1 Simple Multiplexing Protocol (SMP)

1.1 Architectural Goals

Provide a process-to-process multiplexing on a best-effort basis. The protocol/module runs on top of Ethernet protocol. The resultant network stack for an end host that deploys the SMP protocol is shown in the figure 1.1.

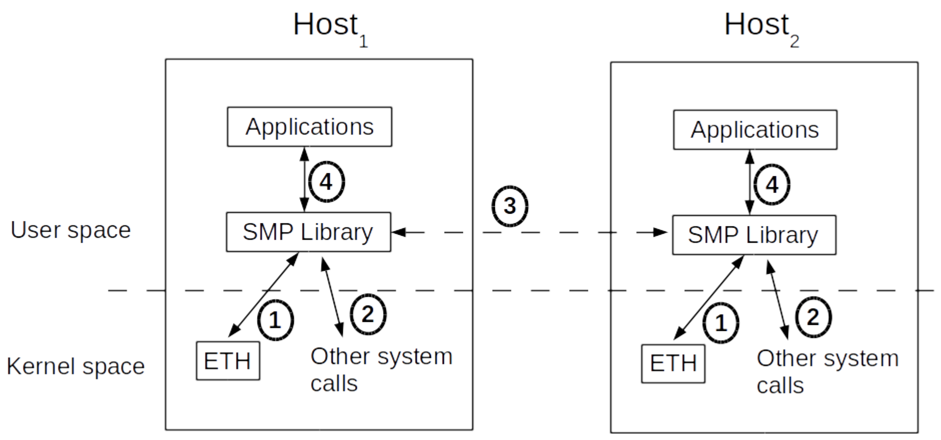


Figure 1.1: Location of SMP in proposed network stack

1.2 Educational Goal

Learn the design and engineering process of protocol development. We will design a UDP-like protocol and learn the process of protocol development.

1.3 Design of Protocol/Module

Protocol Machine for SMP

The protocol machine (PM) specifies the phases of protocol operation and also demarcates the work that needs to be done in each phase. The protcol machine for SMP, profile 0 is shown in figure 1.2.

An instance of SMP PM shall be in **Start** state after initialization. When the application uses *socket_smp()* inteface, the *socket_smp* input is generated for the PM and results in state transition from **Start** state to **Associate** state. The PM transitions back to **Start** state from **Associate** state when the application issues a call to *close_smp()* inteface. The transition to **Send** state from **Associate** state is triggered by application that wants to send a packet. The application gives payload to *sendto_smp()* interface and thus helps generate *sendto_smp* input to PM. Once the transfer PDU is constructed and sent in **Send** state, PM returns to **Associate** state. The *pdu_sent* input to SMP PM is given by the ethernet protocol machine.

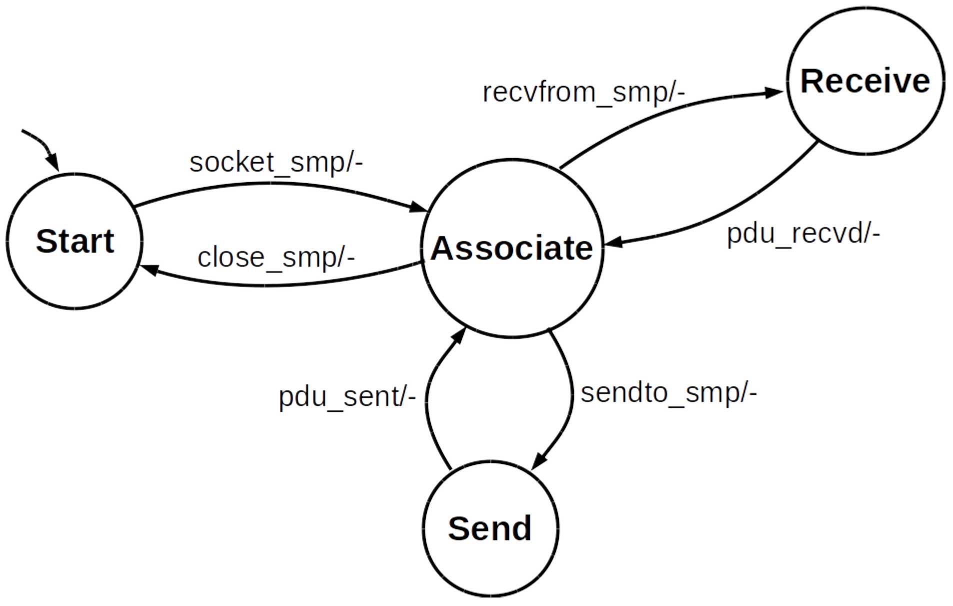


Figure 1.2: Protocol machine for simple multiplexing protocol, profile 0.

Similarly the transition from **Associate** state to **Receive** state is triggered by *recvfrom_smp* input which is generated by an application driven usage of *recvfrom_smp()* interface. PM returns to **Associate** state after receiving the transfer PDU from ethernet.

The input *pdu_sent* is generated by the OS on completing the handover of transfer PDU to Ethernet layer. Similarly, the input *pdu_recvd* is generated by OS upon receiving a transfer PDU from Ethernet layer. By convention the transfer PDUs are not shown in protocol machine. Had the SMP protocol layer incorporated any signaling information to peer SMP layer, those flags would have become outputs of the protocol machine. Since, SMP has only transfer PDU without any flags, we do not have any outputs shown in the protocol machine. A transfer PDU is sent by the SMP protocol machine in the **Send** state; a transfer PDU is received by the SMP protocol machine in the **Receive** state.

Interfaces

One of the goals of the design phase is to design the interfaces with as much exactness as possible. Such clarity would enable modules (protocols) to interact according to well-known, agreed upon structure for event and data exchange. As mentioned by John Day, "A good interface can hide lot of sins."

There are four interfaces to the protocol implementation. The interfaces are:

with ETH module The first interface is provided by Ethernet protocol module implementation as packet sockets. These packet sockets are accessed using standard sockets API. Some of the useful socket calls are: *socket(), sendto(), recvfrom(), bind(), setsockopt(), close(), . . .*

The nature of service (NoS) provided by ETH module to SMP protocol implementation is provided through the first interface.

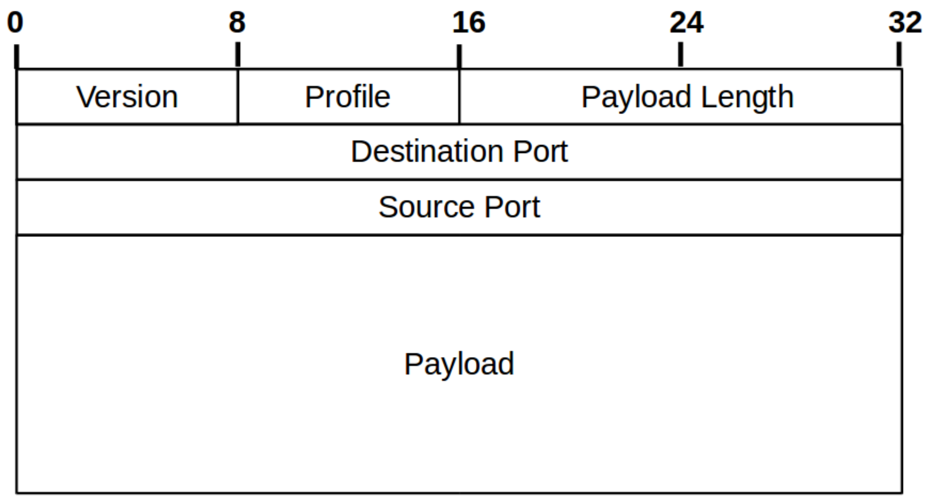


Figure 1.3: Format of transfer PDU

with System The second interface is provided by kernel through system calls. Few useful system calls are: *exit(), perror(), select(), ...* The SIGNALS are also another useful feature provided by the kernel.

with Peer Third interface is provided as interaction between peer SMP protocol implementations. Such interaction is a logical communication facilitated by ETH protocol. In theory, this interface is called protocol data unit (PDU). Since SMP is a very light weight protocol, we will only provide one kind of PDU - **Transfer PDU**. Transfer PDU helps in transfer of application payloads to the other side. The format of the transfer PDU is shown figure 1.3.

Explanation of the PDU fields is as follows.

Version Specifies the version of the protocol. The value must be 0 (zero).

Profile The profile idea is similar to profiles of RTP protocol. Profiles can be specified to tailor the protocol to any desirable type of service (ToS). At present, the valid value of profile is 0 (zero).

Payload Length Specifies the length of payload in bytes.

Ports Source and destination ports are 32-bit numbers that become part of socket <ETH,Port> endpoint specification. This socket idea is similar to TCP/IP sockets, but the format is different for SMP/ETH protocol combination. In traditional TCP/IP combination port is a 16-bit number, but here we will deal with 32-bit ports.

Payload Application data is put in this field. The data format is specific to application. The payload need not align with the 32-bit boundary.

All the header fields are represented in network byte order and are to be interpreted as left-to-right and top-to-bottom sequence.

with Applications Fourth interface is the one provided by the SMP library to the applications. Through this interface the protocol provides type of service (ToS) to higher modules / layers. The interfaces defined under this category are:

Function Name	Description
socket_smp()	creates a socket end point based on Service ID and role.
sendto_smp()	send an SMP packet.
recvfrom_smp()	receive an SMP packet.
close_smp()	close the socket and terminate the association.

SMP Interface Definitions

We will define the interfaces provided by the SMP protocol and will also define the required data types.
SMP PDU becomes a payload for ethernet. We will use the ethernet payload types reserved for experimental purposes - 0x88B5 and 0x88B6. SMP PDU shall be sent using a payload type of 0x88B5; SMP directory service, discussed in section 1.4, uses a payload type of 0x88B6.

Listing 1.1: Required data types and their definitions

```
#define ETH_SMP_TYPE 0x88B5
#define ETH_SDS_TYPE 0x88B6

//specifies the role a communication endpoint is likely to play
enumerate role {
    CLIENT = 0,
    PEER = 1,
    SERVER = 2,
    TRACKER = 3,
    PROXY = 4
};

//each SMP end point is identified by an application level−unique serviceID
//and the role played by the end point
struct smp_ep {
    role r;
    char serviceID[10];
};

//each SMP end point is temporarily mapped to a socket−like combination of
//<ETH address, port>
struct smp_ep_eth {
    char eth_addr[6];
    uint32_t port;
};
```

sock_smp() is used to create the SMP socket.

Listing 1.2: Declaration of socket_smp()

```
int socket_smp(const struct smp_ep *ep, socklen_t addrlen);
                Returns: socket descriptor of end point on success,
                           −1 on error, −2 on duplicate request
```

smp_ep structure contains *serviceID* and *role* fields. *serviceID* contains a case-insensitive character string specifying the service identifier. The service ID is a fixed length string of 10 characters. The service identifier must be unique in the context of applications (users of SMP); *r* defines the role of an end point. If two or more applications request access to same *smp_ep*, the duplicate *socket_smp* request fails with a return value -2.

close_smp() is used to close an existing SMP socket.

Listing 1.3: Declaration of close_smp()

```
int close_smp(int smpfd);
                Returns: 0 on success, −1 on error
```

sendto_smp() is used to send a packet, ie, handover the payload to SMP protocol implementation; recvfrom_smp() is used to receive a packet, ie receive transfer PDU via ETH protocol, and handover the payload to application.

Listing 1.4: Declaration of sendto_smp() and recvfrom_smp() calls

```
int sendto_smp(int smpfd,const void *buf,size_t bytes,
                    const struct smp_ep *to,socklen_t addrlen);
int recvfrom_smp(int smpfd,void *buf,size_t bytes,
                      const struct smp_ep *from,socklen_t *addrlen);

                Returns: number of bytes written or read on success, −1 on error
```

Among the parameters, *smpfd* is the socket descriptor for the SMP socket created through *socket_smp()* socket call. *buf* points to payload of transfer PDU to be sent or received. *bytes* contains the size of the buffer, ie, number of bytes to be sent or received. *to* contains the destination address and *from* contains the source address. *addrlen* specifies the size of the *smp_ep* address.
The receiving SMP protocol knows the <ETH address,port> of the remote peer. But, the SMP protocol has to provide an *smp_ep* structure to the application through *from* arguement. In order to satisfy this requirement, SMP protocol generates a private service ID in the range zzzzzaaaaa to zzzzzzzzzz. The corresponding role field in the *smp_ep* data structure may be ignored. In general the process receiving a payload through *recvfrom_smp()* function call will copy the *from* address structure to the *to* arguement of the matching *sendto_smp()* function call.

1.4 SMP Directory Service

The client applications specify an *smp_ep* structure. The structure contains service ID and role for creating an SMP connection. SMP protocol must resolve *smp_ep* to an Ethernet address. The SMP directory service (SDS) provides such a service. SMP protocol upon receiving an unknown *smp_ep* structure will use the *smp_directory()* call to resolve the *smp_ep* to an Ethernet end point address. Users of SMP, i.e., applications, have no need to use *smp_directory()* function call.

directory_smp() is used to resolve <ETH address,Port> from *smp_ep* address.

Listing 1.5: Declaration of directory_smp()

```
int directory_smp(const struct smp_ep *ep, socklen_t addrlen,
                  char *smp_ep_eth, socklen_t ethlen);
          Returns: 0 on success, −1 on error
```

smp_ep contains an SMP end point identified by service ID and role. *addrlen* specifies the size of *smp_ep*. *smp_ep_eth* structure specifies the identification of end point socket. The size of *smp_ep_eth* is specified in *ethlen*.
The *directory_smp()* function call makes sense only for identifying remote services provided by servers/peers/trackers/proxies. If an application wants to play the role of a client, then clearly remote resolution through SDS is unnecessary.
SMP directory service (SDS) is a centralized directory service for each LAN; one SMP directory server runs at a well known location to provide directory resolution service. Each SMP/ETH stack is configured with the ETH address of the SMP directory server. SDS directory server provides resolution from <service ID, role> to <ETH Address, Port>. A typical message sequence diagram for address resolution is shown in figure 1.4.

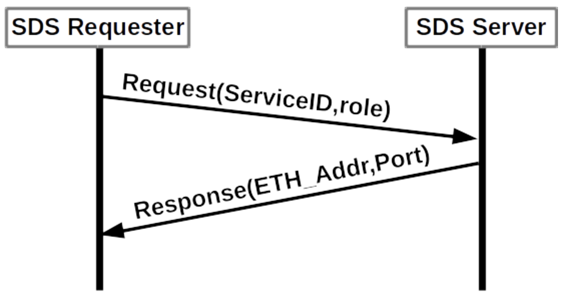


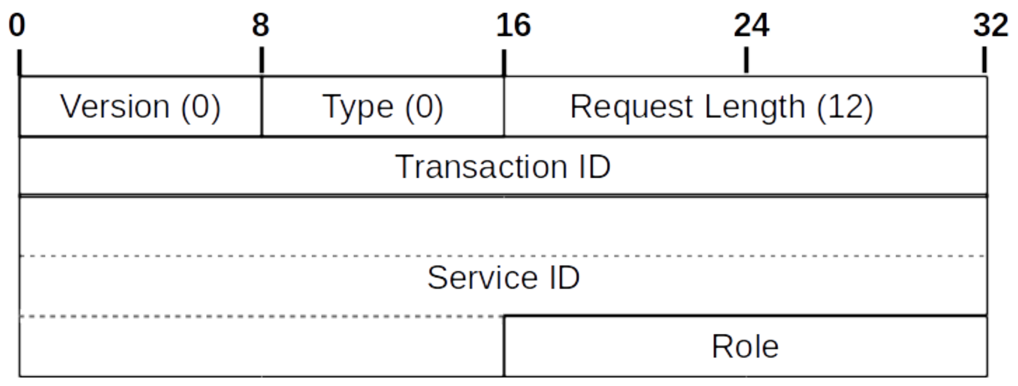
Figure 1.4: Message flow during a typical SMP directory resolution request

The PDU formats for SDS request and response are shown in figure 1.5.
Explanation of the PDU fields is as follows.

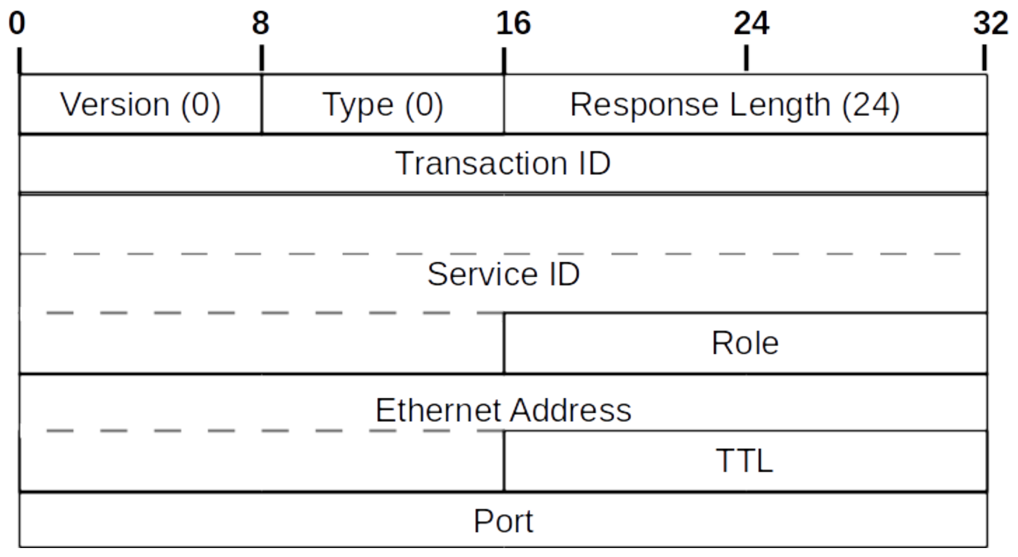
Version Specifies the version of the protocol. The value must be 0 (zero).

Type Type differentiates between SDS directory resolution and EP port mapping requests. For SDS directory resolution, the type value is 0 (zero).

Request Length Specifies the length of request payload in bytes. This value should be 12 bytes.



(a) SDS Request PDU



(b) SDS Response PDU

Figure 1.5: Format of PDUs used in SMP directory resolution process

Response Length Specifies the length of response payload in bytes. This value should be 24 bytes.

Transaction ID Uniques identifies a request and response pair. The requester should generate this randomly and make sure that this ID is unique among all the pending SDS requests from the send host. The transaction ID in response should match the ID provided in request. This field should help mitigate the denial of service (DoS) attacks.

Service ID A ten character service identifier provided to SMP protocol by the application. The characters come from ASCII code (0 - 255). The characters must be restricted to alphabets and numbers. The alphabets are treated in case-insensitive manner.

Role Role an end point is likely to play. Valid values are:

1	PEER
2	SERVER
3	TRACKER
4	PROXY

You should not send or receive a PDU with CLIENT role.

All the fields are represented in network byte order and are to be interpreted as left-to-right and top-to-bottom sequence.
Most of the fields in SDS response PDU are same as request PDU. In fact, SDS directory server can append **ETH Address**, **TTL** and **port** fields to the request PDU and send the new data structure as response PDU. **ETH Address** is the ethernet address of the host providing the service. **Port** is the 32-bit port at which a service end point is located. **Time to live (TTL)** field specifies the time validity of response. The requesting host / end point can cache the answer for TTL time. If a host is interested in using a service beyond TTL specified in response PDU, the host must send a new SDS resolution request. This feature helps in providing mobility of services across hosts.
The mechanism for polulating SDS server is left as open problem. You can use out-of-band means or manual configuration option to populate the SDS server with the information of all the services available on the network.
Each SMP/ETH compliant stack must have an independent EP port mapper process that runs all the time. This EP port mapper process is responsible for providing local and remote mapping of <serviceID,role> to <ETH_Address,Port>. EP port mapper process can maintain a local cache as shown in figure 1.6.

ServiceID	role	ETH Address	Port	TTL	Authoritative

Figure 1.6: Tabular format for cache of EP port mapper

The **serviceID** corresponds to 10-charater string indicating the service; the **role** is the role enacted by an end point; **ETH address** is one of the ethernet address of the host; **TTL** corresponds to time validity of a service. The timeout value defaults to 10 minutes. All the information obtained from local process is deemed **authoritative**.
All the local servers running in a host must register with the EP port mapper by using a local IPC mechanism[1]. If a process tries to register a service with a duplicate <serviceID,server>, EP port mapper must reject the registration and waiting process must abstain from using the rejected serviceID to enact the role of server.
The EP port mapper cache is also used to avoid conflicts of same port usage among client end points. Each time an instance of SMP protocol library creates a non-server end point, the library randomly generates a 32-bit port number. The SMP library then cross checks with the EP port mapper to make sure that no other process in the local host is currently using the same port. If the EP port mapper indicates a conflict, the random generation process is repeated until the conflict is resolved.

[1]You can use a local/UNIX socket or named pipes for this mechanism

Multiple instances of SMP libraries may be in usage at a given time. Each of these instances may use *directory_smp()* to resolve serviceID to <ETH Address,Port> mapping. In order to eliminate duplicate directory resolution, the SMP library instances may store the results of *directory_smp()* resolution in EP port mapper cache. The stores answers must have authoritative column as **NO**.
Thus EP port mapper provides three distinct functions – the storage of <ETH address,port> for all servers as authoritative records, maintain unique usage of port numbers with in a host, and cache the SDS resolution results.

1.5 Engineering the SMP Protocol

Engineering deals with implementation of the protocol / protocol machine and the protocol interfaces on a specific platform. Thus engineering deals with considering the facilities extended by the environment to complete our work. The specific implementation guidelines given here are tailored to Linux environment.
We can use packet sockets facility extended by Linux to implement SMP protocol on top of IP protocol. The behaviour of Linux packet sockets has been explained in Chapter 29: Datalink Access, UNIX Network Programming, Volume 1, 3rd Edition by W. Richard Stevens et al. You can also refer to *$man 7 packet* on any Linux machine for more information on packet sockets. You need to use **SOCK_RAW** as the socket type for packet sockets. This option enables construction of an ethernet frame header with custom ethernet type field.
During the initial development stage, you can use a local text file (database) as a proxy for SMP directory service (SDS) and end point port mapping service. You can use a text file of the following format.

```
service ID,role,ethernet address,port,TTL
```

the caller of *directory_smp()* provides the first two fields which act as a primary keys and provide a unique <ETH address,Port> with a TTL. This matching / query retrieval can be performed as string matching on all lines of the port database file. The values defined so far are:

ca4433b4ad,client,<your_ethernet_NIC_address>,1000,3600
ca4433b4ad,server,<your_ethernet_NIC_address>,1001,3600

Using Userspace Libraries

Those using user space libraries like libpcap/libnet/pcapy have to store a mapping of file descriptors to <serviceID, role, source ETH address, src port, dst ETH address, dst port>. The incoming SMP PDUs contain remote socket end point information, <ETH address, port>. This information must be translated to a corresponding file descriptor. The payload handedover by the application using a file descriptor must be mapped to corresponding <src ETH address, src port, dst ETH address, dst port> entries.