7 Transport Layer

7.1 Task of the Transport Layer

The transport layer (layer 4 in the ISO/OSI protocol stack) connects the technical infrastructure of hardware and network technologies with the world of applications.

Data stations often enable the simultaneous execution of several user processes, e.g. in distributed client-server applications. Several applications can be active on a station at the same time, e.g. email client and FTP service. The tasks of the transport layer consist in the realization of the data transport from end system to end system and in the correct assignment of the transmitted data to a user process.

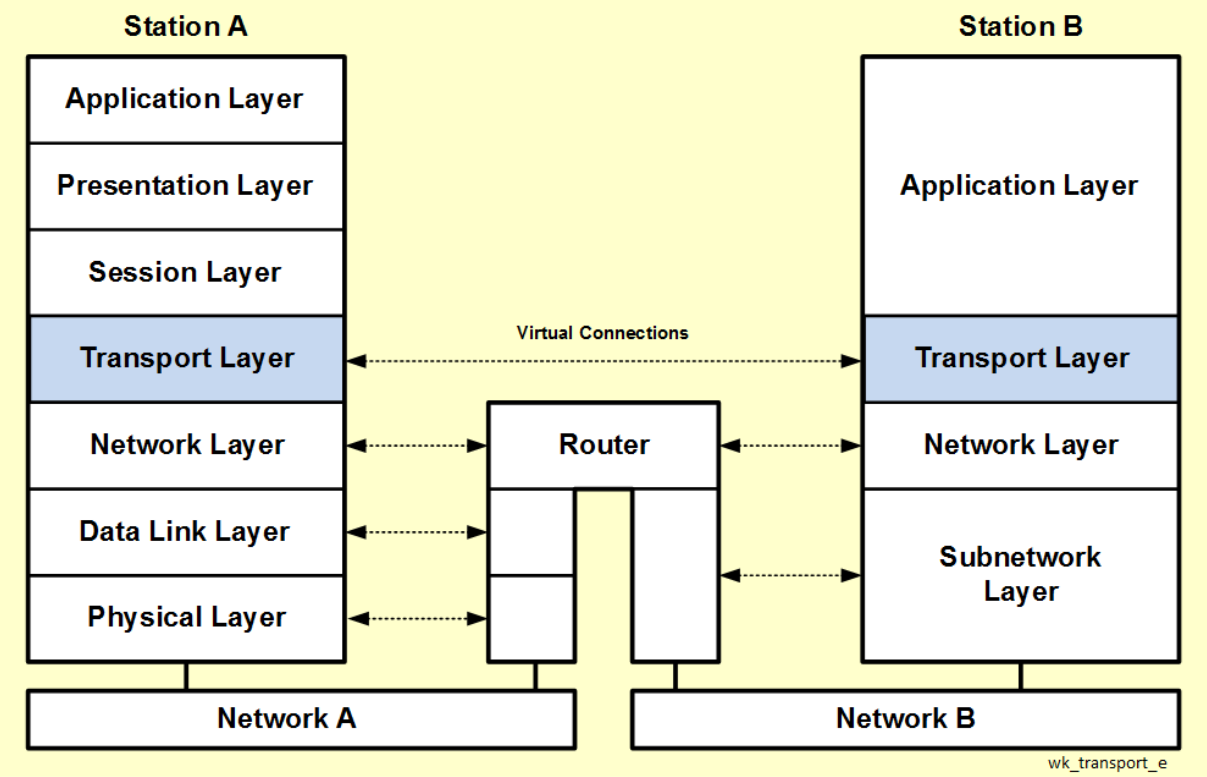


Fig. 7.1-1: Transport layer in the ISO/OSI model and in the TCP/IP architecture

On the one hand, the transport layer ensures error-free data transmission from the session layer (layer 5) to the network layer (layer 3) and vice versa. On the other hand, it is responsible for decoupling the higher layers 5 to 7 from changes in the network hardware. It regulates the data flow within a host computer.

The special tasks of the transport layer are summarized below:

- Establishing a separate network connection for each transport connection required by the session layer,
- Installation of multiple network connections and distribution of data in case of high data flow,
- Multiplex operation of several transport connections on one network connection,

- Insertion of a "transport header" to mark messages in case of multi-program operation in host machines,
- Flow control in case of simultaneous activities of different speed in host machines,
- Determination of the type of service provided to the session layer and the network user (point-to-point channel, broadcasting from one sender to many receivers),
- Simultaneous management of data from several processes on one host computer,
- Fragmentation of session layer messages during transmission,
- Packet reassembly in the network layer during reception.

7.2 Protocol access on the transport layer

Internet addresses uniquely identify each host connected to the network worldwide. To transport the data of a user process from the sender to a second user process at the receiver, the protocol accesses of the transport layer, the so-called ports, provide a higher-level addressing environment.

Each application process on a host is assigned a 16-bit-wide access number (port number, port address) for identification purposes. The address combination of Internet address and port address is called a socket in software terms.

The TCP/IP protocol defines the use of port addresses according to the following conventions:

Port address	**Usage**
Standard services:	
0	Reserved,
1 ... 255	Reservation for standard services (well known ports),
User defined services:	
256 ... 1023	Other reserved accesses (originally used for UNIX services, today also used in routing),
1024 ... 4999	Accesses for short-term clients in client-server applications,
5000 ... 65535	User defined server accesses.

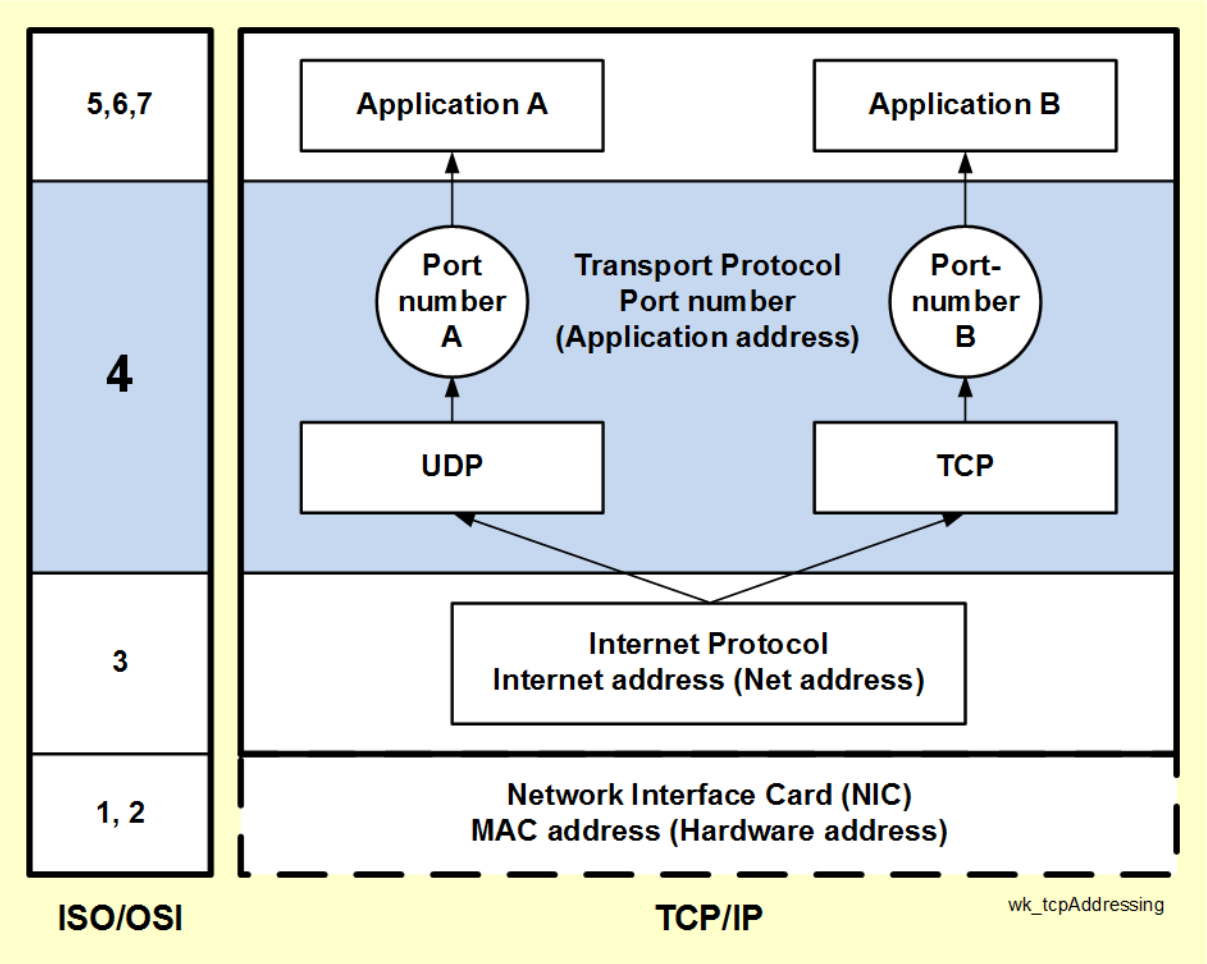


Fig. 7.2-1: Addressing of application processes

The TCP/IP protocol uses two different transport protocols at the transport layer:

- connectionless **User Datagram Protocol UDP**,
- connection oriented **Transport Control Protocol TCP**.

7.3 Connectionless Transport

TCP/IP provides a **connectionless transport** option with the **User Datagram Protocol UDP**. UDP is based on the Internet Protocol (IP protocol). The task of the User Datagram Protocol UDP is to deliver a data packet, the user datagram, from an application process on the source host to a user process on the destination host. UTP datagrams are encapsulated in IP datagrams during information transfer. For datagram delivery, UDP supplements IP information with two additional mechanisms:

- Address mechanism to identify the application processes,
- Minimal reliability mechanism in the form of an optional checksum.

The UDP header contains the destination port of the receiver and possibly (optionally) the sender-side port number for addressing the applications:

- **Source Port:**
  - Optional address specification of the UDP send port, to be able to address the send process directly to the sending process,
  - Value zero if source port is not specified,

- **Destination Port:**
  Port number of the application on the receiver to which the UDP datagram is to be sent,

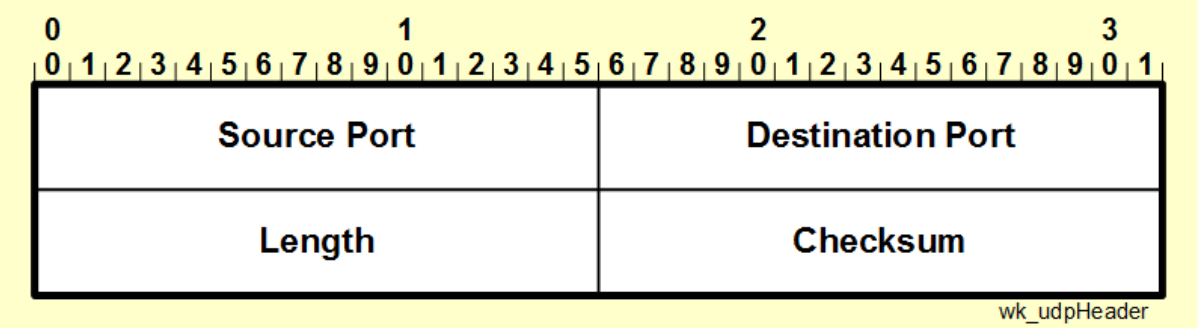


Fig. 7.3-1: Header of the UDP protocol

Examples: Port numbers for UDP services

Service	Port	Description
Echo	7	Message of a received user datagram to the sender,
Discard	9	Deletion of a user datagram,
Daytime	13	Return of values for date and time,
Chargen	19	Returning a character string with a randomly selected length,
Domain	53	Name server process of the Domain Name Service,
BOOTPS	67	Server port for downloading configuration data (bootstrap protocol server),
BOOTTPC	68	Boot protocol for the client,
TFTP	69	Server process of the Trivial File Transfer Protocol,
SunRPC	111	used in the implementation of the SUN-RPC service (RPC: Remote Procedure Call),
NTP	123	Used in the implementation of the Network Time Protocol NTP,
SNMP	161	Organization of the reception of SNMP network management requests (SNMP: Simple Network Management Protocol),
SNMP	162	Receiving SNMP problem messages.

The connection-independent UDP protocol is an unreliable service and can corrupt, lose, deliver in disorderly order, or duplicate datagrams.
The recipient's application process must provide its own reliability checks, if necessary.

In addition to port information, the UDP header contains two additional data fields:

- **Length:**
  Length of the UDP datagram including header and UDP data part,

- **Checksum:**
  Optional calculation of a checksum from the UDP header and a
  96-bit pseudo header (IP source address (32 bits), IP destination address (32 bits), empty field (8 bits), protocol identifier (8 bits), information about the length of the TCP segment (16 bits)).

The receiving stations working with UDP use buffers to queue the UDP datagrams arriving at the application port. In the event of a buffer overflow, the corresponding datagrams are removed without an error message.

The receiver, e.g. the server in a client-server application, does not notice the overflow in the receive buffer.
The system administrator, on the other hand, can use the **`netstat`** command to check for the overflow.

Example:

```
netstat -s
udp:
3 incomplete headers
0 bad data length field
0 bad checksums
34 socket overflows
```

The reasons for using the User Datagram Protocol UDP are different:

- Distributed client-server applications with simple question-answer protocols,

- Data communication with little data, so that an unnecessarily large overhead would be created by establishing and terminating connections.

7.4 Connection oriented Transport

7.4.1 Tasks and Ports

The **Transmission Control Protocol TCP** realizes under TCP/IP a connection-oriented transport in full-duplex technology with reliable data transmission and ordered data stream. Reliable transport means that the receiver either receives all information (TCP messages, TCP sequences) or receives a message that an error occurred during transmission. An ordered data stream has the same octet sequence as sent by the sender.

A logical connection, a link, is established between the sender's transport layer process and the corresponding receiver's process. The Transmission Control Protocol TCP then performs all endpoint to endpoint control.
According to the TCP protocol, the message transmission of the higher layers takes the form of an unstructured stream of octets from the sender's TCP port to the receiver's TCP port. However, the data stream is segmented during transmission and enclosed in IP datagrams. The segment structure of the TCP segments is hidden from both users in the sender and the receiver.
The TCP protocol is independent of the network-specific properties of lower protocol layers. TCP segments can have lengths of up to 65 kBytes.

The applications of the TCP protocol are manifold and varied. Some examples are shown in the table below:

Service	Port	Description
Echo	7	Message of a received TCP message to the sender
Quote	17	Returns a string with the "phrase of the day",
FTP_DATA	20	Data transfer with the File Transfer Protocol FTP,
FTP	21	Service for implementing FTP control functions,
SSH	22	Secure logon to a foreign host and the possibility to use resources there,
Telnet	23	Logging on a remote host and the possibility of using resources there,
SMTP	25	Electronic mail with Simple Mail Transfer Protocol SMTP,
DNS	53	Domain Name Service: administration and resolution of symbolic names,
Finger	79	Information about a user,
HTTP	80	Distributed, electronic information service on the Internet using hyperlinks (HTTP: HyperText Transfer Protocol),
HTTPS	81	Secure HTTP connection,
TLS	465	Transport Layer Security: Secure email, outgoing mail (POP3), t-online,
TLS	995	Secure email, mail inbox (POP3), t-online.

7.4.2 TCP Header

TCP messages (TCP segments) consist of the TCP header, the TCP options and the actual data part.

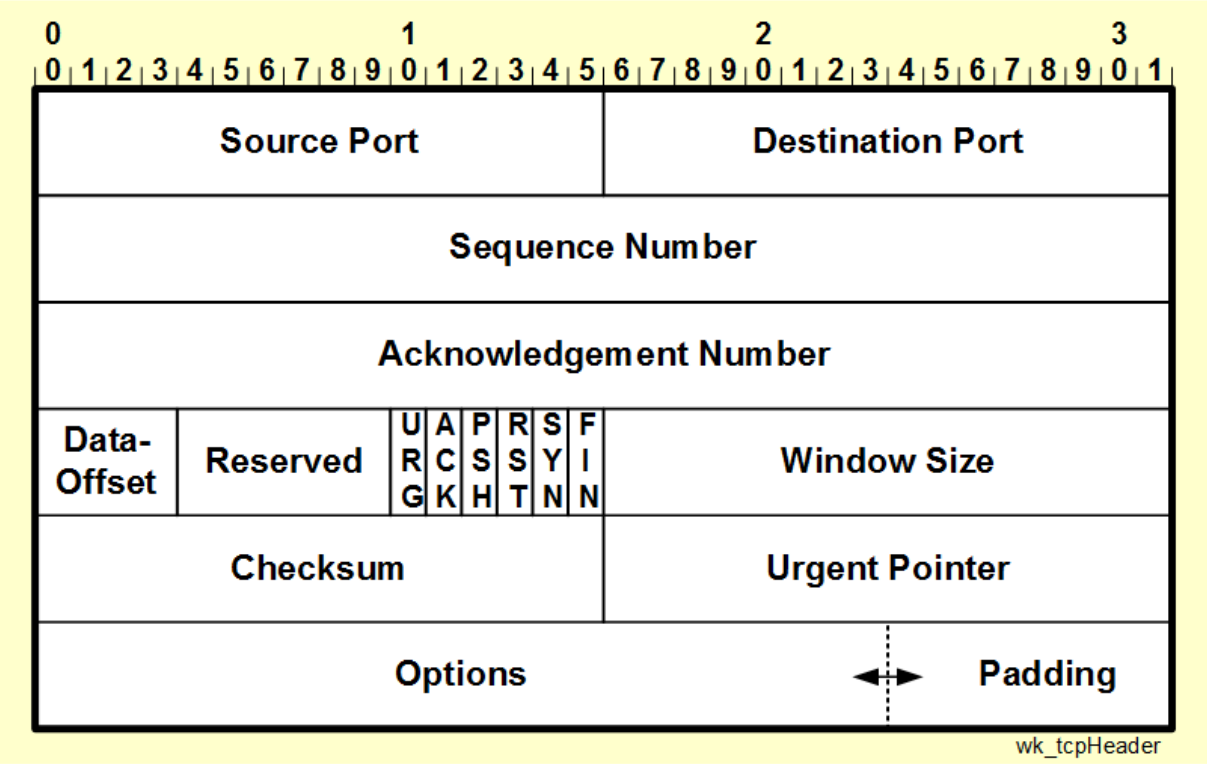


Fig. 7.4.2-1: Header of the TCP protocol

The TCP header has a length of 20 bytes without options:

- **Source Port:**
  Protocol port of the application process on the sender,
- **Destination Port:**
  Port of the application on the receiver to which the TCP message is to be transmitted,
- **Sequence Number:**
  Identification of each octet in the TCP data stream by the sender to enable correct classification by the receiver,
- **Acknowledgement Number:**
  Acknowledgement of all received data and indication of the next expected data byte by the receiver,
- **Data Offset (Data Offset):**
  Number of 32-bit words in the TCP header (including options and padding field),

- **Control Flags:**
  One-bit indicators used to control the establishment, maintenance, and termination of a connection:

  - **Urgent Pointer (URG):**
    Marks precedence data if URG = 1,

  - **Acknowledgement (ACK):**
    Acknowledgement of received data, if ACK = 1,

  - **Push (PSH):**
    - Immediate transmission of data to the higher protocol, if PSH = 1,
    - Normal case: Data buffering by TCP and sending, if a certain minimum amount of data has accumulated,
    - Application: Telnet,

  - **Reset (RST):**
    - Sender's request to reset the connection if RST = 1,
    - Applications: Error case in the data line, correct booting after a hardware or software crash,

  - **Synchronization (SYN):**
    - Sender's request to establish connection, if SYN = 1,
    - Synchronization: sending of an "initial sequence number",

  - **Final (FIN):**
    Final termination of the connection and no further data transmission from higher protocols by the sender, if FIN = 1,

- **Window Size:**
  - Flow control between sender and receiver:
    Notification to the sender about the current buffer size of the receiver (sliding window),

- **Checksum:**
  Mandatory calculation of a checksum from the TCP header and a 96-bit pseudo header (IP source address (32 bits), IP destination address (32 bits), blank field (8 bits), protocol identifier (8 bits), information about the length of the TCP segment (16 bits)),

- **Urgency (Urgent Pointer):**
  - Pointer to information with high urgency,
  - Urgent Pointer points to the end of the precedence data,
    value of the pointer corresponds to the sequence number offset for the end of the precedence data,
  - precedence data is transmitted immediately after the TCP header,
    the value of the pointer transmitted,
  - Examples: Breaks and interrupts in virtual terminal sessions,

- Options:
  TCP options defined so far:

Type	Length	Meaning
0	-	end of the options list,
1	-	no operation (No Option): Separation of two options,
2	4	maximum segment length (number of octets, default value: 536 bytes).

- **Padding information:**
  Padding information to ensure that TCP header has the prescribed length.

7.4.3 Stages and properties of the TCP connection

The transmission of TCP messages within the TCP protocol comprises several substeps and subtasks:

- **Connection establishment by a handshake procedure (three-way dialog, three-way handshake) and synchronization:**

  - Sender (client) sends a connection request to the destination host (server): SYN = 1, passing an initial sequence number,

  - Destination host (server) sends an acknowledgement back to the sender (client) and requests the sending host (client) to synchronize with the initial sequence number:
    SYN = 1, ACK = 1, sending back an acknowledgement number,

  - Sending host (client) acknowledges the synchronization request from the destination host (server) with another acknowledgement: SYN = 0, ACK = 1,

  Remark:
  An acknowledgement number always corresponds to the sequence number that is expected next.

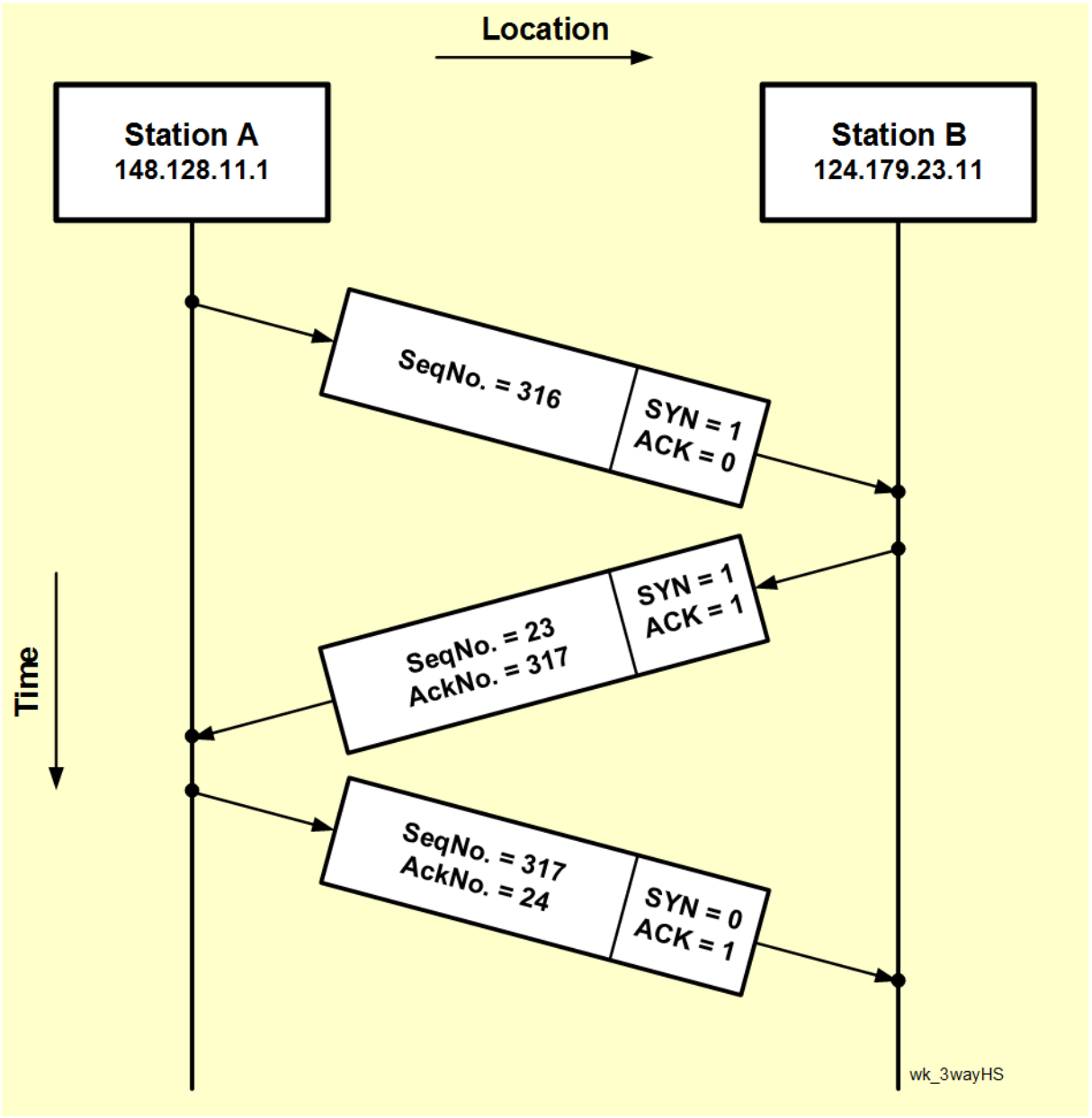


Fig. 7.4.3-1: Space-Time-Diagram of the three-way handshake for establishing a connection

- **Data transmission:**

  - TCP installs a full-duplex path between the two communication participants,

  - Optimization of the network bandwidth by controlling the data flow: Application of the concept of "sliding windows" (sliding windows), instead of sending acknowledgement messages for each message,

  - data is transferred from the sender's transmission buffer, packaged in IP-datagrams, to the receiver's receive buffer,

- **Error detection and correction:**

  - Indication of the location of each octet in the overall data stream by a section number field with the send sequence number in the TCP header,

  - Verification of the send sequence number by the receiver,

  - Receiver includes an acknowledgement number with the section number in the next octet it expects to receive when retransmitting data in each segment (confirming successful receipt of all those octets transmitted before the octet specified in the acknowledgement number field),

  - Use of a retransmission timer to detect datagram loss: retransmission in case of a timeout before an acknowledgement arrives,

  - Calculation of a checksum for each transmitted segment and inclusion in the corresponding segment by the sender,

  - Verification of the checksum by the receiver, removal of the segment in case of error (no confirmation by the sender in case of error -> timeout!),

- **Flow control:**

  - Adjustment of relative transmission speeds between sender and receiver,

  - Definition of a window size at connection establishment, which defines the number of octets to be transmitted before receiving an acknowledgement message,

  - Dynamic adjustment of the current window size with each confirmation message,

- **Blocking control:**

  - Limiting the network load of an Internet if the entire network or a subnetwork is congested,

  - Prevention and reduction of network blocking,

  - Regulation by the sending TCP process,

- **Termination the connection by means of a two-way dialog (Two Way Handshake):**

  - Step 1: "Active closing":
  - Sending a message with the "Terminated" flag set (FIN = 1):
  - Station cannot send any more data, but can still receive data,

- Step 2: "Passive closing":
- Receiver confirms the closing message with set ACK flag and sends again a message with set "Terminated flag" (FIN = 1),

- Termination possible at any time by one of the two communication partners or by the TCP service itself,

- Closing of the data transmission in only one direction: Half Close, Application: Customer-specific network applications.

7.5 Client-Server Relationship

The software development of client-server systems mainly takes place in the application layers.
In a client-server application, the client first establishes the connection to the server and requests an application service. The server executes the service and sends the result back to the client. The processes on the client and the server are addressed via the port addresses.
The entire activities of the necessary coding, securing and transmission technology are carried out by the so-called "socket", which manages and implements the tasks of the lower network layers.
The Internet address and the port address(es) serve as software transfer parameters.

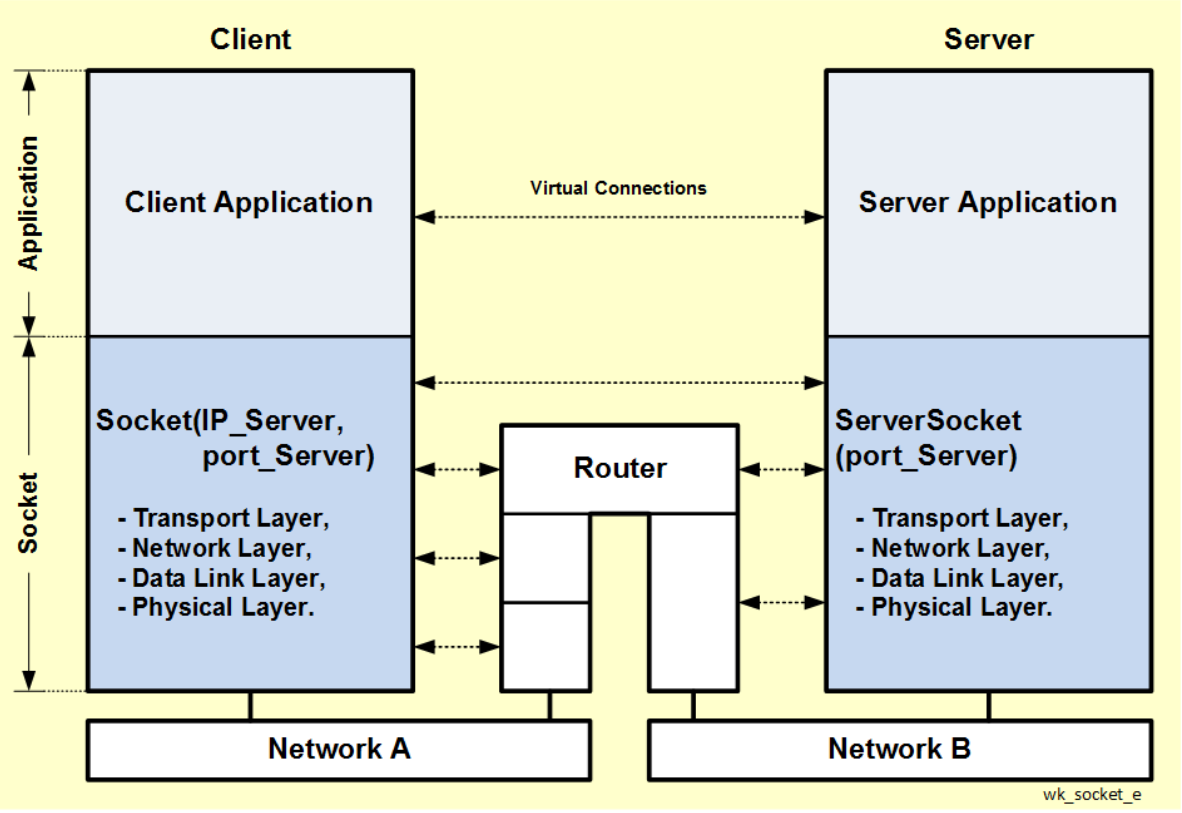


Fig. 7.5-1: Client-server layered structure with application part and socket

7.6 Practical realization of a simple TCP protocol

7.6.1 Form and method of practical implementation

The software implementation of protocols for the transport layer is generally relatively complex. The necessary scope depends on the qualities of the underlying network layer.
In the case of a reliable datagram service, extensive error correction must be taken into account.

The transport instance can be practically implemented in various forms:

- Part of the operating system of the hosts,
- Software package in the form of library routines,
- Integration in coprocessor chip,
- Integration in network card.

In addition to the actual tasks within the transport layer, the software implementation must provide the interfaces for the underlying network layer and the user layers above it.

In the general ISO/OSI protocol, the addresses of the transport layer are referred to as Service Access Points SAPs:

Network Layer Address:	**Network Service Access Point NSAP** Example: IP address in the TCP/IP protocol,
Addresses to the user layers:	**Transport Service Access Point TSAP** Examples: Port numbers in the TCP/IP protocol,

To keep the software dependency on the network layer as low as possible, only descriptive parameters are passed between the protocol layers rather than the data packets themselves.

If a transport layer send attempt is made when the time window of the underlying virtual connection is "filled", access to the network layer must be prevented. The implementation is done within the network layer (enable_transport_layer, disable_transprot_layer functions) and is transparent to the transport protocol.

7.6.2 Task and realization

The software of a connection-oriented, reliable protocol for the transport layer is to be developed.

A structure of the type field **`conn[MAX_CONN]`** manages the connections in single records (maximum **`max_conn`** connections).

The transport layer shall provide five different services. Each service is implemented in the form of a function:

Service	Function, Remarks
LISTEN	**`int listen(local_address);`** - Call if interested in an incoming call, - Specification of a TSAP for addressing, - blocking until remote process tries to establish a connection,
CONNECT	**`int connect(transport_address_1`** **`transport_address_2);`** - Function with the local and remote address, - Attempt to establish a transport connection, - Sending a CALL-REQUEST packet, - Output a "connection number" to indicate the established connection if successful, - Unsuccessful attempt: return of a negative "connection number number",
SEND	**`int send(connection_number, buffer, bytes);`** - Actively send (transmit) a message in a data buffer, - Return value: indication of send success or failure,
RECEIVE	**`int receive(connection_number, buffer, *bytes);`** - Passively receive a message (readiness of the caller to receive the message), - Block until TPDU has arrived, - Return value: indication of success or failure,
DISCONNECT	**`int disconnect(connection_number);`** - Terminate the transport connection, - Symmetric connection termination, - Return value: indication of success or failure.

When a data packet arrives at the receiver, a function
`void packet_arrival(void)` must fetch the incoming packet and process it.

The interface to the network layer is implemented with two functions:

Function	Task
`to_net(...);`	Registration of all parameters by the transport instance,
`from_net(...);`	Splitting of incoming data packets by the network layer for the transport layer.

The following parameters must be passed:

- Parameter 1: Connection identifier,
- Parameter 2: Q-bit (qualifier): Control message, if Q-bit = 1, normal transmission: Q-bit = 0,
- Parameter 3: M-bit: Reference to further subsequent packets (M-bit = 1),
- Parameter 4: Message type,
- Parameter 5: Pointer to the data,
- Parameter 6: Number of data bytes.

Six different message types exist:

Message type	Meaning
CALL REQUEST	Request to establish a connection,
CALL ACCEPTED	Response to CALL REQUEST,
CLEAR REQUEST	Request for connection termination,
CLEAR CONFIRMATION	Response to CLEAR REQUEST,
DATA	Data transmission,
CREDIT	Management of the time window.

Two interrupt functions are to be called by external events:

Function	Remarks
`void packet_arrival(void)`	- Call when a data packet arrives, - Activation only in case of "sleeping" user process,
`void clock(void)`	- Call of the timer, - Activation only in case of "sleeping" user process.

```
/*  Protocol example for the transport layer                  */
/*  A.S. Tanenbaum, Computernetzwerke, Prentice Hall:
    Muenchen u.a. 1997                                        */

/*  File:  transp.c                                           */

#define MAX_CONN 32        /* Maximum number of simultaneous
                              connections                     */
#define MAX_MSG_SIZE 8192  /* Max. message size  (Bytes)      */
#define MAX_PKT_SIZE 512   /* Max. packet size (Bytes)        */
#define TIMEOUT 20
#define CRED 1
#define OK 0

#define ERR_FULL   -1
#define ERR_REJECT -2
#define ERR_CLOSED -3
#define LOW_ERR    -4

typedef int transport_address;
typedef enum {CALL_REQ, CALL_ACC, CLEAR_REQ, CLEAR_CONF,
              DATA_PKT, CREDIT} pkt_type;
typedef enum {IDLE, WAITING, QUEUED, ESTABLISHED, SENDING,
              RECEIVING, DISCONN} cstate;

/* Global variables */
transport_address listen_address; /* Local transport address */
int listen_conn;                  /* Auxiliary variable for
                                     listening to connections */
unsigned char data[MAX_PKT_SIZE]; /* Auxiliary field for
                                     packages                 */

struct conn
       {
          transport_address local_address, remote_address;
          cstate state;                    /* State of
                                              connections     */
          unsigned char *user_buf_addr;    /* Pointer to re-
                                              ceiving buffer  */
          int byte_count;                  /* Sender/receiver
                                              counter */
          int clr_req_received;            /* Packet arrival  */
          int timer;                       /* Timeout for
                                              CALL_REQ packets */
          int credits;                     /* max. number of
                                              messages to be
                                              sent            */
       } conn[MAX_CONN];

void sleep(void);
void wakeup(void);
```

```
void to_net(int cid, int q, int m, pkt_type pt,
            unsigned char *p, int bytes);
void from_net(int *crd, int *q, int *m, pkt_type *pt,
              unsigned char *p, int *bytes);
void packet_arrival(void);


int connect(transport_address l, transport_address r)
/*  User wants to connect to remote process and sends
    CALL_REQ packet                                          */
{
   int i;
   struct conn *cptr;

   data[0] = r;           /* for CALL_REQ packet             */
   data[1] = l;           /* for CALL_REQ packet             */
   i = MAX_CONN;          /* Saerch table backwards          */

   while (conn[i].state!=IDLE && i>1)
         --i;

   if (conn[i].state==IDLE)
   {
      /*  Table entry that CALL_REQ was sent                 */
      cptr = &conn[i];
      cptr->local_address = l;
      cptr->remote_address = r;
      cptr->state = WAITING;
      cptr->clr_req_received = 0;
      cptr->credits = 0;
      cptr->timer = 0;
      to_net(i, 0, 0, CALL_REQ, data, 2);

      sleep();         /* Wait for CALL_ACC or CLEAR_REQ     */

      if (cptr->state==ESTABLISHED)
         return(i);
      if (cptr->clr_req_received)
      {
         /* Refusal of the receiver to connect              */
         cptr->state = IDLE;   /* Return to IDLE state      */

         to_net(i, 0, 0, CLEAR_CONF, data, 0);
         return(ERR_REJECT);
      }
   }
   else
      return(ERR_FULL);        /* CONNECT rejected,
                                  no table position         */
}
```

```
int send(int cid, unsigned char bufptr[], int bytes)
/* Attempt to send a message                                        */
{
   int i, count, m;
   struct conn *cptr=&conn[cid];

   /* Entering send state */
   cptr->state = SENDING;
   cptr->byte_count = 0;     /*  bytes sent so far in
                                 current message                    */
   if (cptr->clr_req_received==0 && cptr->credits==0)
      sleep();
   if (cptr->clr_req_received==0)
   {
      /* Credit available,
         if necessary split message into packets                    */
      do                       /* Send loop                         */
      {
      if (bytes-cptr->byte_count>MAX_PKT_SIZE)
      {
         /* Message in multiple packages                            */
         count = MAX_PKT_SIZE;
              m = 1;           /* More packages later               */
      }
      else
      {
         /* Message in a packet                                     */
         count = bytes - cptr->byte_count;
         m = 0;              /* Last package of this message       */
      }

      for (i=0; i<count; i++)
          data[i] = bufptr[cptr->byte_count+i];

      to_net(cid, 0, m, DATA_PKT, data, count); /* Send a
                                                   package          */
      cptr->byte_count += count;          /* Counting bytes
                                             sent so far            */
      }  while (cptr->byte_count<bytes);

      cptr->credits--;
      cptr->state = ESTABLISHED;
      return(OK);
   }
   else
   {
      cptr->state = ESTABLISHED;
      return(ERR_CLOSED); /* Send operation failed,
                             Desire of the partner to terminate */
   }
}
```

```
int receive(int cid, unsigned char bufptr[], int *bytes)
/* Reception, receiver is prepared for reception             */
{
   struct conn *cptr=&conn[cid];

   if (cptr->clr_req_received==0)
   {
      /* Attempt to receive while connection is established  */
      cptr->state = RECEIVING;
      cptr->user_buf_addr = bufptr;
      cptr->byte_count = 0;
      data[0] = CRED;
      data[1] = 1;
      to_net(cid, 1, 0, CREDIT, data, 2);   /* Send credit   */
      sleep();                  /* Block in waiting for data */
      *bytes = cptr->byte_count;
   }

   cptr->state = ESTABLISHED;
   return(cptr->clr_req_received ? ERR_CLOSED : OK);
}
```

```
int disconnect(int cid)
/* Disconnection                                             */
{
   struct conn *cptr=&conn[cid];

   if (cptr->clr_req_received)
   {
      /* Other side has initiated disconnection             */
      cptr->state = IDLE;      /* Connection is disconnected */
      to_net(cid, 0, 0, CLEAR_CONF, data, 0);
   }
   else
   {
      /* Current disconnection                              */
      cptr->state = DISCONN;      /* Disconnection only when
                                     partner agrees          */
      to_net(cid, 0, 0, CLEAR_REQ, data, 0);
   }
   return(OK);
}
```

```
void packet_arrival(void)
/*   Fetching an arrived package and processing                */
{
   int cid;    /* Connection on which package has arrived     */
   int count, i, q, m;
   pkt_type ptype;  /* CALL_REQ, CALL_ACC, CLEAR_REQ,
                       CLEAR-CONF, DATA_PKT, CREDIT            */
   unsigned char data[MAX_PKT_SIZE];  /* Data part of the
                                         received packet       */
   struct conn *cptr;

   from_net(&cid, &q, &m, &ptype, data, &count);    /* Fetch */
   cptr = &conn[cid];
   switch (ptype)
   {
      case CALL_REQ:          /* Remote user wants to connect */
           cptr->local_address = data[0];
           cptr->remote_address = data[1];
           if (cptr->local_address==listen_address)
           {
              listen_conn = cid;
              cptr->state = ESTABLISHED;
              wakeup();
           }
           else
           {
              cptr->state = QUEUED;
              cptr->timer = TIMEOUT;
           }
           cptr->clr_req_received = 0;
           cptr->credits = 0;
           break;

      case CALL_ACC:    /* Remote user has accepted CALL_REQ */
           cptr->state = ESTABLISHED;
           wakeup();
           break;

      case CLEAR_REQ:     /* Remote user wants to disconnect
                             or deny connection                */
           cptr->clr_req_received = 1;
           if (cptr->state==DISCONN)
               cptr->state = IDLE;       /* Remove collision */
           if (cptr->state==WAITING ||
               cptr->state==RECEIVING ||
               cptr->state==SENDING)
              wakeup();
           break;

      case CLEAR_CONF:  /* Removed user with removal agree   */
           cptr->state = IDLE;
           break;
```

```
      case CREDIT:      /* Removed user with removal agree   */
           cptr->credits += data[1];
           if (cptr->state==SENDING)
              wakeup();
           break;

      case DATA_PKT:    /* Remote user has sent data         */
           for (i=0; i<count; i++)
               cptr->user_buf_addr[cptr->byte_count+i] =
               data[i];
           cptr->byte_count += count;
           if (m==0)
              wakeup();
   }
}


void clock(void)
/* Clock generator has switched on,
   query pending connection request for timeout              */
{
   int i;
   struct conn *cptr;

   for (i=1; i<=MAX_CONN; i++)
   {
      cptr = &conn[i];
      if (cptr->timer>0)
      {
         /* Timer laeuft                                     */
         cptr->timer--;
         if (cptr->timer==0)
         {
            /* Timer ist abgelaufen                          */
            cptr->state = IDLE;
            to_net(i, 0, 0, CLEAR_REQ, data, 0);
         }
      }
   }
}
```

Exercise

E.7.6.2-1: Add a function to listen for possible connection requests and to check if a CALL_REQ has arrived.

7.6.3 Finite-State-Machine as a development tool

Because of the great complexity involved in developing transport instances, it is useful to think of the protocol in terms of a "finite-state machine".

Finite state machines are used to graphically represent complex relationships in communication networks. All possible events in a communication link, e.g. service operations and different possibilities when data packets arrive, are combined and related to the possible states of the transport link.

A matrix form with the events in the rows and the states in the columns is often used for graphical representation. Each entry in a matrix element contains up to three details:

- **Predicate** - Circumstances under which the action takes place,
  - Specification of the predicate optional,
- **Action** - often subdivided into main actions and secondary actions,
  - Tilde: no main action takes place,
- **New state**.

Empty matrix elements indicate invalid or impossible events.

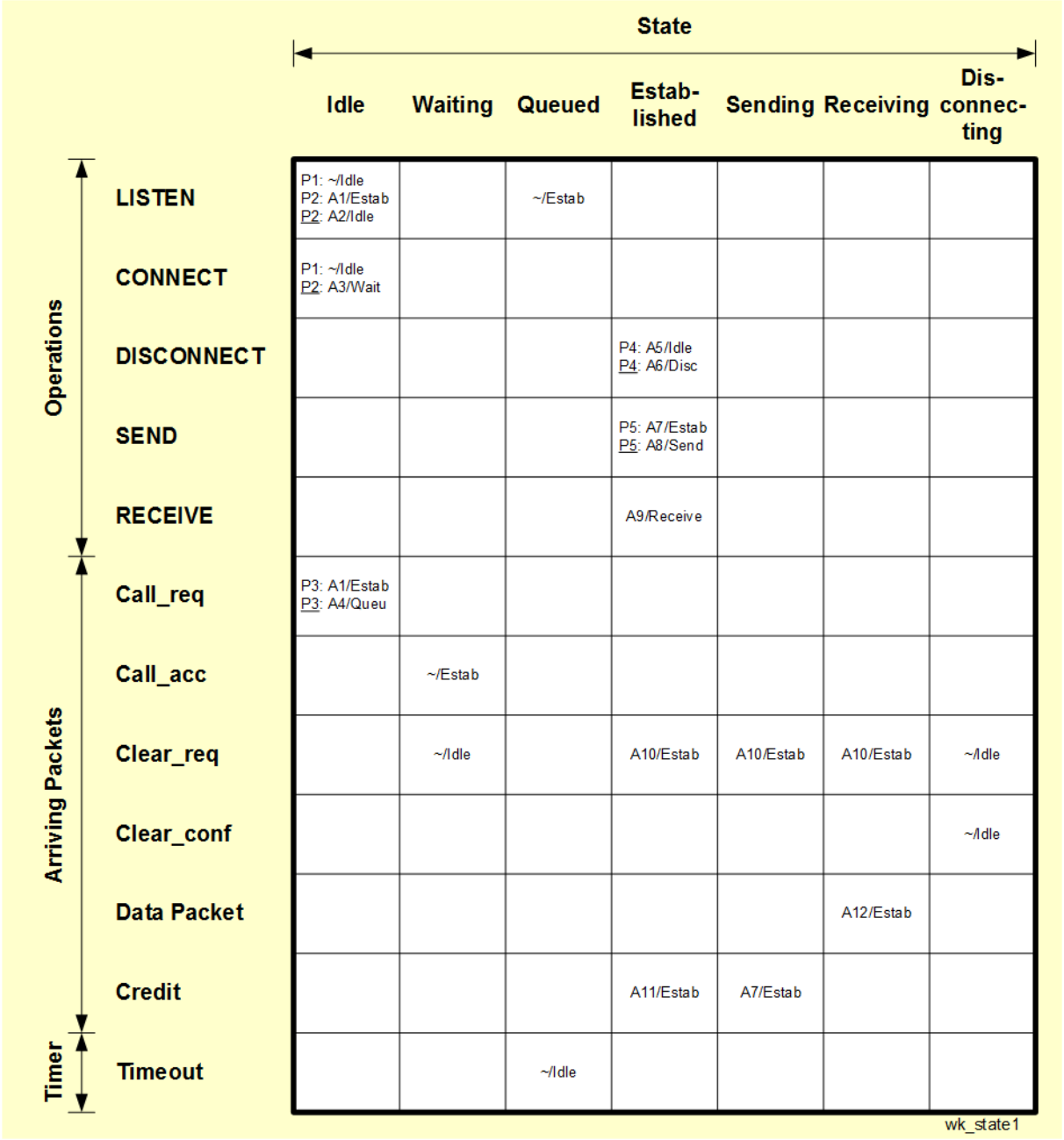
The matrix notation of the finite-state machine offers three serious advantages:

- **Clear representation of any combination of state and event:**
  - Recognition of necessary actions during development,
- **Concise representation of the execution:**
  - Viewing a matrix element `a[i][j]` as a pointer to a function,
  - Function manages the occurrence of event `i` in state `j`,
- Simple and clear protocol description:
  - Use in description of standard documents.

Another commonly used representation of the finite state machine concept is the construction of a flow diagram with the states described as rectangles and the events as directed connections (arrows) between the states.

The characteristics of the flow diagram are:

- Emphasis on dynamic properties,
- Representation of states in the form of rectangles,
- Representation of events: directed connections (arrows) between the states.



Predicates
P1: Connection table full
P2: Call_req pending
P3: LISTEN pending
P4: Clear_req peding
P5: Credit available

Actions:
A1: Send Call_acc
A2: Wait for Call_req
A3: Send Call_req
A4: Start Timer
A5: Send Clear_conf
A6: Send Clear_req
A7: Send message
A8: Wait for Credit
A9: Send Credit
A10: Set Clr_req and Clr_received
A11: Record Credit
A12: Accept message

Fig. 7.6.3-1: Matrix representation of a finite state machine for a transport protocol

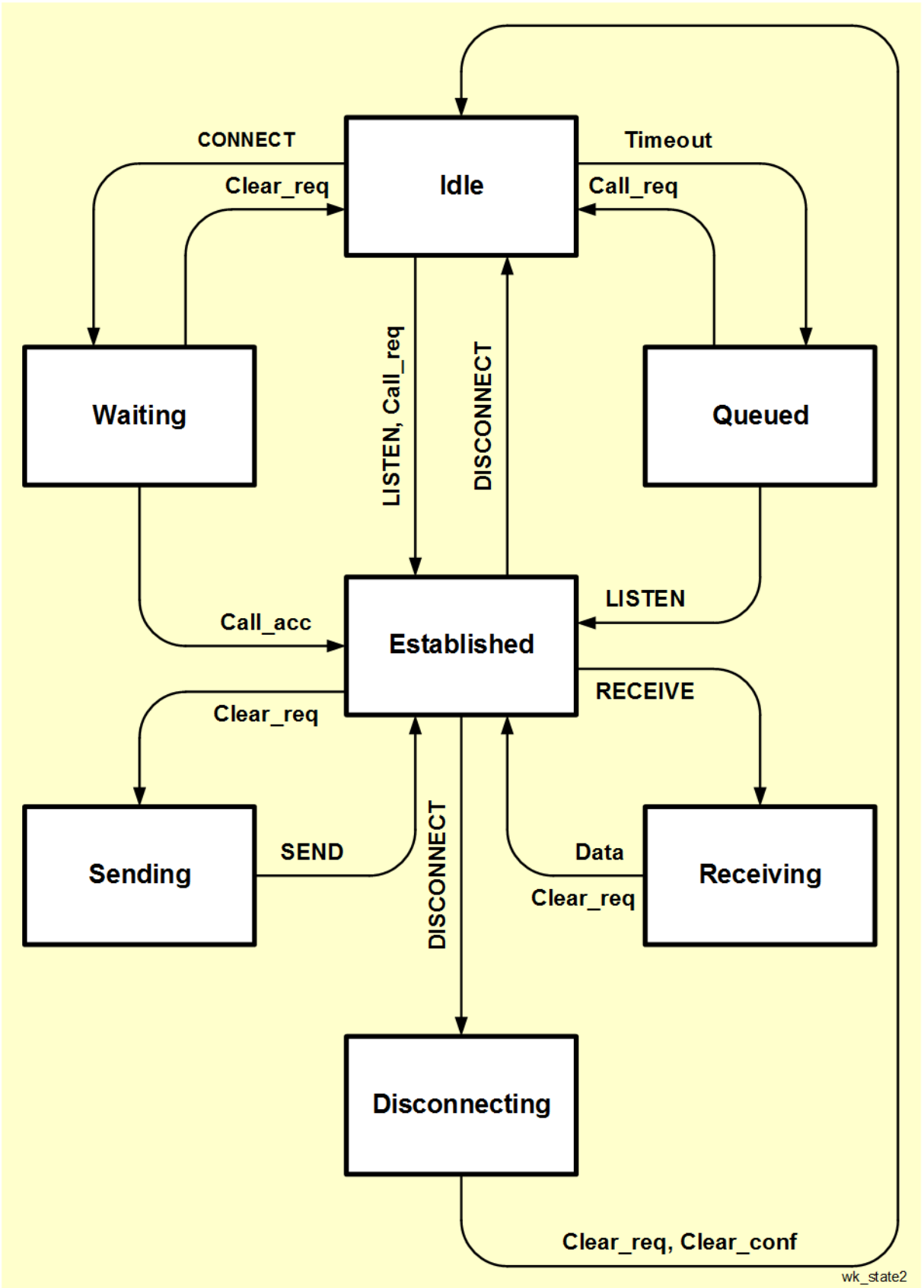


Fig. 7.6.3-2: Finite state machine flow diagram for a transport protocol

Transitions that do not change the current state are suppressed.