegriff verfügen. Man könnte nun annehmen, daß die Lösungsmethoden der klassischen Informatik auch in der verteilten KI leicht einsetzbar sind.

Bei genauerem Hinsehen erkennt man jedoch sehr bald große Unterschiede, insbesondere was die angestrebten Ziele und die Bewertungskriterien der erreichten Lösungen angeht. Aufgrund der Verschiedenheit der angestrebten Ziele kommt es oft schon an einem sehr frühen Punkt zu einer *Divergenz der Problemlösungmethoden.*

Dazu ein Beispiel. Eine frühe Lösung für die Koordination der Ressourcenvergabe war der Gebrauch einer Lock-Variablen (vgl. Kapitel 2.2.1). Dabei haben die Agenten selbst "nachgeschaut", ob die Ressource frei geworden ist. Für die Betriebssysteme war diese Lösung nicht adäquat, dort wurde die Konfliktlösung mit Einführung der Semaphortechnik für die Prozesse transparent gemacht. Für MAS'e ist aber i. allg. der Weg, bei dem sich die Agenten selbst um die Konfliktlösung kümmern, der angemessene. So laufen hier schon die Koordinationsmethoden auseinander.

Im folgenden sollen die verschiedenen Zielsetzungen bei den Problemlösungsmethoden etwas genauer analysiert werden.

4.1 Zielsetzungen der Problemlösungsmethoden

Hauptziele bei der Koordination in Betriebssystemen sind *Effizienz* und *Transparenz.* Die Prozesse sehen nur das Ergebnis einer Koordination, nicht die Koordination selbst. Insbesondere sind sie an der Koordination nicht selbst beteiligt. Dies macht Sinn in zweierlei Hinsicht. Zum einen ist eine tatsächliche Parallelität auf Prozessorebene gar nicht möglich, da es i. allg. nur einen Prozessor gibt. Zum anderen sind die Prozesse i. allg. spezifiziert durch Benutzerprogramme. Die Benutzer möchten sich nicht um Ressourcenkonflikte kümmern.

Bei den verteilten Betriebssystemen sind die Voraussetzungen andere. Hier liegt eine Mehrrechnerarchitektur zugrunde. Hauptziel ist die Bereitstellung von *effizienten* und *natürlichen* Koordinations- und Kommunikations-Primitiven. Die Betonung liegt allerdings

wieder auf effizient. Wie bei den Betriebssystemen ist eine möglichst einfache Abbildung auf die zugrunde liegende Maschinenarchitektur von entscheidender Bedeutung. Koordinationsvorgänge sind für Client-Prozesse transparent, allerdings nicht unbedingt für Server-Prozesse.

Hauptmerkmal bei den klassischen Problemlösungen ist ihr *mathematischer Charakter*. Schon eine kleine Inkorrektheit wie der Verlust einer Nachricht kann zu einem "Absturz" des Systems führen.

Hauptziel bei den MAS'en ist die *Kooperation* und die Ausnutzung der *Synergie*. Transparenz ist nicht nur nicht wichtig sondern geradezu unerwünscht. Die Agenten sind autonom und reaktiv. Es ist ihre Aufgabe, ihr *lokales Wissen* in den Problemlösungsprozeß mit einzubringen. Tatsächliche Parallelität ist möglich. Wichtiger als die Adäquatheit einer Methode für ein bestimmtes Maschinenmodell ist ihre *kognitive Adäquatheit*. Lösungen sind häufig von einem weniger mathematischen Charakter. Ein wesentliches Merkmal von MAS'en ist ihre hohe *Stabilität*. Kleine Ungenauigkeiten führen i. allg. "nur" zu suboptimalen Lösungen und nicht zu Abstürzen des Systems.

Insgesamt läßt sich feststellen, daß Methoden, deren Hauptziel die Transparenz ist, sich kaum für den Einsatz in MAS'en eignen, und daß zwischen den nichttransparenten klassischen Methoden und den VKI-Methoden ein großer Abstand – hauptsächlich verursacht durch die Differenz von maschineller und kognitiver Adäquatheit – besteht. In Anbetracht der Tatsache, daß letztendlich aber auch die Modelle der VKI auf Rechnermodelle abgebildet werden müssen, erscheint eine Untersuchung der klassischen Methoden sinnvoll.

Im folgenden sollen (sehr rudimentär) Möglichkeiten zur Integration klassischer Methoden erörtert werden.

4.2 Integration klassischer Modelle

Ein möglicher Ansatz zur Integration ist die Benutzung der in Kapitel 2.2.2 vorgestellten Basisalgorithmen zur Lösung von Problemen in MAS-Szenarien. Als Beispiel diene hier der Election-Algorithmus auf der einen Seite und die Aufgabe der Task-Decomposition (TD) und Task-Allocation (TA) (vgl. [15]) auf der anderen Seite.

Bei der TD geht es darum, eine Aufgabe in mehrere Teilaufgaben aufzuteilen, die dann mittels der TA an mehrere Agenten verteilt werden.

Eine Lösungsmöglichkeit besteht darin, durch eine *vorgeschaltete Election-Phase* einen Manager auszuwählen, der dann *zentral* die TD und TA vornimmt. Der Manager wird damit dynamisch bestimmt. Man vermeidet so einige der bekannten Nachteile zentraler Stellen in verteilten Systemen. Allgemeiner können klassische Basisalgorithmen als *Unterprozeduren*

von höheren Lösungsmethoden dienen.

Zum anderen kann man die Struktur der Basisalgorithmen als Grundstruktur nehmen und diese zu einer Lösung für höhere Probleme erweitern. So kann man einen Election-Algorithmus sicherlich dahingehend erweitern, daß er zur Lösung der TA dienen kann. Um dabei zu einer adäquaten Lösung zu kommen, bei der die Fähigkeiten der einzelnen Agenten berücksichtigt werden, wird man viel Aufwand in die Maximalitätsfunktion stecken müssen. Bei dieser Vorgehensweise wird dafür sichergestellt, daß das abstrakte Basisproblem durch die hohe KI-Sicht nicht übersehen wird.

Ein weiterer Aspekt der Integration ergibt sich aus der Sicht von großen Systemen als *geschichtete Systeme*. So sehen Brachman und Levesque (vgl. [7]) ein wissensbasiertes System als ein in die Schichten *implementational level, computational level, epistemic level, knowledge level und natural language level* unterteiltes System.

Wesentlich vereinfacht kann man in einem System zwischen einer *technischen* und einer *konzeptionellen Ebene* unterscheiden. Viele der klassischen Verfahren kann man in der klassischen Ebene "verbergen" bzw. zur Bereitstellung adäquater Methoden auf der konzeptionellen Ebene benutzen.

So kann man bei der Kommunikation das Port-Konzept auf der technischen Ebene dazu verwenden, Nachrichten-Klassen und -Hierarchien wie Kontroll-Nachrichten, normale Nachrichten etc. auf der konzeptionellen Ebene bereitzustellen. Kommunikation zwischen Agenten kann auf der technischen Ebene mittels Blackboard-Systemen realisiert werden, auf der konzeptionellen Ebene wäre dies aber i. allg. ein unangemessenes Modell. Zur Modellierung von in der Realität zentralen Stellen wie Börsen ist das Blackboard-Modell auch auf der konzeptionellen Ebene geeignet, vergleiche etwa die Auftrags-Börse bei dem Speditions-Szenario.

Als letzter Punkt der Integration soll nun noch die Integration von Zeitmodellen angedacht werden. Um Lösungsmethoden etwa des Speditions-Szenarios an menschliche Vorgehensweisen anzulehnen, ist ein Zeitmodell unverzichtbar. Vorbild könnte hier die *verteilte Simulation* sein. Nach [19] läßt sich ein verteiltes Simulationssystem als ein System miteinander kommunizierender sequentieller ereignisgesteuerter Simulatoren auffassen, wobei jeder Simulator mit einer eigenen logischen Uhr ausgestattet ist und sich die Simulatoren über Nachrichten gegenseitig Ereignisse einplanen können. Die logischen Uhren dürfen jedoch nicht völlig asynchron laufen, damit ein Simulator kein Ereignis von einem anderen Prozeß eingeplant bekommt, das für ihn in der Vergangenheit liegt (vgl. auch [20]). Analoge Probleme treten beim Speditions-Szenario auf. Aufträge für die LKW's bzw. die Speditionen müssen gegenseitig so eingeplant werden, daß sie innerhalb der zeitlichen Vorgaben erfüllt werden können. Eine Symbiose der beiden Gebiete erscheint hier sehr vielverspre-

chend.

Allerdings lassen die hier vorgestellten Zeitmodelle nur sehr einfache zeitliche Schlüsse zu. Es können nur Zeitpunkte auf ihre zeitliche Reihenfolge hin verglichen werden. Wahrscheinlich werden die Probleme beim Speditions-Szenario wesentlich komplexere Schlüsse etwa im Sinne der Allen'schen Zeitlogik (vgl. [1]) erfordern. Dies muß eine genaue Analyse noch klären.

5 Fazit

Die klassischen Verfahren zur Koordination in Betriebssystemen schließen die "Intelligenz" der Agenten nicht mit ein, die Koordination ist für die Agenten unsichtbar. Sie sind daher für den Einsatz in MAS'en nur sehr bedingt einsetzbar.

Verfahren der verteilten Betriebssysteme und der Verteilten Programmiersprachen gehen zumindest prinzipiell von autonomen Agenten aus und nutzen lokales Wissen zur Problemlösung. Sie können daher schon eher als Grundlage für Methoden der MAS'e dienen, müssen aber für die speziellen Bedürfnisse der VKI erweitert werden.

Insgesamt zeigt die Analyse der klassischen Methoden, daß bereits diese eine sehr hohe Komplexität aufweisen, und daß eine Vorgehensweise in der VKI, die sich auf Teilbereiche bzw. -Probleme konzentriert und einfache Beispiel-Szenarien untersucht, als sinnvoll anzusehen ist.

Literatur

[1] J. F. Allen, *Towards a General Theory of Action and Time*, AI 23, 1984, pp 123–154.

[2] G. R. Andrews, F. B. Schneider, *Concepts and Notations for Concurrent Programming*, Computing Surveys, Vol. 15, No. 1, March 1983, pp 3–43.

[3] H. Bal, J. Steiner, A. Tanenbaum, *Programming Languages for Distributed Computing Systems*, ACM Computing Surveys, Vol. 21, No. 3, September 1989, pp 261–322.

[4] R. Bayer, K. Elhardt, W. Kiessling, D. Killar, *Verteilte Datenbanksysteme*, Informatik-Spektrum, 7, 1984, pp 1–19.

[5] C. Beilken, F. Mattern, *Verteiltes Problemlösen am Beispiel von Zahlenrätseln – Ein Experiment mit CSSA*, Interner Bericht 29–86, Universität Kaiserslautern, 1986.

[6] C. Beilken, F. Mattern, M. Spenke, *Entwurf und Implementierung von CSSA, Teil A: Konzepte*, Interner Bericht 67-83, Universität Kaiserslautern, 1983.

[7] R. J. Brachman, H. J. Levesque, *Readings in Knowledge Representation*, Morgan Kaufman, 1985.

[8] H. J. Bürckert, J. Müller, A. Schupeta, *RATMAN and its Relation to Other Multi–Agent Testbeds*, Research Report RR–91–09, DFKI 1991.

[9] P. Dadam, *Synchronisation in verteilten Datenbanksystemen: Ein Überblick*, Informatik-Spektrum, 4, 1981, pp 175–184, 261–270.

[10] W. Dupre, *Distributed Systems Software Technology Publications Overview*, IBM T. J. Watson Research Center, 1991.

[11] R. Engelmore, T. Morgan *Blackboard Systems*, Addison-Wesley, 1988.

[12] Kurt Gödel, *Einige Bemerkungen über die Beziehungen zwischen der Relativitätstheorie und der idealistischen Philosophie*, in: Albert Einstein als Philosoph und Naturforscher (Hg. P. A. Schlipp), Vieweg Verlag 1979.

[13] J. Gulbin, *UNIX*, Springer Verlag, 1988.

[14] D. Haban *The Distributed Test Methology DTM*, Dissertation, Universität Kaiserslautern 1988.

[15] N. Kuhn and H. J. Müller and J. P. Müller, *Task Decomposition in Dynamic Agent Societies*, eingereicht zur ISADS-92, Tokio, 1992.

[16] L. Lamport *Time, Clocks, and the Ordering of Events in a Distributed System*, CACM, Vol. 21, No. 7, July 1978, pp 558–565.

[17] A. Lister *The Problem of Nested Monitor Calls*, Oper. Sys. Rev., Vol. 11, No. 3, July 1977, pp 5–7.

[18] F. v. Martial, *Coordinating plans of Autonomous Agents*, Dissertation, Universität Saarbrücken, 1992.

[19] F. Mattern, *Verteilte Basisalgorithmen*, Informatik Fachberichte 226, Springer Verlag 1987.

[20] F. Mattern, H. Mehl, *Diskrete Simulation – Prinzipien und Probleme der Effizienzsteigerung durch Parallelisierung*, Informatik Spektrum 12, 1989.

[21] C. Morgan, *Global and Logical Time in Distributed Algorithms*, Information Processing Letters 20, May 1985, pp 189–194.

[22] J. Nehmer, *Softwaretechnik für verteilte Systeme*, Springer Verlag, 1985.

[23] J. Nehmer, *Verteilte Betriebssysteme*, Skriptum zur Vorlesung, Fachbereich Informatik, Universität Kaiserslautern, 1989.

[24] J. Siekmann, H. J. Bürckert, J. Müller, H. J. Ohlbach, *Project Description AKA–Mod*, DFKI 1991.

[25] A. S. Tanenbaum, R. van Renesse, *Distributed Operating Systems*, ACM Computing Surveys, Vol. 17, No. 4, Dec. 1985, pp 419–470.