

## 2 Architektur verteilter Programme

2.1 Kommunikationsoperationen	2
2.2 Klassifikation von Architekturen	8
2.3 Datenfluss-Architektur	13
2.4 Verteilte Algorithmen	21
2.5 Dienst-Architektur	26
2.6 Ereignisbasierte Systeme	30
2.7 Blackboard-Architekturen	34
2.8 Stilbrüche	37
Zusammenfassung	40

## 2.1 Kommunikationsoperationen

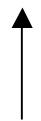
**Kommunikationsoperationen** (*communication primitives*)  
dienen dem Nachrichtenaustausch zwischen Prozessen:

**send:** Nachricht versenden (ausgeben, produzieren, ...).

Die Nachrichtenquelle heißt **Sender** (*sender*)  
oder **Produzent** (*producer*).

**recv:** Nachricht empfangen (einlesen, verbrauchen, ...).

Die Nachrichtensenke heißt **Empfänger** (*receiver*)  
oder **Verbraucher** (*consumer*).



Schlüsselwörter oder Prozedurnamen

## Parametrisierung und präzise Semantik variieren entlang verschiedener Dimensionen:

- *Identifizierung der Partner:*  
direkt/indirekt, einzeln/gruppiert
- *Pufferung der Nachrichten:*  
beschränkt/unbeschränkt gepuffert, ungepuffert
- *Selektiver Nachrichtenempfang:*  
abhängig/unabhängig von Art der Nachricht
- *Typisierung der Nachrichten:*  
getypt/ungetypt

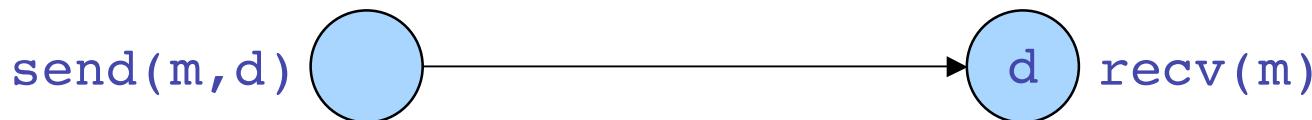
## Pufferung

- *ungepuffert*: Sender und Empfänger kooperieren zwecks Übergabe einer Nachricht, müssen also gegebenenfalls auf den anderen warten („handshake“, „rendezvous“).  
→ *synchrone* Übertragung, *blockierende* Operationen.
- *beschränkt gepuffert*: der Sender kann dem Empfänger um  $n$  Nachrichten ( $1 \leq n \leq N$ ) vorausseilen.  
→ *asynchrone* Übertragung, *blockierende* Operationen.
- *unbeschränkt gepuffert*: der Sender kann dem Empfänger beliebig weit vorausseilen.  
→ *asynchrone* Übertragung, *nichtblockierendes send*.

(Beachte: `recv` ist in jedem Fall *blockierend*)

## Identifizierung des Partners:

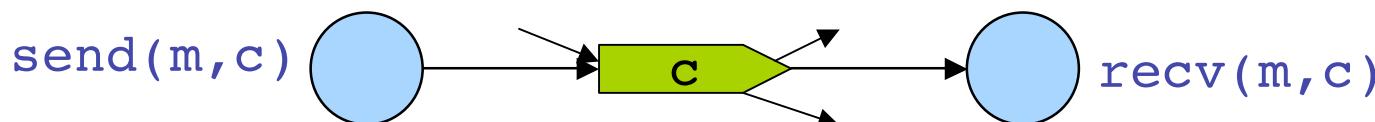
- direkt: Identifizierung eines Prozesses



- indirekt: Identifizierung von *prozesslokalen Ports* (*port*)



- indirekt: Identifizierung eines *Kanals* (*channel*)



+ Mischformen

## Implementierung in verschiedenen Varianten:

- Laufzeitsystem einer nichtsequentiellen Sprache
- lokale Interprozesskommunikation
- Internet-Transportdienst
- ... weitere

**Beispiel Erlang:** funktionale Sprache, dynamisch getypt,  
für verteilte Programme/Systeme

*Senden:* `Consumer ! Message`  
sendet nichtblockierend den Wert der  
Variablen `Message` an den Prozess `Consumer`.  
und liefert diesen Wert.

*Empfangen:* `receive Message -> expression end`  
liest Nachricht - sobald vorhanden - aus dem  
(einzigem) Eingabepuffer des aktiven Prozesses  
in die Variable `Message` und liefert den Wert  
des Ausdrucks `expression`.

Selektiver Nachrichtenempfang möglich.

## 2.2 Klassifikation von Architekturen

**Architektur eines Softwaresystems** =

Komponenten des Systems und ihre Interaktion,  
z.B. aufrufbasierte OSI-Schichtenarchitektur

**Architekturstil und Interaktionsstil** =

Art der Komponenten und ihrer Interaktion,  
z.B. objektorientierter Architekturstil mit Klassen,  
Vererbung, Objekten und Methodenaufrufen

## Architekturstil verteilter Programme?

Externe Komponenten: kommunizierende Prozesse

Interne Struktur: aufgerufene Prozeduren, Funktionen, Methoden

Genauer:

Klassifikation verteilter Architekturen orientiert sich daran, was die beteiligten Prozesse über ihre Partner wissen (müssen).

Partner spielen bestimmte Rollen.

Achtung: Nicht verwechseln mit „Architektur verteilter Systeme“ (1.2)

- ① **Datenfluss-Architektur** (*flow-based architecture*):  
Prozesse arbeiten als **Filter**, d.h. sie wandeln Eingabeströme von Eingabe-Ports in Ausgabeströme auf Ausgabe-Ports um.  
Sie wissen *nichts* über Herkunft und Ziel der Daten!
  
- ② **Verteilte Algorithmen** (*distributed algorithms*):  
Gleichberechtigte **Partner** (*peers*) kooperieren zwecks Erreichung eines gemeinsamen Ziels. Jeder Partner weiß über - und verlässt sich auf - ein *bestimmtes* Verhalten seiner Partner!

- ③ **Dienst-Architektur** (*client/server architecture*):  
**Dienstnehmer** (*Klienten, clients*) senden **Aufträge** an  
**Dienstgeber** (*servers*) und erhalten nach Auftragserledigung  
eine Antwort. Ein Klient erwartet vom Anbieter eine bestimmte  
Funktionalität; dieser weiß aber *nichts* vom Klienten!
  
- ④ **Ereignisbasiertes System** (*event-based system*):  
**Abonnenten** (*subscribers*) sind an bestimmten **Ereignissen** (*events*)  
interessiert, über deren Eintreten sie von den auslösenden  
**Ereignisquellen** (*publishers*) benachrichtigt werden.  
Umgekehrt wissen die Quellen aber *nichts* von der Funktionalität  
der Abonnenten!

„Prozess p kennt Prozess q“ bedeutet im folgenden, dass

- p die Identität von q kennt,
- p die Spezifikation des Programms Q von q kennt,

genauer:

- p ist im Besitz eines Verweises auf q
- Die Korrektheit des Programms P von p hängt von der Korrektheit des Programms Q von q ab.

## 2.3 Datenfluss-Architektur (*pipes and filters*)

Prozess arbeitet als **Filter**, d.h.

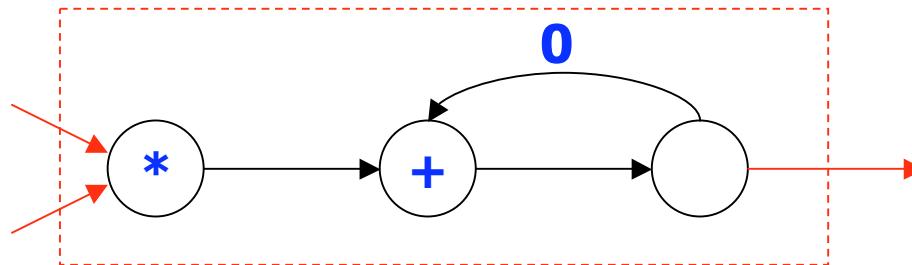
- liest Datenfolgen über Eingabe-Kanäle,
- berechnet daraus neue Daten,
- sendet Datenfolgen über Ausgabe-Kanäle,
- kennt seine Partner nicht.

- **Kanal** (*channel, pipe*) verknüpft *unidirektional* Sender mit Empfängern (vgl. S. 5 unten); häufig genau *ein* Sender und Empfänger.
- **Motivation:** Parallelverarbeitung oder/und natürliche Problemlösung

## Graphische Darstellung:

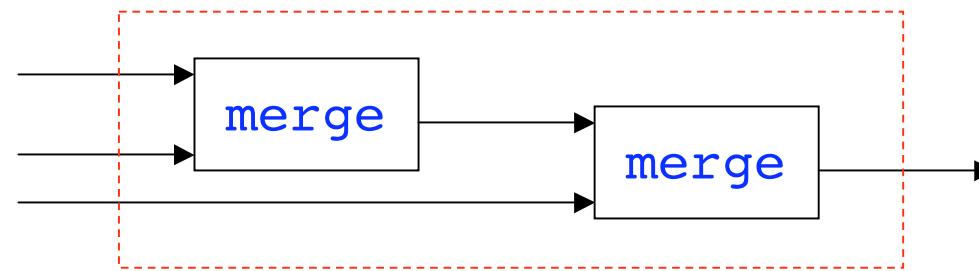
- gerichteter Graph
- mit Prozessen als Ecken,
- und Kanälen als Kanten (falls nur 1 Sender/Empfänger sonst paariger Graph!))

### Beispiel 1:

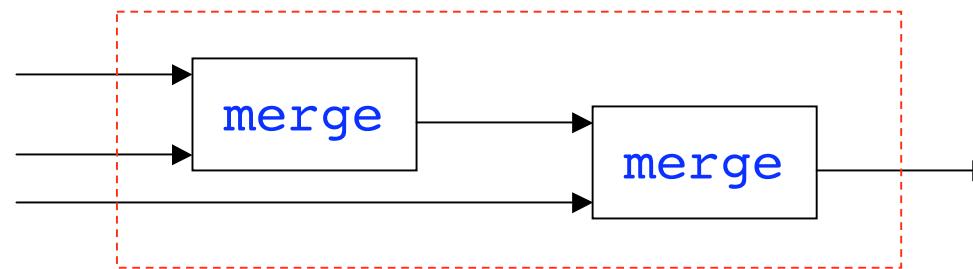


berechnet die Partialsummen eines Skalarprodukts  
(in Hardware: „Datenflussmaschine“)

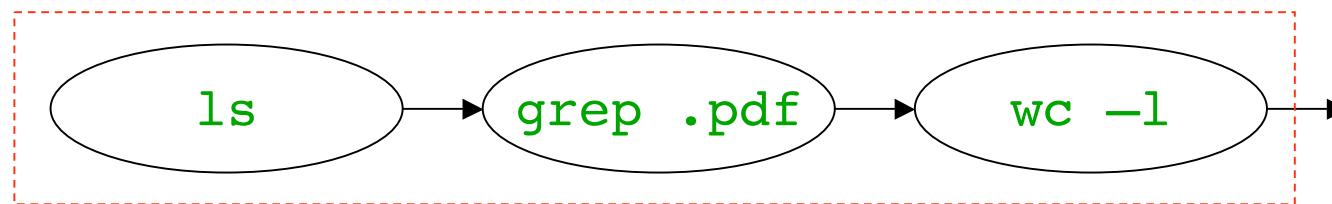
Beispiel 2: Verschmelzen sortierter Zahlenfolgen:



Beispiel 2: Verschmelzen sortierter Zahlenfolgen:



Beispiel 3: Ermittle Anzahl der PDF-Dateien im aktuellen Verzeichnis mittels Unix-Dienstprogrammen:



*Vereinbarung von Ports als formale Parameter, z.B.*

```
merge(in in1, in in2, out out1) {...}  
                                (hier ungetypt)
```

*Kommunikationsoperationen z.B.*

```
send(message, out1)  
recv(mesvar, in1)  
                                (hier ungetypt)
```

*Pufferung: beschränkt oder unbeschränkt*

*Typisierung: Nachrichten wie Ports wie Kanäle !*

*Programmkonstruktion mit Koordinationssprache*, z.B.

```
merge3(in a, in b, in c, out d):  
    channel x; // ref to new channel  
    merge(a, b, x);  
    merge(c, x, d).
```

oder

```
ls | grep .pdf | wc -l
```

*Bemerkung:* Diese beiden Programme sind rein *deklarativ*!

## Programmiersprache versus Koordinationssprache:

- Ports werden durch Kanäle aktualisiert
- Koordinationssprache ist *deklarativ*
- Koordinationssprache *unabhängig* von der Programmiersprache
- Programmiersprache muss Koordinationssprache *unterstützen*
- *Heterogene Programme*: verschiedene Programmiersprachen
- *Komponentenbasierte Software-Entwicklung!*

Beachte: In der Praxis werden Datenflusssysteme mit den unterschiedlichsten Techniken aufgesetzt -

- *imperativ* mit expliziter Kanalerzeugung/lösung
- unabhängige Lebensdauer der Partner
- Unix *Named Pipes* als Kanäle
- TCP-Verbindungen als Kanäle
- .....

## 2.4 Verteilte Algorithmen

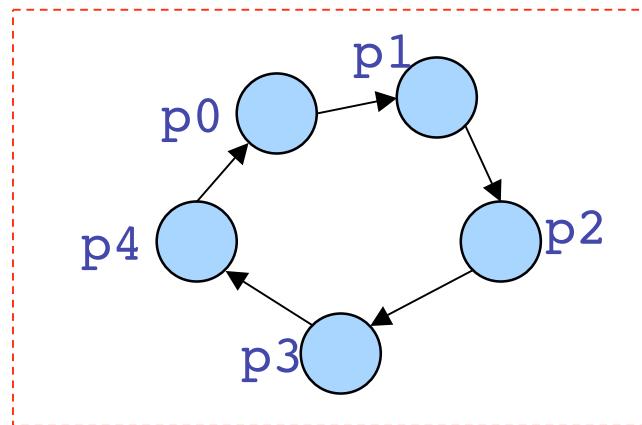
Prozesse kennen gewisse Partner und deren Verhalten und kooperieren zwecks gemeinsamer Problemlösung

- Direkte Identifizierung des Empfängers (vgl. S. 5 oben)
- ... und eines dortigen Ports (bzw. einzige „Mailbox“).
- Partner sind i.d.R. gleichberechtigt,
- ... oft auch mit identischem Code.
  
- *Motivation:* Organisationsprobleme der Netz-Infrastruktur,  
Parallelverarbeitung,  
Peer-to-Peer-Systeme

## Graphische Darstellung:

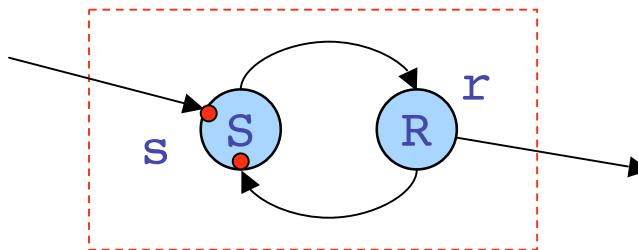
- gerichteter Graph
- mit Prozessen als Ecken;
- Kante von a nach b: a kennt b

Beispiel 1:



z.B. für Rundmeldung mit Quittung

## Beispiel 2: mit Eingabereports als Nachrichtenziele



Zweck sei die sichere Nachrichtenübertragung von **s** nach **r** unter Verwendung zweier unsicherer Kanäle, auf denen sporadisch Nachrichten verloren gehen.

Lösungsidee: Nachrichten quittieren - neue Nachricht erst dann schicken, wenn Quittung eingetroffen; evtl. letzte Nachricht wiederholen.

→ „Alternating Bit Protocol“

*Partner-Prozesse als formale Parameter,  
Ports als Attribute von Prozessen, z.B.*

**R** kennt **S** -  
und umgekehrt!

**R(S s, C c) exports mailbox { . . . }**  
**S(R r) exports port0, port1 { . . . }**

(hier ungetypt)

*Kommunikationsoperationen bezogen auf Ports, z.B.*

**send(message, s.port0)**  
**recv(mesvar, port0)**

(hier ungetypt)

*Pufferung: beschränkt oder unbeschränkt*

*Typisierung: Ports wie Nachrichten !*

*Programmkonstruktion mit passender Koordinationssprache, z.B.*

```
AB(C c) exports mailbox = s.port1 :  
    R(s, c) r;  
    S(r) s.
```

*Beachte:* In der Praxis werden verteilte Algorithmen mit den unterschiedlichsten Techniken aufgesetzt:

- *imperative* statt deklarative Programmiersprache
- explizite Verbindungsherstellung
- unsichere Kommunikation übers Netz
- TCP-Verbindungen
- .....

## 2.5 Dienst-Architektur

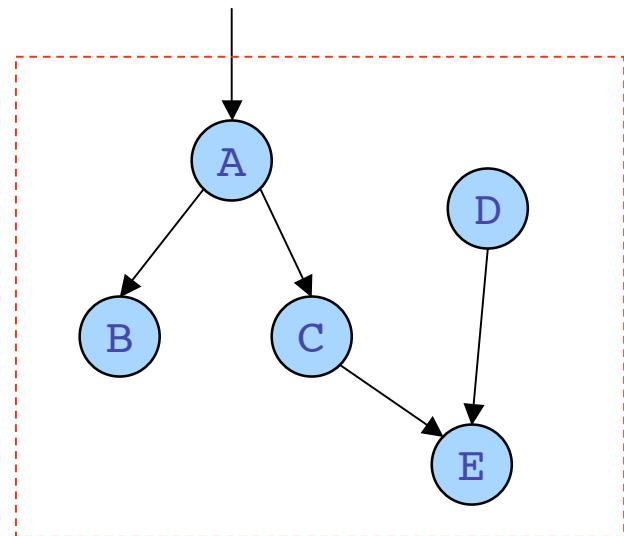
Prozess spielt Rolle als  
**Dienstgeber** (Dienstanbieter, *server*) oder  
**Dienstnehmer** (Klient, *client*) oder beides

- Klient kennt Dienstanbieter und dessen Dienst (*service*)
- Server kennt seine Klienten *nicht*
- Auftrags-Nachricht (*request*) an den Server enthält Argumente
- Antwort-Nachricht (*reply*) an den Klienten enthält Ergebnisse
- *Nichtsequentielles Analogon zu Modul/Objekt-Aufruf!*
- *Motivation:*  
Systemdienste jenseits der Betriebssystem-Dienste,  
entfernte Dienste, dienstorientierte Architekturen (*SOA*)

## Graphische Darstellung:

- zyklenfreier gerichteter Graph
- mit Prozessen als Ecken;
- Kante von a nach b: a kennt b als Server

Beispiel:



*Server-Prozesse als formale Parameter, z.B.*

```
A(B b, C c) { . . . . . }  
C(E e) { . . . . . }
```

*4 (!) Kommunikationsoperationen, z.B.*

Klient:    Port p = sendarg(arg, server);  
              recvres(resvar, p);

Server:    Port p = recvarg(argvar); ...  
              sendres(res, p);

**p** ist temporärer Eingabe-Port des Klienten für die Antwort  
(hier ungetypt)

*Pufferung:* bei **p** einfach,  
beim Server beschränkt oder unbeschränkt

*Programmkonstruktion mit passender Koordinationssprache*  
(häufig mit Programmiersprache identisch), z.B.

```
SuperServer() exports a:  
    A(b,c) a;  
    B() b;  
    C(e) c;  
    D(e) d;  
    E() e.
```

*Beachte:* viele Varianten!

- *dynamisches Auffinden passender Dienste*
- *unabhängige Lebensdauern der Beteiligten*
- *mehrere Eingabe-Ports*
- *explizite Herstellung von TCP-Verbindungen*
- *längerer Dialog zwischen Klient und Server*

## 2.6 Ereignisbasierte Systeme

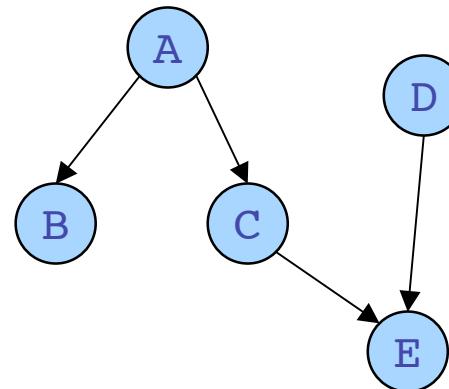
Ereignisquelle benachrichtigt Interessenten über „Ereignisse“

- Interessent (*subscriber*) kennt die Quelle (*publisher*)
- Quelle kennt die Interessenten *nicht*
- Interessent kann zugleich wieder Quelle sein
- Aber: Registrierung („Abonnement“) bei Quelle erforderlich
- Nichtsequentielles Analogon zum Beobachter-Muster
- unabhängig vom Ereignis-Begriff: „Gruppenkommunikation“
  
- Motivation: Informationsdienste im Netz

## Graphische Darstellung:

- zyklenfreier gerichteter Graph
- mit Prozessen als Ecken;
- Kante von a nach b: b kennt a (!), a benachrichtigt b

Beispiel:



*Ereignisquellen* als formale Parameter, z.B.

A( ) { . . . . . }  
C(A a) { . . . . . }  
E(C c, D d) { . . . . . }

*Kommunikationsoperationen*, z.B.

send(message)  
recv(mesvar, c)

(hier ungetypt)

*Pufferung*: beschränkt oder unbeschränkt

## Programmkonstruktion mit passender Koordinationssprache, z.B.

```
EventSystem() :  
    A() a;  
    B(a) b;  
    C(a) c;  
    D() d;  
    E(c,d) e.
```

Beachte: viele Varianten!

- typisch ist *dynamische Registrierung/Kündigung*
- .... bei explizit vereinbartem *Ereigniskanal* -  
Quelle bleibt unsichtbar (*publish-subscribe pattern*)
- *Filterung* von Nachrichten gemäß Empfänger-Interessen
- *umfangreiche Middleware*

## 2.7 Blackboard-Architekturen

Prozesse kommunizieren indirekt über einen globalen  
**Nachrichten-Pool („blackboard“),**  
wo jeder Nachrichten ablegen oder entnehmen kann

- jeder Partner kennt den Pool
- selektive Nachrichtenentnahme
- Konventionen über Form und Inhalt der Nachrichten!
- kooperative Partner (vgl. verteilte Algorithmen)
- niedriges Abstraktionsniveau, aber sehr flexibel
- **Motivation:** Experimente mit zeitlich/räumlich schwach gekoppelten Komponenten

Populäres Beispiel: *Linda mit Tuple Space*:

Kommunikationsoperationen z.B.

Senden: `out("fromBob", "toEve", msg)`

deponiert dieses Tripel im Tupelraum  
(nichtblockierend)

Empfangen: `in(?sender, "toEve", ?message)`

entnimmt passendes Tripel mit  
Musteranpassung (blockierend)

... und weitere

Pool-Kapazität: unbeschränkt

Typisierung: i.d.R. dynamisch

*Programmkonstruktion ist trivial, z.B.*

*Example : A; B; C; A; D.*

*Verallgemeinerbar: mehrere Tupelräume, auch repliziert*

*Attraktiv: andere Architekturstile leicht simulierbar !*

*Problem: mangelnde Effizienz*

*Bemerkung:* der Begriff *Koordinationssprache* wurde von den Linda-Autoren geprägt und bezeichnet im Linda-Kontext die Gesamtheit der Tupelraum-Operationen.

## 2.8 Stilbrüche

Problem:

Gegeben: Plattform/Sprache für Architekturstil A

Gewünscht: Architektur mit Stil B

Was tun?

Beispiel: Gegeben: Plattform für Datenfluss

Gewünscht: Dienst-Architektur

→ Versuch der Nachbildung einer Dienst-Plattform  
unter Zweckentfremdung der Datenfluss-Plattform

```
global:    channel requests;
```

Klient:

```
channel replies;
var request, reply;
send((request, replies), requests);
recv(reply, replies);
```

Server:

```
channel replies;
var request, reply;
rep recv((request, replies), requests);
    ... // service
    send(reply, replies);
per;
```

Unschön!

## Übungsfragen:

1. Wird damit genau die gewünschte Dienst-Semantik gemäß 2.5 (S. 27) erzielt?
  
2. Wie verhält sich dieser Ansatz zu dem beim [Allocator](#) aus ALP 4, Kap. „Message Passing“?  
Vorteile, Nachteile?

# Zusammenfassung

- *Kommunikationsoperationen:* viele Varianten
- *Rollen von kommunizierenden Prozessen*  
    → verschiedene Architekturstile
- Sprachen/Plattformen in der Regel zugeschnitten  
    auf bestimmten Architekturstil
- Koordinationssprache versus Programmiersprache

# Quellen

Erlang: [www.erlang.org/doc.html](http://www.erlang.org/doc.html)

Datenfluss-HW: [de.wikipedia.org/wiki/Datenflussarchitektur](http://de.wikipedia.org/wiki/Datenflussarchitektur)

Named Pipes: [en.wikipedia.org/wiki/Named\\_pipe](http://en.wikipedia.org/wiki/Named_pipe)

AB-Protokoll: [en.wikipedia.org/wiki/Alternating\\_bit\\_protocol](http://en.wikipedia.org/wiki/Alternating_bit_protocol)

Linda: [en.wikipedia.org/wiki/Linda\\_\(coordination\\_language\)](http://en.wikipedia.org/wiki/Linda_(coordination_language))