Datenaustausch zwischen mehreren Rechnern, die über ein Netzwerk miteinander verbunden sind.

Möglichkeiten:

## **Zugriff über Netzlaufwerkfreigaben (Dateien)**

Informationen werden in Dateien geschrieben.  
hat einige Nachteile, so dass es insgesamt für die meisten Fälle zu aufwendig wird, weil viel davon eigens implementiert oder sogar programmiert werden muss

* Freigaben, die nicht im gleichen Netzwerk liegen, müssen über den Router und die Firewall nach außen hin zugänglich gemacht werden oder per vpn verbunden sein. Das ergibt einigen tlw. komplexen Installationsaufwand insbesondere wenn verschlüsselt übertragen werden soll. (Teilnehmer müssten keys austauschen)
* Dateirechte müssen angepasst und Zugangsdaten bekannt sein
* Teilnehmer müssen auf Betriebssystemebene die Software haben, um auf die Freigabe zuzugreifen oder diese zu erstellen (zB samba-server und samba client oder nfs-server und clients, ftp)
* Man benötigt ein eigenes Protokoll (in der Art eines token passings oder einer master-slave Umgebung), um den Verbindungsaufbau und die Kommunikation zu kontrollieren, zB eine Datei in der steht welcher Teilnehmer gerade schreiben darf oder für wen Daten bereitstehen. Die Teilnehmer müssten per polling ständig dort nachschauen.

# Informationen in eine Datenbank schreiben, auf die alle Teilnehmer Zugriff haben

ist ein häufig angewandtes Konzept, am besten mit dezentraler oder verteilter Datenbank  
hat große Vorteile wenn die Daten, die ausgetauscht werden, sowieso Teil eines übergeordneten Datenmodells sind, das in einer Datenbank abgebildet ist. Wenn nur wenige Daten zB für Steuerungsaufgaben übertragen werden, zahlt sich der Betreib einer Datenbank dafür aber nicht aus.  
Nachteile:

* Es muss ein Datenbanksystem (der DB Server und dort die Datenbank) installiert sein und so administriert werden, dass für die Teilnehmer die passenden Rechte und Datenbankeinträge vorhanden sind.
* Sichere Verbindung ist nicht per se vorhanden (zB vpn oder Verschlüsselung der Daten muss man auf Anwendungsebene machen)
* Man benötigt einen Datenbanktreiber (zB ODBC, JDBC) damit man von Programmen aus auf die entfernte Datenbank zugreifen kann
* Eventuell Performance, weil Schreiben und Lesen über sql-Statement an die DBMS geschieht

# Datenaustausch über Sockets (TCP, UPD, websockets)

Vorteile

* TCP/IP ist praktisch auf allen Systemen in irgendeiner Form implementiert, auf PCs die im Netzwerk hängen sowieso. Die Programme für den Datenaustausch benötigen also keine zusätzliche Installation von Betriebssystemtools, nur richtig konfgurierte Netzwerkadapter.
* Sehr allgemein gehalten, keine Lizenzen oder Grenzen durch proprietäre Systeme. Empfänger- und Sendersystem sind unabhängig. Verbindung kann ohne Probleme lokal oder extern ablaufen, wenn die beteiligten Systeme mit TCP/IP arbeiten.
* Fehlererkennung und Identifikation ist bereits implementiert.
* Durch Verwendung der openssl-Bibliothek kann man auch verschlüsselte Verbindungen erzeugen
* Implementierung in Programmen recht einfach, man benötigt keine zusätzlichen Installationen
* Im TCP/IP Protokoll ist der Verbindsaufbau und die Kommunikation schon fertig implementiert
* Anpassbar an Situation durch verschiedne Protokolle
  + Fehleranfälligkeit/Fehlerrelevanz UDP/TCP
  + Sicherheit IP/IPSec
  + Aufgabe http/ftp/SMTP
  + Anbindungsart Ethernet/WLAN
* Verschiedene Klassen zb für RAW Sockets (flexibel, aber da muss man alles selber konfigurieren) bis zu fertigen Listener und Client Klassen, wo man mit wenigen Methoden in ein paar Zeilen die Kommunikation aufbaut

Nachteil:  
Eigene Anwendungen verwenden meist Ports (die nicht well known ports), die man auf Firewalls extra freischalten muss, weil sie üblicherweise standardmäßig gesperrt sind.

### Wo liegt TCP/IP im ISO/OSI Modell

(ISO = International Standarisation Organisation)  
(OSI-Modell = Open Systems Interconnection Model)

Schichten:

|  |  |
| --- | --- |
| **Schicht-Bezeichnung** | **Erklärung** |
| 1. Physical | Kabel Standards |
| 2. Data link | Übertragung über Kabel (Fehlererkennung) |
| 3. Network | Von mir zum Router oder anderem Rechner |
| 4. Transport | Art der Kommunikation = wie rede ich mit jemandem |
| 5. Session | Sich über eine Sitzung identifizieren können |
| 6. Presentation | Veraltet – Für Darstellung der Daten |
| 7. Application | Das Programm welches die Verbinung braucht |

## TCP Struktur

Network (z. B. Ethernet)  
internet (z. B. IP)  
Transmission (z. B. TCP)  
Application (z. B. HTTP)

Die TCP/IP-Protokolfamilie folgt dem ISO/OSI Schichtenmodell. Wobei die Schichten 5-7 In einer Application Schicht und Physical und Data link (1-2) in einer Network-Access Schicht zusammengefasst sind.

# Ablauf der Kommunikation

Ablauf ist eigentlich immer gleich, egal ob Windows, Linux, c, Java, …   
man unterscheidet nur stream oder datagramm Verbindung

udp (hat schlankeres Protokoll, ist aber fehleranfälliger, weil Zustellung und Paketreihenfolge nicht kontrolliert wird, ist also gut, wenn nur einzelne Pakete gesendet werden bzw wenn egal ist falls mal ein Paket verloren geht)

sock = socket (AF\_INET, SOCK\_STREAM, 0); /\* for stream - TCP verwendet\*/

sock = socket (AF\_INET, SOCK\_DGRAM, 0); /\* for datagram UDP \*/

Die wichtigen Informationen bei der Erstellung des Sockets sind:

* Netzwerkklasse: AF\_INET für IP oder AF\_INET6 für ipv6
* Und type UDP oder TCP

Beim Binden gibt man Adresse und port an

* Port: (eine Zahl zwischen 1 und 65536, wobei bei Adressen unter 1023 einige als well konwn ports für Übertragungsprotokolle wie http, ftp usw vergeben sind)
* Man kann auch gethostbyname machen, um eine DNS Abfrage zu machen, wenn man nur den Namen und nicht die IP–Adresse kennt

Ablauf:

#### TCP Sockets:

Ist wie telefonieren, dh es wird eine persistente Verbindung aufgebaut:  
eine Person (der Server) sitzt am Telefon (IP Adresse) mit einer Durchwahl (Port) und wartet auf einen Anruf (**listen**)  
andere Person (client) wählt Nummer und Durchwahl des Servers (IP Adresse und Port an) - macht also ein **connect**   
Server sieht Nummer die anruft und hebt ab (**accept**)  
somit eine exklusive Verbindung zwischen Server und Client erstellt   
Beide sprechen miteinander (**stream**),

*Server-seitig*:

1. Server-Socket erstellen
2. binden des Sockets an eine Adresse (Port), über welche Anfragen akzeptiert werden
3. auf Anfragen an einem Port warten (listening)
4. Anfrage akzeptieren (accept) und damit ein neues Socket-Paar für diesen Client erstellen
5. Einen stream öffnen und über diesen byte-streams lesen und schreiben (ist evtl blockierend)  
   fürs Lesen einen asynchronen callback verwenden
6. Stream und Client-Socket wieder schließen.

*Client-seitig*:

1. Socket erstellen
2. erstellten Socket mit der Server-Adresse und dem Port verbinden auf dem der Server horcht (connect) und auf accept warten,
3. Stream öffnen und Daten senden und empfangen
4. evtl. Socket herunterfahren (shutdown())
5. Verbindung trennen, Socket schließen

#### UDP Datagram Sockets:

Sind verbindungslos (dh keine Kontrolle ob Pakete richtig ankommen)  
ist wie das Versenden von SMS

Partner1 horcht an einem Port (port1)  
Partner 2 sendet Meldung (datagramm) an IP-Adresse und Port von Partner 1 (ip1 und port1)  
optional: Partner 2 wartet evtl auf anderem Port (port2) auf eine Bestätigung  
Partner 1 sieht von wem die Meldung ist (ip2) und wertet die Meldung aus  
optional: Partner 1 sendet evtl Bestätigung an Partner2 (ip2 und port2) zurück

Bestätigung muss aber nicht sein, dann weiß man eben nicht ob es angekommen ist, dann ist es eben nur unidirektional

Sind ein bißchen einfacher, weil nur eine Nachricht gesendet wird, Protokoll ist schlanker und deshalb schneller. Paketgröße ist limitiert (Standard sind ca 1kB auf einmal)

*Server-seitig*:

1. Socket erstellen
2. Socket an port binden
3. warten auf Pakete

*Client-seitig*:

1. Socket erstellen
2. An Adresse ip und port senden  
   Geht auch mit broadcast Adressen, dh Datagramme werden an viele Empfänger gleichzeitig gesendet (zB bei Video Streaming), da wäre blockieren wie beim TCp Stream schlecht

Gibt’s für jedes Betriebssystem und fast alle Programmiersprachen

# Beispielcode Sockets

### UDP Sockets

Ist vergleichbar mit dem Versenden von SMS: es wird gesendet, ohne dass man weiß, ob der Partner es bekommt, außer der Partner bestätigt den Empfang, in dem er etwas zurücksendet

Senden und Empfangen erfolgen über verschiedene Ports:

1. Partner 1 sendet auf port 3380 an Partner2 und wartet dann auf Port 3381 auf eine Bestätigung
2. Partner 2 wartet auf Port 3380 auf eine Meldung und gibt freundlicherweise eine Bestätigung über Port 3381 zurück (sonst wartet Partner 1 unendlich lange)

### Partner 1:

UdpClient udpSocket = new UdpClient(0); //hier spielt der Port noch keine Rolle

//Verbindung mit dem Partner öffnen und „Hallo“ auf Port 3380 senden:  
Byte[] sendBytes = Encoding.ASCII.GetBytes(„Hallo“);

udpSocket.Connect(“127.0.0.1“, 3380);

udpSocket.Send(sendBytes, sendBytes.Length);

udpSocket.Close();

### Partner 2:

IPEndPoint remoteEP = new IPEndPoint(IPAddress.Any, 3381);

//ich horche auf Port 3380, ob der Partner was schickt:

UdpClient udpSocket = new UdpClient(3380);

//warten bis was auf port 3380 kommt, in remoteEP steht dann von wem

Byte[] receiveBytes=udpSocket.Receive(ref remoteEP);

udpSocket.Close();

# TCP Sockets

Ist vergleichbar mit dem Telefonieren

1. Partner1 horcht auf einem Port ob jemand eine Verbindung aufbauen will.
2. Partner 2 verbindet sich mit Partner 1 über den Port
3. Partner1 akzeptiert die Verbindung
4. beide senden und empfangen über diese Verbindung (diesen Port)

Es muss ausgemacht sein, wer mit dem Schreiben und wer mit dem Lesen anfängt, sonst gibt es einen Deadlock und beide Partner warten

## Partner 1 (TCP Server)

IPAddress ipAd = IPAddress.Parse("127.0.0.1");

TcpListener myListener = new TcpListener(ipAd, 8001);

myListener.Start(); //warten bis sich jemand über Port 8001 verbinden will

//jetzt ist eine Verbindungsanfrage da, sie wird akzeptiert:

TcpClient tcpclnt = myListener.AcceptTcpClient();

Console.Write("Verbindung aufgebaut\n");

//Kommunikationsstrom öffnen:

Stream stream = tcpclnt.GetStream();

//Kommunikation starten

int i;

byte[] receiveBytes = new byte[tcpclnt.ReceiveBufferSize];

while (true)

{

//Daten empfangen – blockiert bis etwas gelesen wurde

i = stream.Read(receiveBytes, 0, receiveBytes.Length);

string empfangen = System.Text.Encoding.ASCII.GetString(receiveBytes, 0, i);

Console.WriteLine("Du: "+empfangen);

//Antwort senden

Console.Write("Ich: ");

String str = Console.ReadLine();

Byte[] sendBytes = Encoding.ASCII.GetBytes(str);

stream.Write(sendBytes, 0, sendBytes.Length);

}

## Partner2 (TCP Client)

TcpClient tcpclnt = new TcpClient();

tcpclnt.Connect("127.0.0.1", 8001);

Stream stream = tcpclnt.GetStream();

int i;

byte[] receiveBytes = new byte[tcpclnt.ReceiveBufferSize];

while (true)

{

//Daten senden

Console.Write("Ich: ");

String str = Console.ReadLine();

Byte[] sendBytes = Encoding.ASCII.GetBytes(str);

stream.Write(sendBytes, 0, sendBytes.Length);

//Daten empfangen – blockiert bis etwas gelesen wurde

i=stream.Read(receiveBytes, 0, receiveBytes.Length);

string empfangen = Encoding.ASCII.GetString(receiveBytes, 0, i);

Console.WriteLine("Du: " + empfangen);

}

Natürlich sollten die Kommunikation, der stream und die connection vernünftig mit Close() geschlossen werden

# Websockets:

Arbeiten mit einem TCP Verbindungsaufbau und http Paketen, wobei mit einem Handshake begonnen wird, bei dem ausgemacht wird, dass die Teilnehmer ihre Pakete zwar als http senden, aber auf ein erweitertes Protokoll, nämlich websocket, updaten. Dadurch wird aus dem eigentlich zustandslosen http Protokoll eine persistente bidirektionale Verbindung zwischen den Teilnehmern.

Beispiel:

var socket = new WebSocket(urlToWebsocketServer);

// callback-Funktion wird gerufen, wenn die Verbindung erfolgreich aufgebaut werden konnte

socket.onopen = function () {

console.log("Verbindung wurde erfolgreich aufgebaut");

};

// callback-Funktion wird gerufen, wenn eine neue Websocket-Nachricht eintrifft

socket.onmessage = function (messageEvent) {

console.log(messageEvent.data);

};

// callback-Funktion wird gerufen, wenn ein Fehler auftritt

socket.onerror = function (errorEvent) {

console.log("Error! Die Verbindung wurde unerwartet geschlossen");

};

socket.onclose = function (closeEvent) {

console.log('Die Verbindung wurde geschlossen --- Code: ' + closeEvent.code + ' --- Grund: ' + closeEvent.reason);

};

# 5) MQTT:

Eine weitere sehr elegante Möglichkeit des Informationsaustuschs zwischen Rechnern ist das MQTT Protokoll, welches das Broker, Subscriber, Publisher Konzept verwendet. (dahinter steckt eigentlich wieder TCP, aber mit einem eigenen sehr schlanken Protokoll)

Einfach geht die mqtt Verbindung mit Java Script zb bei node.js, hier wird geschrieben

var hostname = "http://iot.eclipse.org";

var clientId = "myusername";

clientId += new Date().getUTCMilliseconds();;

mqttClient = new Paho.MQTT.Client(hostname, 1883, clientId);

mqttClient.connect();

mqqtClient.publish(topic, msg);

Fürs Empfangen müsste man in regelmäßigen Abständen einen request senden und den response in einer callback Funktiona auswerten

mqttCclient.onMessageArrived = onMessageArrived;

function onMessageArrived(message)   
{

var info=message.payloadString

}