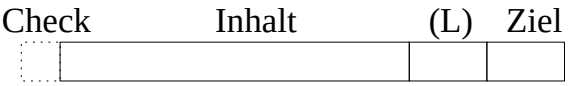
2 Routing

- Im Netzwerk
  - Wege auswählen
  - Pakete versus “Rohrleitung“
  - Pufferstrategien
- Ende-zu-Ende Protokolle
  - Pufferung
  - Synchronisation
  - Zuverlässigkeit
  - Reihenfolgetreue
  - Fairness
- Protokollstapel
- Beispiele

2.1 Router-(Hardware)

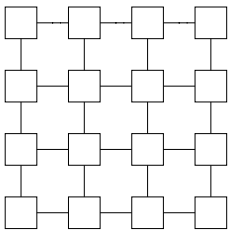
2.1.1 Netzwerkpakete



- Zielknoten (z.B. x, y bei Gitter) $\neq$ Prozessor, Prozeß, ... Wichtig für dynamic routing / packet switching
- Länge (falls nicht fest)
- Inhalt (incl. Headerinformation höherer Schichten)
- ggf. Testbits für Fehlererkennung/Korrektur
- Bei shared memory:
  - Cache line fester Länge
  - Addresse

2.1.2 Wege Auswählen

x-y-Routing für Gitter.



1. Innerhalb einer Zeile zur richtigen Spalte
2. Innerhalb der Spalte

\+ Einfach, Schnell

– Kann zu unnötigen Stauungen führen

Fortgeschrittene Verfahren: adaptiv

\+ Höherer Durchsatz

– Kompliziert

– Reihenfolgetreue gefährdet

2.1.3 Store-and-Forward

- Ganzes Paket über die Leitung schicken bevor Weiterleitung beginnt

\+ Zuverlässig und gut analysierbar. Zum Beispiel ist x-y-Routing *im Mittel* annähernd optimal. $\leadsto$ effizienter randomisierter Algorithmus durch Umleitung über zufällige Zwischenstation.

– Hohe *Latenz* bei langen Paketen

– Benötigt viel Pufferspeicher

2.1.4 Cut-Through Routing

- Weiterleiten sobald nächstes Ziel erkannt
- \+ Sehr geringe Latenz bei niedriger Netzlast
- \+ Puffer sind nicht unbedingt notwendig. (*Wormhole-Routing* = Keine Puffer)
- – Nachrichten können sich leichter behindern
- – Schwer analysierbar
- Deadlockvermeidung wichtig
  - x-y-Routing bei Gittern
  - Zusätzliche Maßnahmen bei Tori („virtuelle Kanäle“)

2.2 Beispiele

2.2.1 IBM „High Performance Switch“

- Mehrstufiges Netzwerk aus 8×8-Kreuzschienenverteilern
- Umsetzung 8-bit *flits* 64-bit Pufferung
- Cut-Through-Routing mit begrenzter Pufferung
- 2.4μs Netzwerklatenz bei 512 PEs
- Mikrokanalkarte mit eigenem Kommunikationsprozessor

2.2.2 Dolphin SCI

- Wormhole-Routing auf unidirektionalen Ringen
- Switch kann mehrere Ringe über einen Bus koppeln (Store-and-Forward von Cache Lines)
- Adapterkarte für PCI Bus

2.2.3 Cray T3E

- 3D-Torus, 600MB/s Kanalbandbreite
- Cut-Through-Routing (Pufferung? Adaptiv?)
- Proprietäre Anbindung

2.3 Ende-zu-Ende-Protokolle

2.3.1 Schnittstelle zur Hardware

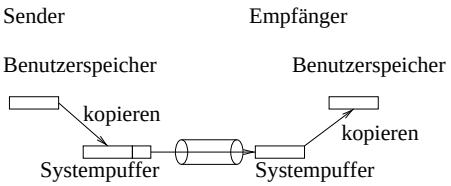
Typische Qualitätseigenschaften:

- Zuverlässig: Keine Paketverluste, Keine Verfälschungen ($\neq$ Telematik). **Wichtig für viele parallele Algorithmen**
- Reihenfolgetreu: Pakete von A nach B werden in der Absendereihenfolge ausgeliefert
- Fair: Kein Paket wird unbegrenzt lange von anderen Paketen aufgehalten.

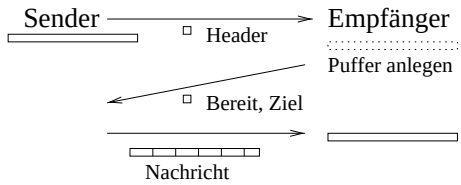
Das heißt **nicht**, daß höhere Schichten dieselben Zusicherungen bekommen.

2.3.2 Pufferverwaltung

Kurze Nachrichten: Einwegeprotokoll



Lange Nachrichten: Dreiwegeprotokoll



+ Weniger Kopieroperationen
+ Universell
– Dreifache Anzahl Nachrichten
– Empfänger muss reagieren

2.3.3 Kanäle

Ein schnelles Zweiwegeprotokoll

- Genau ein Sender-Empfänger-Paar
- Vereinbarte Puffergröße
- Bestätigungen
- Sender muss nicht auf Bestätigungen warten wenn genug Puffer übrig (Sliding Window)
- Bei bidirektionelem Verkehr können Bestätigungen an Datenpakete gekoppelt werden (Piggibacking)

– Kann nicht direkt eingesetzt werden wenn unbekannt ist woher empfangen werden soll

2.4 Wer handhabt die Protokolle

- Hardware
  + Schnell
  + Anwendungsprogramme werden weniger aufgehalten
  – Unflexibel
  – Oft wenig Unterstützung
- Interrupt
  + Flexibel
  + Reagiert sofort
  – Muss mit Betriebssytem zusammenpassen
  – Langsam, vor allem bei heute üblichen Betriebssystemen und Prozessoren
- Polling
  + Einfach
  + Geringer Overhead
  – Unflexibel

- Legt Prozessor lahm (Multithreading schwierig)

2.5 Beispiele

2.5.1 IBM SP

- Software auf Anwendungsprozessor (und Adapterkarte)
- 2 `memcpy` + 1 DMA + Adapteraktivität
- Eine Kopieroperation weniger für lange Nachrichten
- Polling (Interrupts als langsame Notlösung)
- ca. 40μs Latenz

2.5.2 Scali MPI

- (Uncached) Shared Memory wird benutzt um Kanäle zu implementieren.
- Kanäle werden direkter genutzt
- 1 `memcopy` + 1 DMA
- 3 Protokollvarianten für kurze, mittlere und lange Nachrichten
- Schnelleres Protokoll innerhalb eines SMP-Knotens
- Kurze Nachrichten werden direkt übertragen
- ca. 10μs Latenz

2.5.3 Cray T3E

- Hardwareunterstützung für kurze Nachrichten
- 2.7μs Ende-zu-Ende Latenz! (leider nicht für MPI)
- (nicht Cache kohärentes) Shared-Memory + Barrier in Hardware

2.6 Ein Schichtenmodell a la ISO/OSI

1. Physikalisch. Drähte, Glasfasern, Signalformen, Clocks
2. Verbindungsschicht (Data-Link-Layer). Kommunikation zwischen zwei Routern
3. Netzwerkschicht (Network-Layer). Verbindungsnetzwerk incl. Routing
4. Transportschicht. Fast der Service den MPI bietet. Nachrichten über das Netzwerk übertragen. (Zerstückeln, puffern, ...)
5. Sitzungsschicht. Leer?
6. Präsentationsschicht. Ggf. Konversion zwischen inkompatiblen Datenrepräsentationen
7. Anwendungsschicht. Weitergehende Protokolle, z.B. kollektive Operationen

3 Gehversuche mit MPI

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char** argv)
{
   MPI_Init(&argc, &argv);

   printf("Hello world!\n");

   MPI_Finalize();
}
```

- `g++ hello.c -o hello -lmpi`
- `mpirun -np 4 a.out`

3.1 Kommunikatoren

```
int main(int argc, char** argv)
{
   int myId, numProcs;

   MPI_Init(&argc, &argv);

   MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myId);

   printf("I am PE %d out of %d PEs.\n",
          myId, numProcs);

   MPI_Finalize();
}
```

- Numerierte Gruppe von Prozessoren
- Kontext

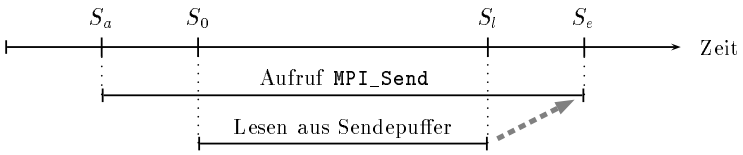
3.2 Ping-Pong

```
#define PING 42
#define PONG 17
if (myId == 0) {
   for (i = 0;  i < tries;  i++) {
      MPI_Send(buffer, length, MPI_DOUBLE,
               1, PING, MPI_COMM_WORLD);
      MPI_Recv(buffer, length, MPI_DOUBLE,
               1, PONG, MPI_COMM_WORLD,
               &status);
   }
} else {
   for (i = 0;  i < tries;  i++) {
      MPI_Recv(buffer, length, MPI_DOUBLE,
               0, PING, MPI_COMM_WORLD,
               &status);
      MPI_Send(buffer, length, MPI_DOUBLE,
               0, PONG, MPI_COMM_WORLD);
   }
}
```

3.3 MPI_Send

Die ersten drei Parameter sind

```
(void *buf,
int count,
MPI_Datatype datatype, .....)
```



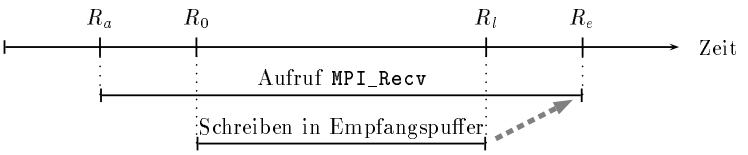
Die restlichen Parameter sind

```
(..... ,
int dest,
int tag,
MPI_Comm comm)
```


3.4 MPI_Recv

Die ersten drei Parameter sind

```
(void *buf,
int count,
MPI_Datatype datatype, .....)
```



Die restlichen Parameter sind `int source,`
`int tag,`
`MPI_Comm comm,`
`MPI_Status *status)`

- `MPI_ANY_TAG`
- `MPI_ANY_SOURCE`

Ergebnis aus `status` ablesen

Absender: `status.MPI_SOURCE`

Tag: `status.MPI_TAG`

Länge: `MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)` umständlich oder geschickt?

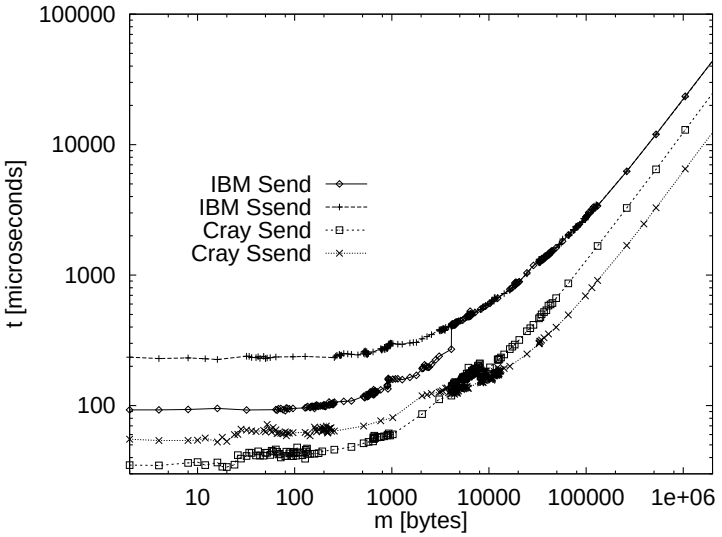
Zwei Operationen S und R passen zueinander:

- Gleicher Kommunikator `comm`.
- R hat von S als `dest` angegebenen Rang.
- S hat von R als `source` angegebenen Rang, oder R hat `MPI_ANY_SOURCE` angegeben.
- Gleiches `tag`, oder R hat `MPI_ANY_TAG` angegeben.
- Gleicher Datentyp ($\approx$)

3.5 Zeitmessungen

- `double MPI_Wtime();` misst Zeit in Sekunden
- `MPI_Barrier(MPI_COMM_WORLD);` synchronisiert alle PEs. Wozu?
- Mittelwertbildung
- Ausreisser rausfiltern. (Wirklich?)

3.6 Ergebnisse auf IBM-SP und Cray T3E



Bandbreiten:

IBM ≈ 100 MB/s

Cray ≈ 300 MB/s