Design of a Session Layer Based System for Endpoint Mobility

Prototype System Design, Implementation and Evaluation

PETTER ARVIDSSON
MICAEL WIDELL

Master of Science Thesis
Stockholm, Sweden 2006

ICT/ECS-2006-109

Design of a Session Layer Based System for Endpoint Mobility

Prototype System Design, Implementation and Evaluation

PETTER ARVIDSSON
MICAEL WIDELL

Supervisor
Dr. Yuri Ismailov
Ericsson AB

Examiner
Assoc. Prof. Vladimir Vlassov
ECS/ICT/KTH

Master of Science Thesis
Stockholm, Sweden 2006

ICT/ECS-2006-109

Abstract

Traditional mobility is highly focused on computers moving from one network to another. This is a very narrow-minded way of thinking about mobility. Mobility might just as well be some *documents* moving from one computer to another, or a computer moving in *time*. A computer might be disconnected from a network for a long time and then reconnected to the very same network. There has been no mobility from the network's point of view, but we have had mobility in time. This calls for a major change in how we look at mobility.

Therefore we have introduced a new design where any object can be mobile, in both time and space. We call these mobile objects *endpoints*. They can be devices, documents, users or anything you can imagine, and they are named with globally unique names which are mapped to their current positions on the network. In the old IP address system, names act both as identifiers of computers and descriptions of how to reach them. We have separated the name of an endpoint from the information about how to reach it, so the routing information can be updated at any time. Now objects can move anywhere in the network, and always be reached with their name.

But we want to do more than this. We have created the possibility to establish a *session*. A session is a persistent connection between two objects. Whenever an object is reconnected to the network any applications using a session for communication will automatically continue.

We have implemented and tested this design and we have shown that it is possible to achieve this high degree of mobility. We clearly see that the new possibilities that open up with a system like this are nearly unlimited.

Contents

List of Figures

1 Introduction

Until recently, Internet relied on static mappings between users, computers, network and link addresses. It resulted in a reliable, high performance and well functioning infrastructure for naming, name resolution and routing in the "traditional" Internet. However, considering the increasing degree of network heterogeneity and a shift towards dynamic networking and mobility among networks, the usage of only the existing names is not enough in order to efficiently and simply fulfill new requirements put on the networks today.

As mobile devices that are able to connect to the Internet grow in popularity, new usage patterns emerge. It is getting more common for a computer to be connected to the Internet from several different geographical positions, as well as from several different network addresses. These changes in user behavior calls for mobility support from the underlying communication systems.

Our basic definition of *mobility* is that a computer attached to a network changes its network address, and therefore possibly its network attachment point. Mobility taking place induces problems with today's Internet infrastructure, since computers are identified and addressed with their network address. We need a way to name a computer independently of its current network address, as well as a way to contact the computer using this name.

When a computer changes its location on a network, ie. mobility takes place, there is a need to continue communication where it was stopped before the change in location. Therefore there is a need for ways to suspend and resume connections.

Definition *Suspend* means to pause data communication during a communication session, without in any way destroying the session.

Definition *Resume* means to continue data communication on a previously suspended communication session.

There is also a need for ways to relocate connections to use new interfaces, network addresses and transport addresses. There is a vast amount of work already done in this field, but it is far from mature. A lot more work is needed before a user may do simple things such as bringing his laptop from work to home without loosing any connections.

1.1 Problem statement

The goal of this master thesis project is to design and implement a *Session Layer* in the OSI stack [3], that provides a new way for applications to initiate, suspend, resume and relocate communication sessions. It must be emphasized that the main goal of the project is not to simply provide mobility support for computer networks. We want to make it possible to name devices, documents and users in a way such that they all can be completely mobile.

1.1.1 Session layer

Instead of creating connections when we want to initiate communication, we create sessions. A session is very similiar to a connection (ie. a Transmission Control Protocol (TCP) connection). The advantages of sessions are that they are not statically bound to network addresses, transport addresses or protocols like traditional connections. Sessions can be changed dynamically at any time. Sessions are also more persistent than sockets. If a session looses network connection, it will not be destroyed after a while like a traditional connection.

If a network interface is lost, the session will automatically be suspended. As soon as the device on which the session resides has some form of network access, the session will automatically be resumed without any loss of data.

Users might want to define their network connection preferences. For example, a user might want to have as low latency as possible, without taking bandwidth into account. Another user might only want to use Wireless Local Area Network (LAN), even if wired connections exist. For that reason a policy management system is needed. This system decides when and how to change between different network interfaces, connection types and protocols when network conditions change.

1.1.2 Naming system

We will need a new naming system for these sessions. We cannot use network and transport adresses to identify sessions, as these addresses could be changed during communication. So there is a need to develop a new naming system to globally and uniquely identify sessions and the endpoints they are bound to.

Another important part of this project is that we do not only want to name devices. We want to be able to give names to information entities, like a collection of documents. In this way we will be able to move information entities between computers and networks, and it will still be accessed with the same name. Also we want to give network names to persons, so that we can contact a person through the same endpoint address, independently of what country he is in, what computer he is using, or what network address this computer has at the moment.

1.1.3 Requirements

To summarize the requirements above, we will need to have the following functionality.

- A Session Layer between the transport layer and application layer.
  - This Session Layer should be designed and implemented.
  - Full support for mobility, ie. when moving between networks and locations, the Session Layer should seamlessly update the sessions so that the connections are not lost.
  - Support for suspend and resume; the application should be able to pause the communication through a session and later resume it.

  - The sessions should be at least as secure as the sockets of today are.
- A naming system for the endpoints to which sessions are attached.
  - Should define globally unique names for sessions and the endpoints to which they are attached.
  - Software for this naming system should be designed and implemented to work seamlessly with the Session Layer.

1.2 Expected results

We expect to have a working prototype of the Session Layer, complete with a accompanying naming system, meeting the requirements stated in section 1.1.3. Along with this prototype we expect to have well designed state machines for the Session Layer, and complete specifications of the protocols used in the Session Layer and its naming system.

1.3 Evaluation strategy

For this type of software implementation, we think the best way to evaluate it is by making use cases. The degree of success of the implementation will be determined by how well the requirements defined by the use cases are met. The use cases should cover different everyday scenarios as well as odd scenarios and boundary tests.

2 Background

2.1 Important concepts

Before discussing the related work in the area of mobility and the Session Layer, explanations of a few important concepts are given below.

Proxy
A *proxy* (or proxy server) is a host that in some way forwards data between two other hosts. In the context of mobility, proxies are often used as gateways to the Internet, to make mobile devices appear to have static IP addresses.

Home agent
A *home agent* is a form of proxy that mobile devices can connect through. Hosts can always reach a mobile device through its home agent. When the mobile device moves and changes IP address, it will only have to inform its home agent, so that the home agent can redirect the traffic to the mobile device at the new IP address.

Triangle routing
A problem that will arise when using home agents, is that all packets destined to the mobile device will have to be routed through its home agent. If the mobile device is communicating with a host that is geographically close, but its home agent is far away, the network performance could be very bad because of the triangle routing.

2.2 Protocol stack

When designing network protocols there is often a need to put one protocol on top of another. This is called protocol stacking. When several protocols are stacked on top of each other they will form a protocol stack. An example is the Internet as it is structured today.

2.3 OSI Model

The OSI Model[3] is an abstraction of a protocol stack, see figure 1. It consists of seven layers. At each layer there is a definition of the services that a protocol at this layer should implement. The idea is that a layer should only use functionality provided by the layer below it in the stack, and only provide functionality to the layer above it. If the functionality provided between layers is standardized it is possible to use implementation of different layers from different vendors. This gives a very flexible type of network in which most devices would be able to communicate with eachother. We will now briefly describe the different layers in the OSI Model.

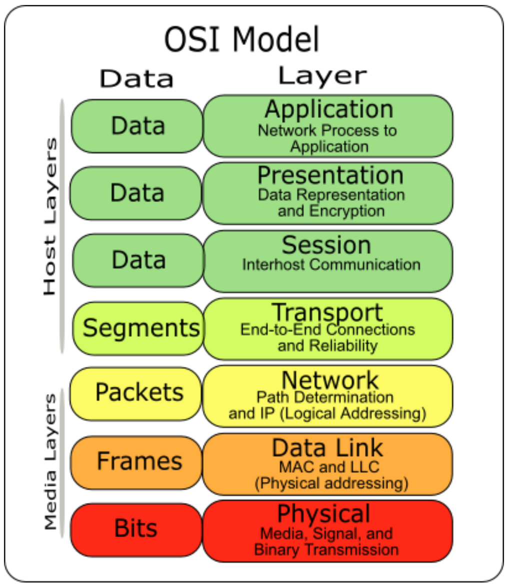

OSI Model
Data
Layer
Host Layers
Media Layers
Data
Application
Network Process to Application
Data
Presentation
Data Representation and Encryption
Data
Session
Interhost Communication
Segments
Transport
End-to-End Connections and Reliability
Packets
Network
Path Determination and IP (Logical Addressing)
Frames
Data Link
MAC and LLC (Physical addressing)
Bits
Physical
Media, Signal, and Binary Transmission

Figure 1: The OSI layer model. Picture from Wikipedia, `http://en.wikipedia.org/wiki/Image:Osi-model-jb.png`, licensed under GNU Free Documentation License.

2.3.1 Application Layer

This layer should provide the functionality needed by the user to access information on the network. The user will typically use an application that will use the functionality provided to deliver the information requested. An example is the web browser that uses the Hyper Text Transfer Protocol (HTTP) to present Hyper Text formatted web pages for the user.

2.3.2 Presentation layer

The Presentation layer should provide functionality to "identify transferable syntaxes", "select transfer syntax" and access services of the Session layer (which is directly below the Presentation layer). The idea is that this layer should decide how the information provided by the Application layer should be formatted.

2.3.3 Session layer

The Session layer should provide the following functionality to the Presentation layer.

Session-connection establishment
The Session Layer should enable two presentation-entities to establish a

session-connection between them. The presentation-entities are identified by session-addresses, and both sides negotiate session parameters.

Session-connection release
The session-connection release service allows presentation-entities to release a session-connection without loss of data.

Normal data transfer
The ability to send data between presentation-entities.

Token management
Allows the presentation-entities to control explicitly whose turn it is to carry out certain control functions.

Session-connection synchronization
The presentation-entities should be able to define and identify synchronization points and to reset the session-connection to a defined state and agree on a resynchronization point. The Session Layer is not responsible for any associated checkpointing or commitment action associated with synchronization.

Exception reporting
The Session Layer should provide exception reporting to inform the presentation-entities of exceptional situations.

Activity management
The user of the Session Layer should be able to divide logical pieces of work into *activities*. A session could span several activities, and these activities can be interrupted and then resumed.

2.3.4 Transport layer

The Transport layer should relieve the Session layer from the task of actually deciding how to deliver the data. This can be done in different ways and it is important to separate connection and connectionless Transport layer services.

Connection services

Transport-connection establishment
The Transport layer should enable two session-entities to establish a Transport layer connection. Session-entities are identified by a transport-address.

Transport-connection release
It should be possible for a session-entity to release a transport-connection and this should inform the correspondent session-entity.

Data transfer
The Transport layer should provide the means for two session-entites to transfer data between them.

Expedited data transfer
The transport layer should provide functionality to send urgent data that will be allowed to bypass the ordinary data stream.

Suspend facility
The Transport layer should make it possible for two session-entities to suspend a transport-connection.

Connectionless services The Transport layer should provide functionality to deliver data between session-entities, but it does not give any guarantee for delivery or correct ordering of the data. Session-entities are identified by the transport-address.

2.3.5 Network layer

The network layer should deliver data between transport-entities. Those entities are identified by the network-address. This means that the the network layer is responsible for routing the data to the correct network-address. This can be done both by conctions or connectionless data delivery. The difference between the two modes are very much the same as on the transport layer. The Network layer should also map between network-addresses and data-link adresses. It is also responsible for error detection and it should report any errors to the Transport layer.

2.3.6 Data-link layer

The Data-link layer should provide functionality for the Network layer to use connections between different netwrok-entities. Different network-entities are identified by a data-link address.

2.3.7 Physical layer

The physical layer should perform the actual delivery of data. It consists of connections between different data-link entities which are used to transfer data between them.

2.4 The Internet Protocol Stack of Today

Today, the protocol stack used for communication over the Internet uses Internet Protocol (IP) as the network layer protocol. On the transport layer, the two most commonly used protocols are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). The network address used by IP is called IP address. It is a 4 byte number identifying a host, and also providing information about the route to that host. The transport address used by TCP and UDP is called *port number*, which is an integer defining which application to connect to on the host at the current network address.

2.4.1 TCP

TCP is the most commonly used transport layer procotol on the Internet today. TCP provides functionality to the layers above to create connections between two hosts. When a connection is established, *TCP packets* containing data can be sent over it. TCP emulates connections over IP packets, the IP protocol itself is not connection oriented.

2.4.2 UDP

UDP is used to send packets (datagrams) between two hosts. The protocol is not connection oriented, and it does not guarantee that a packet that is sent reaches its destination.

2.5 Related work and existing solutions

In this section we will briefly describe other efforts to achieve our goals. Some of them are complete systems, others just solve some specific problems. This is what we will later base our design decisions on. We look at the definitions of the Session Layer, mobility on different layers in the IP stack and different types of connection management.

2.5.1 At what layer does mobility belong?

There have been a lot of discussion about what layer or layers in the OSI model (see figure 1) is suitable for holding support for mobility. Wesly [17] discusses this important question in a paper that is summarized below.

Placing mobility support in the network layer has the benefit that it is in the middle of the hourglass model[1], and therefore it can benefit every higher layer using IP. The downside is that we cannot escape triangle routing, which can be a serious problem under some circumstances. Another disadvantage is that the mobile node's home agent becomes a single point of failure. Also, the communication between IP and higher layer protocols is currently not rich enough to provide the higher layers with information about mobility taking place, which would be a very useful feature.

Transport layer mobility is a rather good solution, since it does not need any changes to the infrastructure of networks. We do not need any home agents or home networks, nor any changes in infrastructure besides having access to DNS and DHCP services.

Session Layer mobility has basically the same advantages as transport layer mobility, but we do not have to make any significant changes in how the transport protocols work. The major downside of a Session Layer implementation is that it requires changes in applications.

As a sidenote, it can be argued that the need for changes in applications is a bigger problem than the need for changes in the operating system. If the

[1]The IP stack can be seen an hourglass with IP in the middle, holding it together. You can use any protocol on top of or below IP.

operating system is changed in a way that does not affect the API towards applications, it is very simple to install an upgrade to the operating system, and then continue using the old applications as before. If a change is needed in every application, they must all be recompiled. This can be very hard to realize, since every developer of closed-source applications must be convinced that the changes should be made.

2.5.2 Network layer mobility

As stated above implementing mobility in the network layer has its advantages and disadvantages. The standard for IP mobility, Mobile IP, is implemented in this layer, but there are also other solutions.

Mobile IP Mobile IP [8] is one of the simpler approaches to mobility. When a host is mobile it will relay all incoming traffic through a home agent. This home agent has a static IP address and is always possible to reach. Every time the mobile host moves it will update its home agent. This ensures that the home agent will forward traffic to the new address. When the mobile host sends a packet it will send it to the host but change the source address to be the address of its home agent. This way the communication will look completely normal from the other host's point of view.

Internet Indirection Infrastructure (i^3) The i^3 [14] approach to mobility works like an event system. The user registers for a specific type of messages, and also sends messages of a specific type. All services that have registered for a specific type of messages will get all messages of this type. Therefore the destination of a message is not defined by an IP address, but by a unique type identifier.

Messages will be of the form (ID, data). All messages will be sent through i^3 enabled servers. A request for a message will be of the form (ID, IP). Only one server is responsible for an ID, which means that all messages with that ID will be forwarded to this server, which will have a list of recipients. This makes it possible for an application to form any type of overlay network and the system also supports connection migration.

The i^3 architecture is a lot more complicated than the summary above, and it also has some more advanced features like ID stacking. Those features are not really interesting for our work as we are mainly looking for a way to achieve mobility and connection migration.

The HIP project The basic concept of the Host Identity Protocol (HIP) [6] is that a host should not be identified by an IP address but by a Host Identifier (HI). The HI should be a public key so that a host can easily know that it communicates with the same peer all the time. With this in mind, there is a simple solution to mobility. Since the connection is identified by the HI (along with the ports), a reconnection is still possible after a change of IP address.

2.5.3 Transport layer mobility

As stated in section 2.5.1, transport layer mobility has some advantages over network layer mobility. Therefore a lot of research has been done in this area and we will here present some of the different approaches.

Reliable Network Connections Zandy et al. [18] present two systems for implementing transparent mobility and disconnection handling. They are transparent in the way that applications do not have to be rewritten or recompiled for the mobility and disconnection handling to work. The applications can still behave as if they were using conventional sockets.

The first system is reliable sockets called *rocks*. They do not require any changes in the system kernel, but are linked in to replace the standard sockets API. This system is backwards compatible with conventional sockets, in the sense that they can detect if the other peer has the extended rock functionality. If it has not, the local rock can revert to behaving like a conventional socket. The basic idea with rocks is to keep endpoints open when network failures occur, or if the other host changes its IP address. They also have functionality to suspend and resume connections.

The second system is reliable packets called *racks*. They have functionality similar to rocks, but they are implemented as a packet filter instead of replacing the existing socket API. This packet filter monitors the packets flowing into and out from user space, and performs necessary modifications of these packets. It also inserts new packets when needed.

Migrate The Migrate project [13] focuses on TCP connection migration, and how that can be used to enhance mobility. The approach is to add an extra option to the TCP header, namely the migrate option. When this option flag is present in a SYN message and the server knows about migrate, it will send a SACK with the migrate option set. This assures that this implementation is completely backwards compatible with old TCP implementations. In the three-way handshake of two migrate enabled hosts they will also use parts of the TCP header for communicating a pair of public keys. Those keys will later be used to identify the other peer when it resumes the connection. When a connection is lost TCP saves all states and waits for a migrate packet. This packet is validated with the public keys, and if it came from the right peer the connection is resumed at exactly the state it was terminated in. This means that communication is started from the last acknowledged byte in the TCP window.

2.5.4 Session Layer mobility

Session Layer mobility has the same advantages over network mobility as transport layer mobility. It can also assure that we do not change the underlying transport protocols. On the other hand, inserting a new layer between application and transport layer will need every application to be modified and recompiled.

Robust TCP Connections for Fault Tolerant Computing The robust TCP socket is a way to make TCP connections invulnerable to network or link failures. Ekwall et. al. [1] implemented a Session Layer in Java, which has the same API as the normal Java sockets. This Session Layer reconnects TCP connections if they go down, without any loss of data. The application using this library will not notice anything, it will seem as if the connection was never lost.

A mobile TCP socket Qu et al. [9] explain how they have implemented a mobile TCP socket using some small modifications to the socket implementation. The basic idea is to put a layer between the application and the socket. This layer will be responsible for keeping the connection open when the TCP connection goes down. Therefore it is also responsible for opening the connection when the other peer comes up again.

From the application's point of view this socket will always have the same IP addresses both local and remote. This will assure backwards compatibility with the BSD socket API. The mobile socket will also buffer the data sent by the application so that it may resend it if the underlying connection goes down. It also keeps a server socket open on the same port as any connection, so that it may receive resume messages from connection peers that has lost and resumed their connections.

SLM The Session Layer Mobility (SLM) [4] approach to mobility is backwards compatible with the existing sockets as it does not need any changes to the current IP stack. Mobility is achieved by using proxies on both sides of the communicating link. Therefore the applications will only connect to their local proxy and not directly to the other host. This way all mobility is managed between the proxies, hiding any mobility from the application. Naming is done by adding an extra naming server on top of DNS. This server is to be asked about movements of the user.

MSOCKS+ MSOCKS+ [5] is an architecture which aims at providing mobile devices the means to change network addresses and the NICs used during connections. It also provides a way to specify which traffic that should go to which interface.

This is done via proxies. The proxy server should be placed between the mobile devices and the rest of the Internet. When a mobile device wants to initiate a connection, it connects to the proxy server providing information on which host and port number it wants to connect to. The proxy server initiates a new connection to the static host, and thereafter forwards all (even acknowledgment and synchronization) packets between the hosts. The two end-points will see these two connections as one single connection. This is called the *split connection model*.

It is assumed that the "static host" never changes IP address, so only the mobile host will need a library to handle reconnections and mobility. The MSOCK

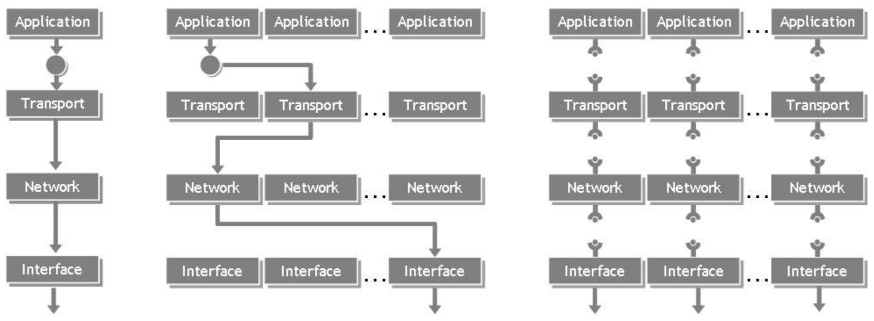


Figure 2: The matrix stack paradigm.

library is placed as a Session Layer between the application and the traditional TCP socket.

If the mobile device moves to another location (another IP address) or to another interface, it must reconnect to the proxy using a reconnect function in the MSOCK library. The proxy sees the reconnect message, and un splices the two previous connections, replacing the old mobile/proxy connection with a new one. The static host will not notice anything, and sees it as the same end-to-end connection is still open.

2.5.5 Multi layer mobility

Some projects handle mobility at several layers. This way one can achieve a lot more flexibility in how the mobility is done. At the same time this increases the complexity of the system.

Mobility Support for Networked Applications built in the TCP/IP Stack In Wang's master thesis report[16] it is explained how to modify the Linux TCP/IP stack to make it dynamic in the bindings between different protocols at different layers. The goal is to make the network stack (and therefore also the socket) fully dynamic, so that the IP address, TCP port and IP protocol version can be changed at any time during an established connection. This way it can react to any changes in the underlying network, even theoretically change the transport protocol used, although this has not yet been implemented.

Figure 2 illustrates the evolution from the traditional stack paradigm to this new matrix stack paradigm, where no binding between different protocols in different layers are static. Everything could be dynamically rebound during ongoing connections.

The big problem with this is of course that changing the stack in this way is fatal to protocols like TCP, if the changes are not carried out the same way on both sides simultaneously. Therefore Wang suggests that a Session Layer is needed.

2.5.6 Connection management

Session Initiation Protocol The Session Initiation Protocol (SIP) [2] protocol is used for initiation of sessions. It basically provides means to transport requests from one user to another, only identifying users with their SIP Uniform Resource Identifier (URI). The URI is a name and a Fully Qualified Domain Name (FQDN), e.g name@example.com. SIP relies on SIP proxies to deliver requests to users. SIP can be used in conjunction with e.g, Session Description Protocol [12], which describes what kind of session we want to establish. Those messages can contain anything from port numbers to data compression type. When both peers have agreed upon the type of session they want, they can establish an ordinary connection to transfer data of the session. As SIP is mainly aimed at Voice over IP (VoIP) [15] this connection is often done via RTP [10, 11].

2.6 Development environment

In order to implement a Session Layer, modifications to TCP/IP stack will be needed. Therefore we will need access to the source code of a TCP/IP stack. We will also need at least two networked devices to be able to run tests of the networking capabilities of the Session Layer. To make the necessary changes in the TCP/IP stack, and to develop the code for the Session Layer, we will of course also need an editor and a compiler.

2.6.1 The TCP/IP-stack

We want to be able to test our implementation in a simple way, therefore we want some kind of user space TCP/IP-Stack. We have investigated the following two methods of getting this.

Daytona stack[2]
: This is wrapper code around the Linux 2.3 IP-stack so that it may be run in user space. The implementation seems very unstable and the code is not very well structured.

User Mode Linux (UML)[3]
: UML is a system that enables us to run a Linux kernel as a virtual machine in a host system. This is distributed as a set of patches, included in the main Linux kernel since version 2.6.9. With UML, it is possible to run several virtual Linux machines on top of one machine.

After this investigation UML was a clear winner. It seems very stable and it enables us to run several virtual machines on the same computer. Booting a UML kernel takes very little time, as compared to booting a physical machine with the same kernel. A good thing with UML compared to the Daytona stack

[2]`http://nms.csail.mit.edu/~kandula/data/daytona.pdf`
[3]`http://user-mode-linux.sourceforge.net/`

is that any changes we make can be easily integrated into the main Linux kernel. This will make real tests easier as we can run our modified kernel directly on physical machines.

2.6.2 Networked devices

During design and implementation of our work we have used ordinary PCs with Pentium 4 processors. On these machines we have run the Gentoo Linux distribution[4].

On top of this we run our UML machines. To be able to test the functionality of the TCP/IP stack, the UML machines need networking capabilities. To arrange this we have used the Ethernet bridging functionality in the Linux kernel. The network testing setup is explained in figure 3.

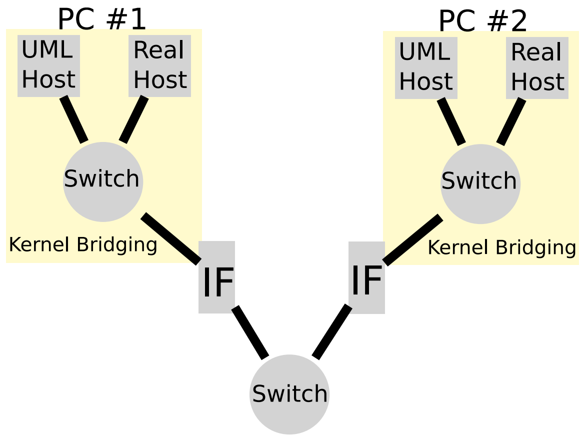


Figure 3: The development network.

Every virtual device needs its own Media Access Control (MAC) address, but they should have the same address every time they run. To achieve this we will make a MAC address every time we start UML by hashing static values that are different on different hosts. As of now we use the output from “pwd” and “uname” as input to the hash.

2.6.3 Compiler and editor

As we are working on the TCP/IP stack inside the Linux kernel we have to use the Gnu C Compiler, (GCC)[5]. The Linux kernel will not compile with any other compiler as it depends on special features of GCC.

[4]`http://www.gentoo.org`
[5]`http://gcc.gnu.org/`

2.7 Conclusions

We have seen that there has been many attempts to achieve parts of what we want to do. Several of these attempts were successful and work in a good way. However, it must be emphasized that the related work we have looked at mainly describes methods of achieving mobility in networking, while we are trying to establish a new way of thinking about connections and endpoints. The benefits of our work will go beyond the mobility techniques that have been developed so far.

The development environment we have chosen is very mature and easy to work with.

3 Design

In this chapter we will describe and define the different aspects of our system design. We will start by explaining the basic design ideas and decisions. Then we will define which *services* the Session Layer will provide to applications using it. Following this we will define the different subsystems that the Session Layer consists of. Finally we will describe the Session Layer *protocol* used for session management communication.

3.1 Basic design decisions

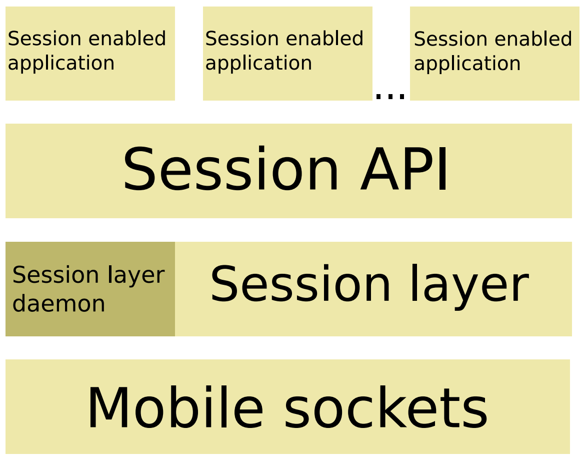

Session enabled application
Session enabled application
...
Session enabled application
Session API
Session layer daemon
Session layer
Mobile sockets

Figure 4: The architecture of the Session Layer and its control mechanisms.

This section will introduce the basic ideas and concepts which our Session Layer design will rely on. We will not only describe the ideas and concepts, but also briefly explain why we have made these decisions. The overall design of the Session Layer is shown in figure 4.

3.1.1 Endpoints

In our design communicating entities are called *endpoints*, see figure 5. An endpoint can be anything from a computer to a collection of documents. The Session Layer has the responsibility to manage endpoints in a way that enables them to be truly mobile. An endpoint is a named entity, the name is globally unique. To be useful endpoints need to provide services, such as HTTP or FTP, to other endpoints.

Those services also need to be named, but they only need to be unique within the scope of the endpoint. An endpoint might of course have any number of services. When an endpoint connects to a service at another endpoint, there is a need to create a *session* to deliver data between the endpoints. Sessions should be persistent. This means that a session should not be broken due to any event of mobility. Sessions are named with a unique identifier, which consists of two public keys in an RSA system. This assures that sessions can verify the authenticity of eachother when they are mobile. An endpoint might contain any number of sessions.

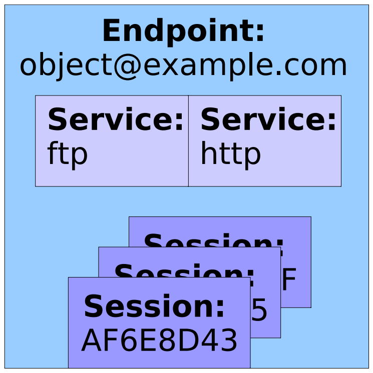


Figure 5: An endpoint is a named entity that contains the basic functionallity needed for mobile communication. They provide *services* and contain *sessions*.

3.1.2 How will the user perceive the Session Layer?

The user closest to the Session Layer will be the application programmer. Of course, normal users of the applications on a system will also perceive the Session Layer, but the way they perceive it will be dictated by the application programmer. Since we are not in control of how the programmer wants to forward information about the Session Layer to the actual user, we will concentrate on describing how the programmer perceives the Session Layer.

We have decided to design the interface to the programmer in a way that is as close as possible to the BSD socket system calls. This way the programmer will not need to learn a totally new paradigm of network programming. Of course some things will be new to the programmer, like the concept of endpoints and sessions. But sessions can basically be seen as sockets that are more persistent and are allowed to be suspended.

3.1.3 Naming and services

To achieve true mobility, there is a need for a new naming convention for resources. It is not possible to use network addresses anymore, since we can not assume that they do not change during the session. Transport addresses (like TCP ports) can also be changed during a session, so there is a need for other means of referring to a specific application at a host. This also follows from the definition of the Session Layer in the OSI model.

A name in our naming system is of the format "object@domain". Object is here the name of an object that belongs to this domain. We call an object that is named in this way an *endpoint*. At an endpoint there may reside a number of *services*. A service is basically a process listening for session establishment requests. In our new paradigm for networking, the service has big similarities to a traditional BSD socket listening on a port. Our service will still be a socket listening on a port, but we will hide the port number from the user. The domain part of a name is a standard domain name like "example.com" which can be reached using DNS. The naming system is explained in more detail in section 3.3.

3.1.4 Mobility

To tackle mobile events, the Session Layer needs support from underlying layers to change the socket accordingly. If a device is moved to a new address we need to update the socket connected to this device with new addresses. We will call this concept *rebinding*. We have found two possible solutions for rebinding, to create new connections and to use the Mobile Socket.

New connections One possible way to achieve mobility is to open a new transport layer connection every time there is a mobility event. This makes it possible to handle any mobility events as the socket can be set up in any way when it is connected for the first time. This is an easy and reliable solution.

The Mobile Socket To achieve a higher degree of flexibility for the Session Layer we may use the Mobile Socket[16]. The mobile sockets function like traditional BSD sockets, except for the extended functionality that enables the user of the socket to dynamically change source and destination network-addresses and transport-addresses. The concept of mobile sockets goes beyond that. It also enables the user to dynamically change what network layer protocol to use (e.g. change from IPv4 to IPv6).

Session Layer Mobility Directly using one of the mobility concepts described above will not work. If a user changes the connection's destination network-address in an established connection, the user on the other side must simultaneously change its source network-address if we want to be able to continue communication. This is where the Session Layer serves an important purpose. The Session Layer will administer and synchronize these dynamic rebindings of network-addresses, transport-addresses, protocols etc. adequately.

To solve the problem of synchronization when rebinding connections between different network and/or transport addresses, we will use the concept of *suspend* and *resume*. When we need to rebind a connection, we will suspend the session, meaning that we stop all data transfer immediately. When we have stopped the

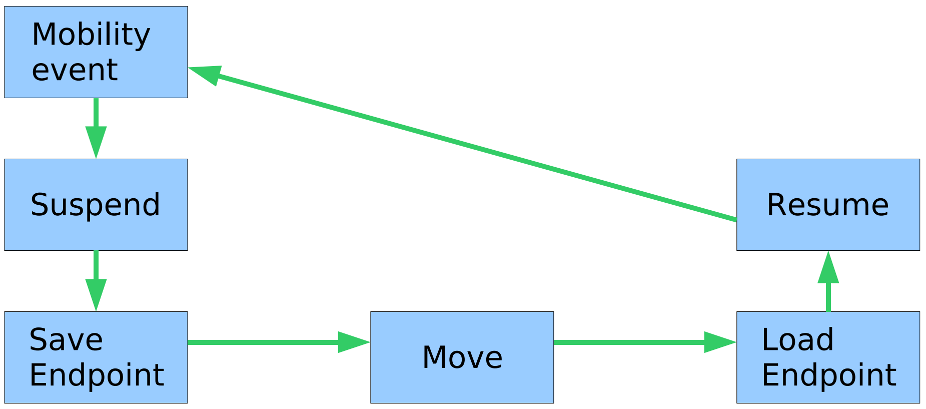


Figure 6: Session Layer mobility mechanism. This is the basic mechanism for session mobility which supports a broad range of different mobility events. For the simplest types of mobility events, e.g. network changes, all steps except resume are optional.

data transfer, we perform our rebind of the connection. After that we inform the other peer of this rebind by sending a session layer message, RESUME, to the other peer. This resume message contains information of what our new network/transport addresses or protocols are.

The other peer will now perform the necessary changes on its side (rebinding the connection). When it is done it will respond with a session layer message, RESUME_OK, which means that it is now ready to resume the data transfer.

The suspend and resume commands are not only useful when rebinding the connection. They also enable the user to pause communication at any point, for example to perform network maintenance, and then resume communication without any network-addresses, transport-addresses or protocols being changed. See figure 6 for a step by step description of the mobility concept.

3.1.5 Data integrity

Besides handling foreseen disruption of communication (e.g. the suspend command), the Session Layer must handle *unforeseen* disruptions in communication appropriately. An example of this is that a user goes out of range of his Wireless LAN access point and looses the connection. Later, the user plugs in an Ethernet cable and expects all sessions and data transfers to automatically continue without any loss or corruption of data.

To handle this scenario gracefully, we have decided to use checkpoints. The design of the checkpointing system is described in section 3.4.1.

3.1.6 Automatic rebinding and the Session Layer Daemon

In order to make the rebinding really useful, there is a need to somehow automate it. When a user has been using Wireless LAN and wants to switch to wired Ethernet, there should be no need to manually suspend the session, rebind the socket to the Ethernet interface, and then resume the session.

To make the Session Layer more useful and user friendly, we have decided to design and implement a *Session Layer Daemon* (SLD). This is a process that will monitor the status of all network interfaces on a system. It will also check for new network interfaces, and see if any interface disappears (e.g. when a laptop user pulls out a PCMCIA network card).

When the SLD notices an event that is useful information for the Session Layer (e.g. a new and faster network interface is available) it will forward this information to the Session Layer. The Session Layer will then decide if rebinding of the connection is needed. With this design approach, the user will not have to think about how to connect to the network. The Session Layer will automatically (according to a policy) change between network interfaces when conditions change.

Another important task for the session layer daemon is to automatically detect new devices other than network devices, that are attached to the computer. Imagine that you have a Universal Serial Bus (USB) Mass Storage Device with some web content. When you attach this device to your computer the content should automatically, if you want it to, be available from your computer through a HTTP server. When you want to remove this device you want it to automatically save all sessions to it so that they can be resumed when the device is attached to another computer. This is what makes the session layer go beyond traditional mobility.

3.1.7 How our design relates to previous work

Most of the other work done in the mobility area focuses on maintaining the traditional way in which applications interact with sockets, to ensure backwards compatibility. We argue that mobility requires such a big change to the traditional network thinking, that there is a need to introduce a new paradigm for handling network connections. This means we cannot escape changing the interaction between applications and the sockets.

A lot of work is done on the same layer as our work, the Session Layer, but that is where the similarities end. Most of those projects are concerned with TCP only and will not address things such as naming and on/off (resume/suspend) semantics[9, 5].

This said, our project still has a lot in common with other projects. For example the naming system we propose is very similar to the one used in SIP and the control messages in the session management protocol could just as well be SIP messages. Also the mechanism we use to achieve mobility on the lower levels, the mobile socket, is very similar to the one developed in the migrate project [13] to achieve mobility on the transport level.

3.2 Session Layer services

The Session Layer should provide the functionality needed to achieve the goals stated in the beginning of this paper.

3.2.1 Session services

This is a list of services that the session layer must provide.

For endpoints:

- Creation of endpoints.
- Destruction of endpoints.
- Provide one or several default endpoints which can be used by e.g. clients. This is to simplify for applications that do not have any special interest in device mobility and are happy to share an endpoint with a lot of other applications.
- Rebinding of endpoints, e.g. change interface.
- Load endpoints from disk.
- Save endpoints to disk.

For services:

- Creation of services.
- Destruction of services.
- Accepting a client on a service.

For sessions:

- Connect to a service.
- Suspend a session.
- Resume a session.
- Close a session.

Note that session objects can only be created out of a connection to a service. With BSD sockets, you first create the socket object, then use it to connect to another host. This is an important difference and it is also an important design decision. A session does not have a meaning unless it has been established between two endpoints. A socket on the other hand might be connectionless.

3.2.2 Use cases

In this section we will list and explain the event flow in a range of use cases. We have tried to cover most of the functionality that the Session Layer should have.

Use case 1 – Unexpected disconnection

Name	Unexpected disconnection
Summary	The system unexpectedly looses all network interfaces.
Preconditions	• A session is established. • Data is being sent in both directions.
Triggers	The local system suddenly looses all network communication capabilities, e.g. no working network interface is available.
Main path	All communication stops, the functions for sending and receiving data will block.
Alternative path	–
Postconditions	No send or receive calls will return until a new interface is available.
Notes	–

Use case 2 – New network interface, no previous interfaces exist.

Name	New network interface, no previous interfaces exist.
Summary	This use case describes what happens if no network interfaces are available, and suddenly a new interface comes up.
Preconditions	• A session is established. • No network interfaces are available.
Triggers	A new network interface is made available on the system.
Main path	1. The Session Layer Daemon notices the new network interface and forwards the event to the Session Layer. 2. Communication on all endpoints and sessions continues. All receive or send calls that the application was blocked in when the network was lost will now return successfully.
Alternative path	If the policy associated with an endpoint prohibits it from resuming, for example if the policy says that the endpoint must only use a special type of interface, the endpoint's sessions will remain suspended.
Postconditions	If the main path was executed, all communication will be allowed again without any data loss or data corruption.
Notes	What happens with each endpoint is very dependent on the policy associated with the endpoint. The policy may state that the endpoint must only use 1 Gigabit Ethernet, and if a 10 Megabit Ethernet interface is made available the sessions associated with that endpoint will remain suspended.

Use case 3 – New network interface, previous interfaces do exist.

Name	New network interface, previous interfaces do exist.
Summary	This use case describes what happens if one or more network interfaces are available, and suddenly a new interface comes up.
Preconditions	• A session is established. • At least one network interface is available.
Triggers	A new network interface is made available on the system.
Main path	1. The Session Layer Daemon notices the new network interface and forwards the event to the session layer. 2. The policy manager will check the policies for every endpoint, and decide if the endpoint should switch to the new interface or not. 3. All endpoints that have policies preferring the new network interface to the previous ones available will suspend, rebind to the new interface, and resume communication.
Alternative path	–
Postconditions	Communication will be enabled as before, but some endpoints may be communicating on new network interfaces.
Notes	As usual, the applications on the system will not have to know or care about this happening. The transition to new interfaces will not be noticed by them.

Use case 4 – Both peers simultaneously rebind endpoints

Name	Both peers simultaneously rebind endpoints
Summary	A session is established. One peer moves from one network to another, this leaves this peer without connection during a certain amount of time. During this time the other peer changes its connection to the network.
Preconditions	• A session is established. • Data is being sent in both directions.
Triggers	Both peers simultaneously rebind their sessions to new source IP addresses.
Main path	Communication continues as before, the application notices no difference.
Alternative path	–
Postconditions	To the application it appears as if nothing has happened, except for a short pause in communication.
Notes	–

Use case 5 – Application performs suspend

Name	Application performs suspend
Summary	An application suspends the session.
Preconditions	• A session is established. • Data is being sent in both directions.
Triggers	The application calls the suspend command, specifying a session to suspend.
Main path	All communication stops. Any receive or send calls that the application is in will block.
Alternative path	–
Postconditions	All communication through the suspended session has stopped.
Notes	–

Use case 6 – Application performs resume

Name	Application performs resume	
Summary	The application calls the resume command, specifying a session to resume.	
Preconditions	The session has previously been suspended.	
Triggers	The application calls the resume command.	
Main path	The session will be enabled for further communication. If the application was blocked in a receive or send call when suspending, these calls will now return successfully.	–
Alternative path	–	
Postconditions	–	
Notes	Communication through the resumed session is now possible again.	

Use case 7 – Create endpoint

Name	Create endpoint
Summary	The application wants to create a new endpoint to use for communication.
Preconditions	–
Triggers	The application calls the function to create an endpoint, providing a name that identifies it.
Main path	The new endpoint is created and can now be used.
Alternative path	If the endpoint name already exists within the current computer's domain, the function returns with an error indicating this.
Postconditions	A new endpoint with the name provided by the application now exists.
Notes	–

Use case 8 – Close endpoint

Name	Close endpoint
Summary	The application wants to destroy an existing endpoint.
Preconditions	An endpoint exists.
Triggers	The user calls the function for closing an endpoint, providing the name of the endpoint to be closed
Main path	1. All sessions associated with this endpoint are closed and removed. 2. All services associated with this endpoint are removed. 3. The endpoint itself is removed.
Alternative path	If the computer on which the application resides does not have ownership of the endpoint, nothing will be done to it and the function will return with an error stating this.
Postconditions	The closed endpoint does not exist anymore.
Notes	–

Use case 9 – Create session

Name	Create session
Summary	The application connects to a service attached to an endpoint, thereby creating a new session.
Preconditions	–
Triggers	The application calls the function to connect to a service.
Main path	1. The session layer connects to the service and negotiates the session ID for a new session. 2. The function returns a pointer to the new session.
Alternative path	If the service on the other side rejects the creation of this session, the function will return an error stating this.
Postconditions	If the service accepted the creation of the new session, it is now ready to be used.
Notes	–

Use case 10 – Create service

Name	Create service
Summary	The application creates a new service that will listen for new session establishment requests.
Preconditions	An endpoint exists, which the local computer has the ownership of.
Triggers	The application calls the function for service creation, specifying the desired service name.
Main path	The service is created.
Alternative path	If a service with the name specified already exists, no service will be created. Instead, the service creation function will return with an error message stating that the service already exists.
Postconditions	A service with the specified name now exists and listens for new session establishment requests.
Notes	–

Use case 11 – Accept incoming session establishment request

Name	Accept incoming session establishment request
Summary	The application accepts an incoming session establishment request. The new session is established.
Preconditions	There is an open service on an endpoint which the current local computer owns.
Triggers	The application calls the function which accepts incoming session establishment requests. An incoming session establishment request is accepted by the local session layer.
Main path	1. The characteristics of the new session is negotiated. 2. The session is created. 3. A pointer to the new session is returned to the calling application.
Alternative path	–
Postconditions	A new session now exists and is ready for use.
Notes	–

Use case 12 – Close session

Name	Close session
Summary	The application closes a session.
Preconditions	A session exists which the application has ownership of.
Triggers	The application calls the function to close a session, specifying the session's name.
Main path	The session is closed and deleted.
Alternative path	–
Postconditions	The session that was closed does not exist anymore.
Notes	–

3.3 Naming

In this section we will describe the design of the naming system that comes with our session layer. We begin by explaining the problems with naming that need to be addressed, followed by basic important concepts, which are needed to understand the naming system.

3.3.1 Naming problems

The biggest problem concerning naming in mobility-enabled networks is that the conventional naming system for the Internet, IP addresses, is both used to identify devices and to describe how to forward packets to them. Therefore there is a need to separate the information about how to reach devices (and other mobile objects) from their names. A name of an object should always be constant, as it is always the same object we are referring to, but the information about how to reach it will obviously change often when moving objects.

Not only IP addresses may change when moving objects. Also the port numbers associated with services attached to an object may change. Suppose we want to move a web service currently listening on TCP port 80 to a different computer. But the computer we are moving to already has a web service listening on port 80. The port of the moved service must now be changed, and peers connecting to the service must be made aware of this port change.

3.3.2 Introduction to the naming system

We have developed a naming system to resolve the problems stated in the previous section. This naming system makes it possible to identify endpoints independently of which host they are currently residing at. Additionally, it enables users to identify services independently of which port they are listening to at the moment, or which protocol they are currently using.

For endpoints, we have decided to use the same naming convention as the one used by SIP. The names for endpoints are of the format "object@domain". An example of such a name would be "web@example.com", identifying the "web" endpoint of the domain "example.com". The domain name itself, "example.com" is a standard domain name which can be found in a conventional DNS server.

When it comes to services, we have extended the SIP naming convention a little. A service name is written before the endpoint name, with a dot separating the service name from the endpoint name. So a valid identifier for a service we want to reach could be "www.web@example.com", identifying the service called "www" at the endpoint "web" which belongs to the domain "example.com".

In the traditional TCP/IP paradigm, applications connect from a port, to another port at the other peer. Now we connect to a service (which is mapped to a port), but where do we connect *from*? The answer is that we connect from an endpoint, e.g. an endpoint that is used for all outgoing session connections. We use a randomized TCP port, as in a conventional TCP connection.

A problem may arise here. Assume that we move our default endpoint to another computer and are forced to change the port from which our connection was initiated, and the other computer looses its network connection during our movement. When we have moved, we will try to resume the session with the other host, but it can not be found since it is disconnected from the network. We will give up and wait for the other host to resume the session when it has a network connection.

A few moments later, the other host gets network reachability again, and it tries to resume. The potential problem now is that we have changed our port, and the other host does not know about it. We have solved problems of this nature by having every host always listening for session control channel messages on a specific port which is always the same. The resuming host will send a RESUME message to the session control channel (described in section 3.7) specifying its IP address, port number and protocol. When we reply to this resume message with a RESUME_OK message, we specify our own current IP address, port number and protocol. This way both hosts will always now what specific network settings that should be used.

3.3.3 Session Name Server

As the syntax described in section 3.3.2 is not compatible with the original syntax of the DNS, there is a need for a new naming service. We will refer to it as the *Session Name Server* (SNS). This server keeps track of what IP address, port and protocol, that all services and endpoints associated with its domain, are currently using. When the application wants to start listening on a new service, it will make a request to create such a service at an endpoint which it has previously created. The request will be forwarded to the local SNS, and the current attributes (e.g. port, protocol) of the service will be recorded in the SNS. These attributes will of course be updated whenever they change.

There needs to be a way for the SNS to check if a request to create or delete an endpoint or a service is legal. Only certain users should be able to create new endpoints within a domain, and only the creator of an endpoint should be allowed to attach services to it.

3.3.4 Name lookup

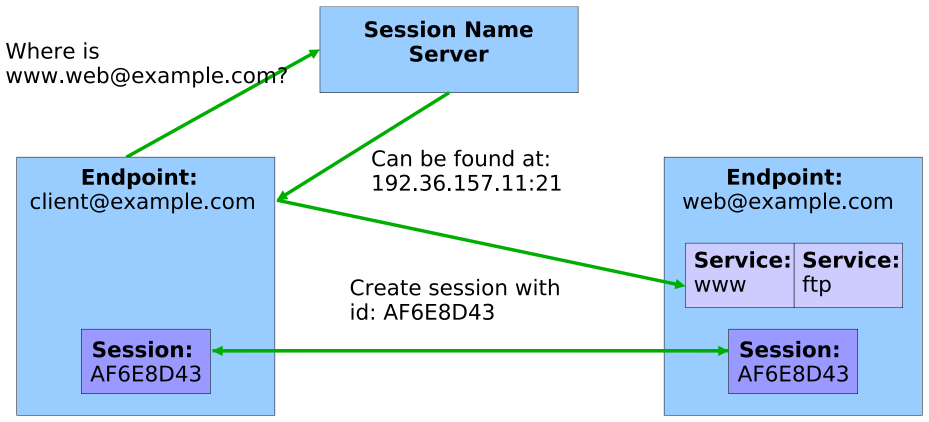

Session Name Server
Where is www.web@example.com?
Can be found at: 192.36.157.11:21
Endpoint: client@example.com
Endpoint: web@example.com
Service: www
Service: ftp
Create session with id: AF6E8D43
Session: AF6E8D43
Session: AF6E8D43

Figure 7: When creating a session there is a need too lookup the name for both the endpoint and the service.

The name lookup is a two step process. We will explain it by providing an example. Assume that we want to establish a session with “www.web@example.com”. We will first ask our DNS about the IP address for “example.com”. The SNS of example.com must always be present at the IP address which is stored in the DNS.

Now we will contact example.com and ask for information about how to contact the service “www.web” at this domain. The SNS at “example.com” will provide IP address, port and protocol information, and now we may connect and establish a session with the service. See figure 7.

There are some problems associated with this design we will discuss them in the next section. The reason why we have not addressed those problems is because we want to use the same syntax as SIP.

3.3.5 Shortcomings of the naming system

The naming system used have one major flaw in its basic design. It is actually not possible to distinguish it from an ordinary URL. Imagine that you want to use “scp” to copy a file to another host, you will then use a user@host URL to make sure that “scp” uses the right user to login to the host. If you want to use “scp” to a host named with the proposed naming system your identifier will be object@domain. Those two URLs can not be distinguished from each other.

3.3.6 Session Name Server Protocol

In this section we will describe the protocol used when communicating with the Session Name Server (SNS). This is a very simple protocol that we have designed and implemented with the only intention to use it in our prototype. No security mechanisms or extensive error handling is used. However, this protocol has served us well during our development of the prototypes.

There are eight different message types for requests to the SNS. Half of them concern endpoints, and the other half concerns services. The message types and the functionality are described below. See figure 8 for detailed information about the fields in messages of different types. The only two message types that are answered are GET_EP and GET_SERVICE. If the endpoint or service is found in the SNS, the corresponding IP address or port will be returned. If the endpoint or service is not found, 0 will be returned.

NEW_EP
This message type is used when we want to create a new endpoint. The payload of this message is the endpoint name followed by the 32 bit IP address that the name should refer to.

CHANGE_EP
When we want to change the IP address that an endpoint refers to we send a message of this type. The payload consists of the endpoint name followed by the 32 bit IP address.

DEL_EP
We send this message when we want to delete an endpoint. In this case that means that the SNS should no longer be aware of the endpoint, and it should also delete all services attached to this endpoint. The payload consists of the endpoint name.

GET_EP
This message type is used when we want to lookup the IP address of an endpoint. The payload consists of the endpoint name. The SNS will reply with the 32 bit IP address, or zero if the endpoint was not found.

NEW_SERVICE
When we want to register a new service at the SNS we use this message type. The payload starts with the service name followed by the endpoint name, with a dot in between, e.g. "servicename.endpoint@example.com". After this we supply a 16 bit port number that the service currently listens for new connections at.

CHANGE_SERVICE
This is the service equivalent to CHANGE_EP. The payload starts with the service name followed by the endpoint name, with a dot in between. After that comes the 16 bit port number that we have changed to.

DEL_SERVICE
This message type is used for deleting a service. The payload consists of the service name followed by the endpoint name, with a dot in between.

GET_SERVICE
When we want to know at which port a service is listening we use this message type. The payload is the service name followed by the endpoint name, with a dot in between.

3.4 Data integrity

An important role for the session layer is to provide means of restarting rebound connections. A rebound connection may or may not be in a state that the transport layer protocol can handle. Therefore the session layer needs to be able to retransmit any packets that were lost due to the rebinding, and synchronize the data stream appropriately so that the user of the session layer does not receive any corrupted data. This can be done in several ways. We have developed and tested two designs.

The first design is based on the concept of data segments that are acknowledged continuously, see Appendix A. After we had developed and implemented the first design, we were made aware of some drawbacks when testing it. We then created and implemented a second design, keeping the drawbacks of the first design in mind. We also tried to make the second design more in line with the semantics of the OSI definition of the session layer [3].

3.4.1 Checkpoint oriented integrity

This is a description of the design of our second data integrity system. The idea with our checkpoint oriented data integrity is that we continuously establish new checkpoints in the data stream. All bytes in this stream must be correctly transfered in order. When we have established a *checkpoint*, both peers have agreed that all data has been transferred correctly up to this point. With this knowledge we may drop all data before the checkpoint, as we know it has been safely transferred to the other side. If the connection is broken in any way we will resume from the last established checkpoint.

A checkpoint consists of two values, one for the send and one for the receive stream. See figure 9. For any given checkpoint identity number these two values must be identical on both hosts. To assure this we have developed a simple protocol that will continuously negotiate new checkpoints during data transfer. This protocol uses the same message headers as described in section 3.7.1. In the rest of this section we will describe the ideas behind our checkpointing protocol in detail.

How to reach an agreement We will now try to describe the procedure that is followed when two hosts want to agree upon a checkpoint. Figure 10 illustrates the procedure. A checkpoint is always negotiated between two hosts, we will here call them host A and B. The first thing to do when host A creates a new checkpoint is to send a checkpoint message. This message contains the relative position in the send and receive stream. These positions will be used

Header type	Field 1 size (bytes)	Field 1 content	Field 2 size (bytes)	Field 2 content
NEW_EP	64	Endpoint name	32	IP address
CHANGE_EP	64	Endpoint name	32	IP address
DEL_EP	64	Endpoint name	N/A	N/A
GET_EP	64	Endpoint name	N/A	N/A
NEW_SERVICE	64	Service name	16	Port number
CHANGE_SERVICE	64	Service name	16	Port number
DEL_SERVICE	64	Service name	N/A	N/A
GET_SERVICE	64	Service name	N/A	N/A

Figure 8: The SNS protocol message types and payloads.

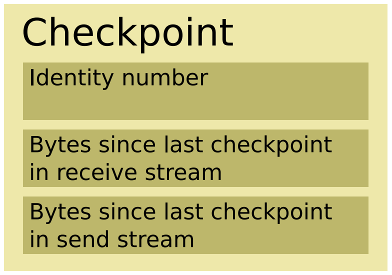

Checkpoint
Identity number
Bytes since last checkpoint in receive stream
Bytes since last checkpoint in send stream

Figure 9: The structure of a checkpoint.

by host B to understand at which byte this checkpoint should be created. The checkpoint that A just sent will be called the *pending* checkpoint. It is now important that host A does not send any further checkpoint messages until it has received an answer from host B. This is to assure that host A will always have a checkpoint in common with host B, by avoiding to have several of A's own checkpoint messages on the network at the same time.

Host B will receive this checkpoint message. The byte positions that the message contains can now be compared with the byte positions host B has. It is quite obvious that the amount of bytes that B has received should be equal to the amount of bytes that A has sent, but the amount of bytes that B has sent may be greater than the amount of bytes that host A has received. This is due to the full duplex nature of the stream that the checkpoint protocol is working on. Therefore host B now needs to decide which value to use, its own or host A's. The right thing to do is for B to choose its own value, the ordered nature of the stream will assure that if the checkpoint message arrives, all bytes that have been sent before it also arrived. Another way to look on this is to always use the highest of the two positions, this will give the same functionality as described above and also solve another problem which will be discussed later.

It is now time for host B to create a new pending checkpoint. This pending checkpoint will use the method described above the get its positions in the send and receive streams. The last thing to do is to send a checkpoint message back to host A. This message will contain the same positions as B's pending checkpoint.

When host A receives the checkpoint message, that was sent by host B as an answer to its request, it will update its pending checkpoint in the same manner as described above. That is, it will always choose the bigger of two values. After this it is safe for host A to send a new checkpoint message.

We previously talked about a problem that was solved by always using the bigger of two positions. This problem occurs when both hosts want to create a new checkpoint at the same time. The meaning of “at the same time” here is that the they we will not receive the other hosts message before they sent their own. When they receive the other hosts checkpoint message they will treat it as an acknowledgment of their own message and treat it as described above. As all values can be different there is a need for a common consensus (remember

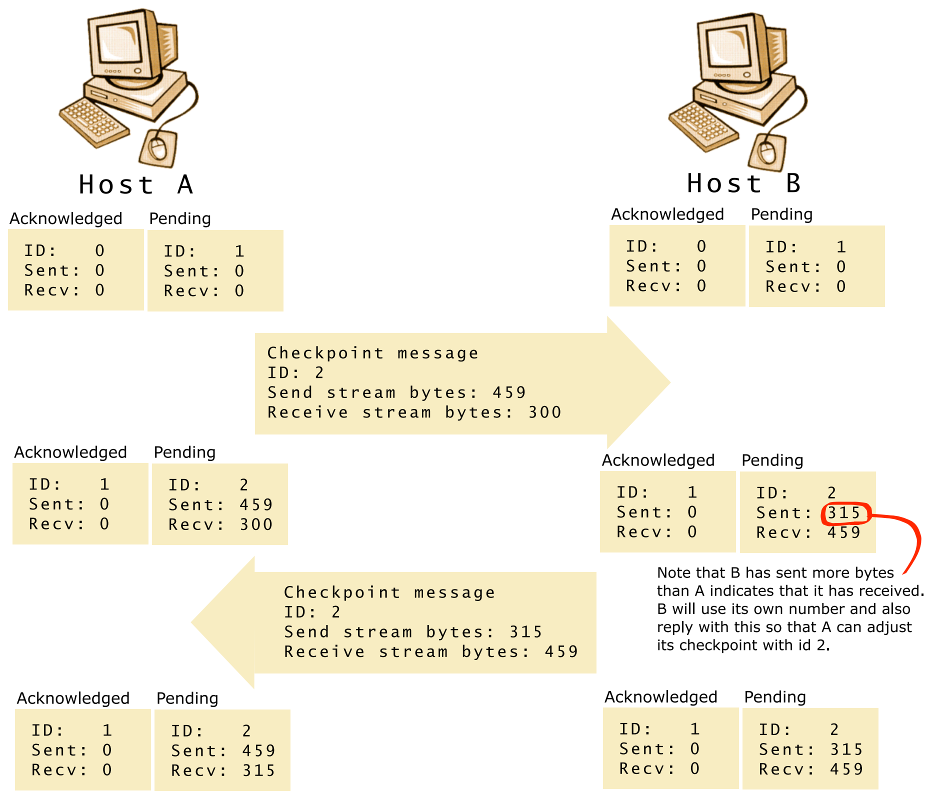


Figure 10: The prcedure followed when establishing a new checkpoint.

that the sole purpose of a checkpoint is to point out the same byte position on both hosts) and this consensus is reached when both hosts use the same method to set their positions. This method is as we said before, to always choose the biggest position.

Another problem is that there is a need to identify different checkpoints in the stream. Therefore we have developed a protocol in which there are three valid checkpoint identifiers – 0, 1 and 2. When creating new checkpoints we will give them one of these three identifiers. When asking to resume from the checkpoint with a certain identifier, like 2, it is very important that both sides refer to the same checkpoint. How this is ensured and why we need three identifiers is explained below.

We have talked about *pending* checkpoints but each host actually always keeps track of two checkpoints. They are called the *acknowledged* and the *pending* checkpoint. At the beginning of a session we assign the identifier 0 to the acknowledged checkpoint, and identifier 1 to the pending checkpoint. They are both placed at position 0 in the send and receive streams of a host, so in

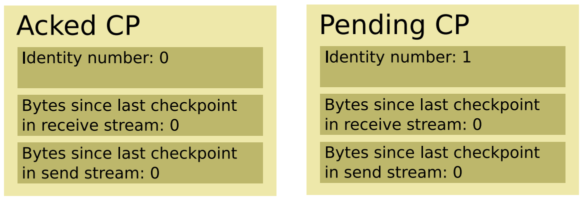

Acked CP
Identity number: 0
Bytes since last checkpoint in receive stream: 0
Bytes since last checkpoint in send stream: 0
Pending CP
Identity number: 1
Bytes since last checkpoint in receive stream: 0
Bytes since last checkpoint in send stream: 0

Figure 11: The local checkpoints when a session has just been initiated.

practice they refer to the same checkpoint at this stage. See figure 11.

When we later establish a new checkpoint, we want to add this as a new pending checkpoint. The pending checkpoint is the checkpoint that we currently are negotiating. When we add a new pending checkpoint, we must do something with the old pending checkpoint. As we can not send a checkpoint before we got an answer to the last one, we know that the old pending checkpoint, actually is an acknowledged checkpoint. There is no need for two acknowledged checkpoints so therefore we will drop all data before the old acknowledged checkpoint and make an acknowledged checkpoint of the pending one. This algorithm will now be described in detail.

1. Initially the host will have the acknowledged and pending at the same stream positions, as shown in figure 11.

2. Send and receive data.

3. Now two things may happen which will trigger new checkpoints, depending on which we will act differently.

    - We receive a checkpoint message from the other host.
        - If this checkpoint message has the same identifier as our pending checkpoint, then we should compare values between the checkpoints and take the maximum value of them.
        - If this checkpoint message has the identifier that comes after the one that the pending checkpoint has, e.g. pending is 2 and the new is 0, then the other side clearly wants to create a new checkpoint. We will start by dropping our acknowledged checkpoint and free its identifier. Then we will make our current pending checkpoint the new acknowledged checkpoint. We can do this because we are getting a request to make a new checkpoint, and that means both sides have agreed on the previous one, since we never send new requests of making checkpoints if we have any old requests pending. We will set our pending checkpoint to have the identifier and stream positions that the received CHECKPOINT message states.

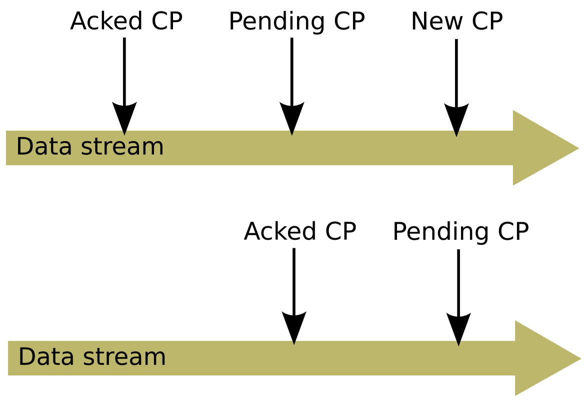


Figure 12: Creation of a new checkpoint.

We should also send a checkpoint message back to the other host, to inform it that we have received and acted on this CHECKPOINT message.

- If this checkpoint message has the same identifier as our acknowledged checkpoint, then this is a protocol violation and we should terminate this session.

- The send buffer at this host has less free space than a given amount, e.g. 50%, left. This buffer will soon be full so we need to establish a new checkpoint in good time to be able to make room for new data in the buffer.
  - If our sent_checkpoint variable is false, then we should send a checkpoint with the current values from our send and receive buffers. This checkpoint should have the identifier following our pending checkpoint. This checkpoint will also be set to pending and the pending checkpoint will be set to acknowledged.
  - If our sent_checkpoint variable is true, we need to await this checkpoint before we may create a new one. Otherwise we will not be sure that the other host got our last checkpoint message. We do not want to have two checkpoint requests on the network at the same time.

4. Optionally send and receive more data.

5. Repeat from step 2.

An example To explain our checkpointing system, we will now present an example which demonstrates how it works. In this example, a session is established between hosts A and B.

When host A has sent and received a number of bytes and decides to establish a checkpoint, it will send out a request to establish a checkpoint with the identifier 2. Host A will drop the current acknowledged checkpoint and make the pending checkpoint its new acknowledged checkpoint. It will then set its new pending checkpoint to the current position in the receive and send stream, and assign it the identifier 2 before sending the CHECKPOINT message to B. Figure 12 illustrates the process of introducing a new checkpoint. At the top, we see how the checkpoints position in the data stream when the new checkpoint is created. In the bottom, we have the same data stream, but here we have made the changes necessary for the new checkpoint to be introduced to the system.

When asking to establish new checkpoints, the identifier that is not currently used by the pending or acknowledged checkpoints is always chosen. This is why we need a third identifier, only two of them would cause misunderstandings about what checkpoint is really meant. Besides the identifier of the checkpoint it wants to create, host A supplies the byte position in its send and receive stream that the checkpoint should be placed at.

Host B will accept the request to create a checkpoint with the identifier 2, drop its acknowledged checkpoint, making the current pending checkpoint the new acknowledged checkpoint. Then it will assign identifier 2 to its pending checkpoint and set it to the specified position in its send and receive stream. In case host B does not agree with what A specifies as the number of bytes sent and received in the stream since last checkpoint, it will compare the "sent bytes" field from host A with its own "received bytes" field, and establish the checkpoint at the largest of these two numbers. Host A's "received bytes" field will be compared to host B's "sent bytes" field in the same way, and again it will pick the largest of these two numbers. This is done to assure that even if the checkpoints were sent simultaneously they will still refer to the same position in the stream.

To acknowledge the creation of the new checkpoint, host B will now send back a CHECKPOINT message to host A specifying at what values the checkpoint was created at host B. When receiving this message, A will adjust its own values if they differ from what host A originally sent out.

The next time one of the hosts wants to establish a checkpoint, it will have the identifier 0, since this is the only one of the three identifiers that is not occupied by the pending or acknowledged checkpoint at the moment. When checkpoint 0 is being established, the pending checkpoint will be set to this identifier. The checkpoint established before as pending with identifier 2 will now be the acknowledged checkpoint. Since we have started the establishment of a new checkpoint, we can be sure that the old pending checkpoint is set to the same values at both sides. This is because we never start the establishment of a new checkpoint if we have not received the acknowledgment of our latest checkpoint establishment request.

Rollback Now if we assume that the session is broken some time after checkpoint 2 was established. Host B is the first to try to resume the session. To be sure that data integrity is maintained, host B sends a RESUME message to A stating that it wants to begin from checkpoint 1, our acknowledged checkpoint.

A host will always try to resume from the acknowledged checkpoint.

Why not begin from the latest checkpoint? It could seem more reasonable to resume from the pending checkpoint, since that is the one most recently established. We do not do this because the pending checkpoint could be illegal due to previously lost messages. Suppose, for example, that host A tries to establish a checkpoint “0” with host B. Host A sets the pending checkpoint to 0 at its own side and then sends out the CHECKPOINT message to host B. The message is then lost and never reaches host B. Now the checkpoint that A refers to as 0 may not be the same checkpoint as B refers to as 0. So the pending checkpoint is always seen as a checkpoint undergoing establishment. We never resume from it.

Problems As Palmskog[7] discovered in his evaluation of this protocol it is possible to have data corruption under certain circumstances. For this to happen both host A and B need to send their checkpoints simultaneously. They will both treat the other hosts checkpoint as an acknowledge for their own. When A gets the checkpoint from B it will act as if this checkpoint was an answer to the checkpoint it sent. This means that it is now possible for A to send another checkpoint. This is the problem, there is no guarantee that B has received the previous checkpoint before A send this one. This means that it is possible for A to have two CHECKPOINT messages on the network at the same time. If both are lost there is no common checkpoint and therefore the session will be corrupted. Clearly this error will not happen very often, but a solution is needed and Palmskog is currently working on one.

3.5 Session Layer Daemon

The Session Layer Daemon is used to intercept what happens to the local network configuration and other mobile events such as attachment of mobile devices (such as USB mass storage devices). It should act as a data collector and an event generator. The basic responsibilities of the Session Layer Daemon are the following.

- Collect information about the current state of the network configuration.
- Collect information about the current state of mobile devices.
- Properly configure new network interfaces or old ones that get a carrier (e.g. someone plugs in a cable).
- Properly configure mobile devices and add any endpoints associated with them.
- Forward important changes to the session layer.

To achieve this the Session Layer Daemon will need means of monitoring and configuring network interfaces. Those means can often be provided by the operating system in some way. We think that each operating system should continue to use its own way of discovering and configuring new interfaces, but we should intercept those events so that we may forward them to the session layer. In case no such mechanisms exists, the Session Layer Daemon will need to contain all the functionality described above.

There is also a need to intercept events about mobile devices and configuring them. When a new device is discovered, it might contain endpoints that we need to insert in to the session layer. How this is done is also very operating system dependent.

3.6 Endpoint policies

Endpoint policies are used to decide if an endpoint should rebind to a new interface or stay with the old one, when an event is received. There needs to be different policies for different situations. To define policies is a hard task and is also very dependent on how the user wants his computer and applications to behave.

A policy could be a request to always try to keep the endpoint on an interface that has connection. It could also be an advanced policy that only allows the application to work during certain circumstances. That could be a scenario like an application may only use links that do not have any cost associated with the amount of data transferred. E.g. an FTP client that downloads a “not so important” file.

Policies are an important part of the session layer design, but it is beyond the scope of this paper. Therefore we have only implemented a simple policy that tries to maintain network connection. In other words our policies will only change interface when it is needed to maintain communication. We will not take important things like interface speed, latency and cost into account in our policy.

3.7 Session Management Protocol

There is a need for session layers on different hosts to communicate both data and control messages with each other. Therefore we have designed a protocol that has this functionality. This protocol is split in to two different parts. Control and data messages. Those messages are actually never sent on the same channel, they just have headers of the same structure.

When a session is established, a data channel is created using a protocol on the transport level, e.g. TCP. This channel is used for ordinary data transfer between hosts. When one host wants to suspend the connection or move to another address it will create a new channel to the other host to deliver a control message. We call this the *session control channel*.

Data messages Those messages are used to deliver data and to reach agreements about checkpoints. They are also used when we setup the connection. The main scenario for data transfer is to first send a CONNECT message, then await a CONNECT_OK. After this we are free to send DATA and CHECKPOINT messages. Finally we send a CLOSE message and await a CLOSE message in response.

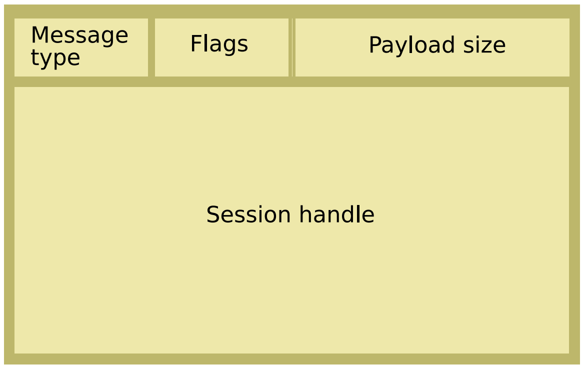


Figure 13: The session protocol header.

Control messages Those messages are used to deliver control events and are not sent over the data channel. Control events are mainly products of mobility in both time and space. e.g. a pause in the connection or a move in the network (or both). There are two types of messages, SUSPEND and RESUME. The SUSPEND message will suspend the other host, so that it will stop sending data and receive. The RESUME message will resume a previously suspended host (this does not mean that the host received a SUSPEND message, it may also have been disconnected by the network). This message may also contain information about any mobility in space, e.g. a change of network address. To assure that this resume is wanted and that the resume is *synchronous* this message will be answered with either a RESUME_OK or a RESUME_DENIED message. Those messages will also contain any information about mobility in space, e.g. a change of port number.

3.7.1 The session header

The main purpose for the session header is to be flexible so that it can be used on top of several different transport protocols. Therefore we have design it to be as simple and general as possible. See figure 13 which illustrates the different fields in the session header.

The session handle field This should be a big integer, preferably at least 128 bits[6], that is generated in a way so that it is unique to this session. It will then be used to identify this session when packets go through the network and is demultiplexed at hosts.

Combining the session handles (SH) from two hosts generates a session identifier which can uniquely identify a session, that is the communication channel between two hosts. The session handle could e.g. be a public key in an RSA

[6]This number needs to be unique in the network for session routing to be functional. Therefore it should be at least *statistically* unique

system, which could make it possible for hosts at both ends to securely identify each other. We will not discuss this in more detail as it is out of the scope of this paper.

The message type field This field is used to identify which type of message this is. We will discuss the different message types used later.

The flags field This field is for different flags that can be set in the session header. As of now only two bits are used by the session checkpoint numbers to distinguish between different checkpoints. Therefore six bits are left for extensions to the protocol.

The size field This 16 bits unsigned integer contains the size of the payload in this packet.

The payload The payload is different for different messages, the simplest is of course the data in a DATA message. But it can also be information about a change of address in a RESUME message.

3.7.2 Data channel initialization messages

Below is a description of all message types that will be transferred over the data channel. The data channel is the main connection between the two hosts, and every session has its own data channel.

CONNECT
: The session CONNECT message is used to initiate a session. It needs to contain both the endpoint name of the server and the endpoint name of the client. The name of the server will be used by any session gateway to route the message to the correct endpoint. The name of the client is used in the same way, but for sending back the CONNECT_OK or CONNECT_DENIED messages described below. The header will contain the SH of the sender instead of the receiver. This is to inform the other host of the SH that the local host has generated for this session.

CONNECT_OK
: This message is used to acknowledge a connection. It should contain the endpoint name of the client in its payload. The SH in the header should be the one for the host sending this packet, just as in the CONNECT message, to inform the other host about the responding host's local SH generated for this connection.

CONNECT_DENIED
: This message is sent if the host does not want to communicate with the host connecting. It should contain the endpoint name of the client in its payload. The SH in the header should be set to zero.

CLOSE
This message is used to finish a session. As sessions are persistent, this message is important.

3.7.3 Data channel messages

Data channel messages are supposed to be sent over an already active data channel. Therefore they do not need any kind of endpoint name in the message as the session is already established.

DATA
This message is used to deliver data from one application to another through the Session Layer. The payload should of course be the data to be delivered.

CHECKPOINT
This message requests initiation of a checkpoint. How this works is described in section 3.4.1.

3.7.4 Control channel messages

All messages sent over the control channel should have the endpoint name of the receiving session first in the payload. This to assure that the packet will always reach its destination.

SUSPEND
This message is sent when we want to inform the other host that we need to suspend this session. When this message is delivered no data messages should be sent from any of the hosts.

RESUME
This message is sent when we want to resume a suspended or broken session. The payload consists of the senders IP address and port number (this could be generalized to support other transport protocols). The host that receives this message will decide if it needs to rebind to this address.

RESUME_OK
This message is sent when a host has acted upon a RESUME message, e.g. rebound to a new address. This informs the other host that we are done with every action needed to resume, and that ordinary data communication can now be continued. As in the RESUME message the payload consists of the senders IP address and port number.

RESUME_DENIED
This message is sent to inform a host sending a RESUME message that this host is suspended by the user space application, and will not allow resuming the connection at this point. The other side should wait until the suspended side sends a RESUME message. Even here the payload consists of

the senders IP address and port number. This assures that the other host has updated information about our current network attachment point.

3.8 Session layer states

The inner workings of the Session Layer are built upon a state machine. In this section we will explain in detail how this state machine works. Figure 14 shows the state diagram. The transitions T1 to T18 are described and explained below.

3.8.1 ACTIVE (state)

This state is the normal state for the session to be in. This is the only state where any actual data is transferred. There are a few things that may happen in this state:

Transition 1:

Event	Our current network interface enters down state. There *are no* other interfaces in up state.
Action	Enter READY_RESUME state.
Description	As we lost our connection we want to resume as soon as possible. The READY_RESUME state will perform a resume as soon as there is a new working interface.

Transition 2:

Event	A user space application wants to suspend the session.
Action	Send a SUSPEND message to the other host and enter the SUSPENDED state.
Description	We will inform the other side that we are going to suspend the session by sending the SUSPEND message. Then we go to the SUSPENDED state where we will wait for a user space application to resume.

Transition 3:

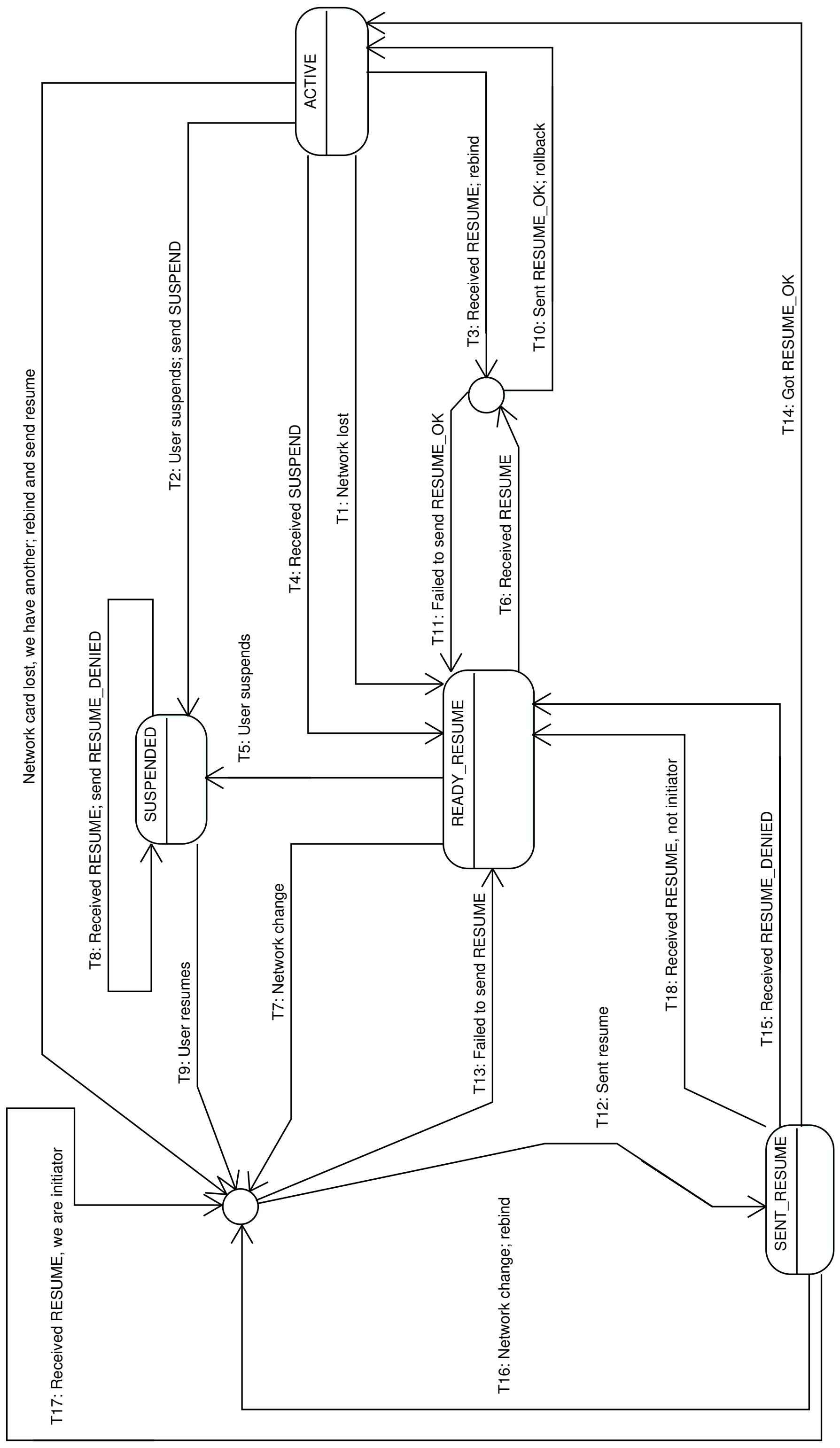

ACTIVE
SUSPENDED
READY_RESUME
SENT_RESUME
Network card lost, we have another; rebind and send resume
T1: Network lost
T2: User suspends; send SUSPEND
T3: Received RESUME; rebind
T4: Received SUSPEND
T5: User suspends
T6: Received RESUME
T7: Network change
T8: Received RESUME; send RESUME_DENIED
T9: User resumes
T10: Sent RESUME_OK; rollback
T11: Failed to send RESUME_OK
T12: Sent resume
T13: Failed to send RESUME
T14: Got RESUME_OK
T15: Received RESUME_DENIED
T16: Network change; rebind
T17: Received RESUME, we are initiator
T18: Received RESUME, not initiator

Figure 14: State diagram for the Session Layer

Event	We received a RESUME message.
Action	Rebind to the address and/or port specified in the RESUME message if they are different from before. Then go to the SEND_RESUME_OK junction.
Description	Here we will do a rebind only if the address and port specified in the RESUME message differs from the address and port we are currently using in the session's socket. Then we send a RESUME_OK message to the other host to acknowledge that we are finished with our rebind.

Transition 4:

Event	We received a SUSPEND message.
Action	Change to the READY_RESUME state.
Description	The other host tells us that it wants to suspend the session, so we enter the READY_RESUME state.

3.8.2 READY_RESUME (state)

The basic concept of this state is that we want to resume the session as soon as possible, but we have been put into this state because of network connection problems. Once we have entered this state, we will wait here until we get new information about the network interfaces that may enable us to resume communication. We also exit this state if the user wants to suspend the session, or if we get a RESUME message from the other host.

Transition 5:

Event	A user space application commands us to suspend the session.
Action	Change to the SUSPENDED state.
Description	We must change to the SUSPENDED state because we do not want to resume the session under any circumstances other than the user space application commanding us to resume it.

Transition 6:

Event	We received a RESUME message.
Action	Rebind the socket if needed, rollback to previous checkpoint, and enter the SEND_RESUME_OK junction.
Description	We did get a RESUME message, and since we are in the READY_RESUME state, we also want to resume as soon as possible. So we rebind our socket if the information in the RESUME message indicates that we have to. We will also perform a rollback to the previous checkpoint if we did a rebind. If we did not perform a rebind, there is no need to rollback, since the TCP connection was only paused and nothing critical has changed. When we are done with rebind and rollback we send a RESUME_OK message to inform the other host that we are done.

Transition 7:

Event	We received a network change event.
Action	Enter the **send_resume** junction.
Description	The network change event probably indicates that we have a new connection opportunity available, so we try to resume the session by sending a RESUME message.

3.8.3 SUSPENDED (state)

The concept of this state is that it will block the session from doing anything until the user space application commands the session to resume. We only enter this state if the user space application explicitly commands the session to suspend, and the only event that can make us exit this state is that the user space application commands the session to resume.

Transition 8:

Event	We received a RESUME message.
Action	Send a RESUME_DENIED message to the other host.
Description	Since we will not exit this state until an user space application wants to resume, we answer the resume request with a RESUME_DENIED message, to inform the other host that we are not ready to resume yet.

Transition 9:

Event	A user space application wants to resume.
Action	Change to the SEND_RESUME junction.
Description	Since the user space application wants the session to be resumed, we proceed by sending a RESUME message.

3.8.4 SEND_RESUME_OK (junction)

In this junction we attempt to send a RESUME_OK message, and react on the result of that attempt.

Transition 10:

Event	Successfully sent RESUME_OK.
Action	Perform a rollback to the most recent checkpoint. Then enter the ACTIVE state.
Description	We have now sent a RESUME_OK message, so we know that both sides has rebound successfully. To re-synchronize the connection, we perform a rollback. Then we will enter the active state to resume communication at the checkpoint we rolled back to.

Transition 11:

Event	Failed to send RESUME_OK.
Action	Enter the READY_RESUME state.
Description	Since we failed to send the RESUME_OK message, something must be wrong with the network connection. Therefore we enter the READY_RESUME state in which we will wait until we get updated network interface information.

3.8.5 SEND_RESUME (junction)

In this junction we send a RESUME message to the other host and take different paths depending on the result of this action. This is a junction more than a state, since we just perform an action and then proceed to a state, instead of waiting for some event to trigger the state change.

Transition 12:

Event	Successfully sent RESUME.
Action	Change state to SENT_RESUME.
Description	Since we sent the RESUME successfully, we can safely enter the SENT_RESUME state.

Transition 13:

Event	Failed to send *RESUME*.
Action	Enter the READY_RESUME state.
Description	Since we failed to send the READY_RESUME message something must be wrong with the network connection. Therefore we enter the READY_RESUME state in which we will wait until we get updated network interface information.

3.8.6 SENT_RESUME (state)

The concept of this state is that we have just sent a RESUME message, and now we are awaiting the answer in the form of a RESUME_OK message. We will also handle other events as described below.

Transition 14:

Event	We received a RESUME_OK message.
Action	Rollback to a previous checkpoint, and change to the ACTIVE state.
Description	The RESUME_OK message informs us that the other host has performed any required rebind and rollback, and is now entering the ACTIVE state. So we also perform a rollback to a previous checkpoint, and we then enter the ACTIVE state.

Transition 15:

Event	We received a RESUME_DENIED message.
Action	Enter the READY_RESUME state.
Description	The other host denied our resume request, which means that it is in the SUSPENDED state. We enter the READY_RESUME state, where we will wait for the other host to send us a RESUME message.

Transition 16:

Event	We received a network change event.
Action	Rebind the session socket if needed, then enter the SEND_RESUME junction.
Description	The network change event indicates that we must change the settings of our socket, and therefore we need to rebind to these new changes. When we have performed a rebind, the other host will have trouble finding us if we don't inform it about the changes. So we go to the SEND_RESUME junction to resume the session again.

Transition 17:

Event	We received a RESUME message and we are the initiator of the session.
Action	Change to the SEND_RESUME junction.
Description	If we get a RESUME message after we have just sent a RESUME message, this means that the two sides almost simultaneously have sent out these resume requests. In order to solve this potential conflict, we make the initiator of the session (the host that started the session) go to the SEND_RESUME junction, while the other host enters the READY_RESUME state to await the new RESUME message.

Transition 18:

Event	We received a RESUME message and we are *not* the initiator of the session.
Action	Change to the READY_RESUME state.
Description	We have the same conflict as described in the previous state transition, and since we are not the initiator of the session we enter the READY_RESUME state to await a new RESUME message.

3.9 Additional state machine properties

The state machine presented in section 3.8 is a good overview of how the Session Layer works, but it is not complete. To make the state machine easier to understand we have deliberately left some things out. Those things will be described and discussed here.

3.9.1 The CLOSED state

There is a state called CLOSED which is not shown in figure 14. The reason why we do not show this state in the figure is that it would make it harder to read and understand. So instead, we explain the CLOSED state in this section.

If we in any state receive a SESSION_CLOSE message, we will reply with a SESSION_CLOSE message. After that we will enter the CLOSED state.

We will also enter this state any time the local user space application decides it wants to close the session. In the later case, a SESSION_CLOSE message will be sent to the other host before we enter the CLOSED state.

When we are in the CLOSED state, the session has ended. No further communication will take place.

3.9.2 Handling of unanswered RESUME messages

We have found a valid scenario where the state machine will deadlock. This is indeed a big problem, but it also have a very simple solution. First we will describe the problem.

If host A successfully sends a RESUME message to host B, host A will be in the SENT_RESUME state. It will wait there until it either gets a RESUME_OK or a RESUME_DENIED message. If host B now tries to send a RESUME_OK or RESUME_DENIED message but fails, host A will still be waiting in the SENT_RESUME state. Host B on the other hand will notice that it failed to send the message and therefore not change the state. This is were the problem is, now it is posible for host B to send a RESUME message. This message, if received by A, would indicate a collision and therefore A would take appropriate actione depending on if it is the *initiator* or not. B on the other hand jsut sent a RESUME message and will not se this as a conflict. Therefore the states of host A and B will be out of sync. This could for example lead to a live lock where A and B will send RESUME messages back and forth between eachother. Luckily, there was an easy solution to this problem.

If B would have set an variable that indicates that it have received a REUME message, but failed to respond, it could send a RESUME_DENIED message, that would put A in the READY_RESUME state. After this B knows that A is not in SENT_RESUME anymore, and can therefore send a RESUME message.

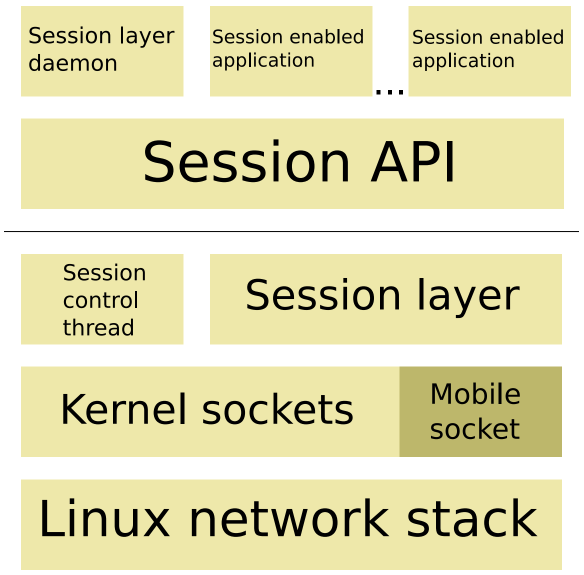


Figure 15: Implementation of the Session Layer design. The line marks the boundary between kernel and user space.

4 Implementation

In this chapter we will explain which parts of our design we have implemented, and *how* we have implemented these parts.

4.1 Basic implementation decisions

In this section we will outline how our implementation of the Session Layer was done, describing how we implemented the different parts that together build the Session Layer. The goal with our implementation is to demonstrate that the key features of the Session Layer can be implemented and that these features work as intended.

We carried out the implementation gradually during our design of the Session Layer. This was very helpful, since it made us aware of weaknesses in the design that we had to resolve.

While reading the following sections please look at figure 15 to understand the big picture. Our implementation is built around several important parts, which all more or less implement one of the conceptual parts in the design. We will start explaining the top of figure 15 and then slowly move downwards.

Session enabled applications An application that is linked to the session API is called a session enabled application. It can do anything that involves communication with other applications. We have developed a server and a client application to test the Session Layer with.

Session Layer Daemon The Session Layer Daemon is responsible for monitoring the network for changes. We have tried to save time by benefiting from an existing application that has capabilities similar to those our Session Layer Daemon should have. We have extended *netplugd*[7] which is very good at monitoring state changes in network interfaces. It will even handle Wireless LAN interfaces, so that when we get in and out of reach of a wireless network, netplugd will generate an event for us. Of course it also detects traditional Ethernet cable connections and disconnections.

We will extend netplugd so that it forwards any relevant information to the session layer. As you may notice this part has moved out from the Session Layer. This is due to the decision of putting the Session Layer in *kernel* space. As we want to use netplugd as our Session Layer Daemon we need to run it in *user* space.

Another responsibility of the Session Layer Daemon is to monitor events of mobile objects. In our implementation mobile objects are USB mass storage devices. To monitor events from those we have used udev[8]. Udev is used to invoke user space programs on kernel space events. Therefore we have built this system outside of Session Layer Daemon.

Session API This is a helper API that invokes the system calls provided by the Session Layer.

Session layer This is the central part of the design and thus also the central part of the implementation. This is where we have the data structures for keeping endpoints and sessions. It also provides the system calls used by the Session API. It will also take care of the buffers and functions that are needed to provide data integrity. This is done by checkpointing.

Session control thread This thread is responsible for communication between Session Layers. It is not in the design, but it is needed to listen for incoming connections from other Session Layers. It will act on the different control messages received to alter the state of the corresponding session.

Kernel sockets To be able to communicate with the world we need communication channels. These channels are provided by the kernel sockets. They work just like ordinary sockets, only that we keep pointers to them instead of file descriptors.

All these different parts work together to make this a valuable system. The central point is the Session Layer, to which all threads do changes and execute in. Threads may or may not continue into the *kernel sockets* and the *Linux*

[7]http://people.debian.org/ enrico/netplugd.html
[8]http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html

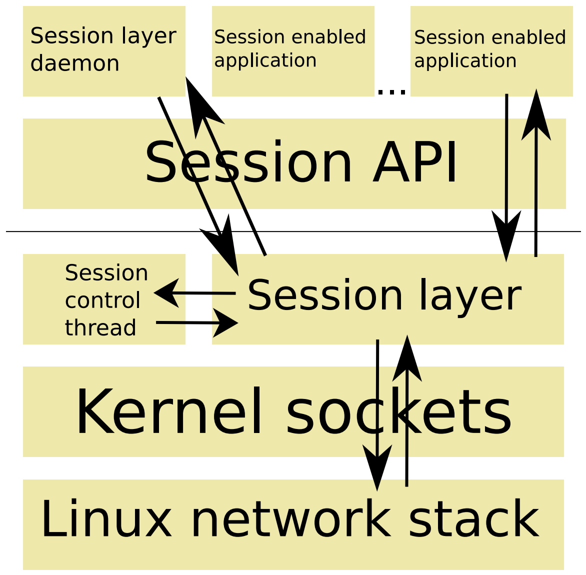


Figure 16: Different execution contexts in the Session Layer implementation. The arrows indicate which modules that communicate with each other.

network stack to get their work done. See figure 16 for a layout of the different execution contexts that execute the Session Layer code.

4.2 Basic concepts

In this section we will describe some basic concepts that are important to know about when reading about the implementation. Those concepts are used in many functions and we want to avoid explaining them every time they occur.

4.2.1 Locks and waitqueues

To protect different execution contexts from each other we need means of locking data structures. There is also a need for the session control thread to suspend execution of user threads. Locks are built around semaphores and we suspend execution using waitqueues that are native to the Linux kernel.

Communication locks At any given point in time there must not be more than one thread sending or receiving on a session. Therefore we have introduced a send and a receive lock. Those locks must be held when sending or receiving.

State lock This lock should be held when reading or altering the session state. This is to assure that transitions between states are *atomic*. This means that only one thread is granted to change the state of the session at any given time.

It is important that this lock is held during the execution of all code that is a part of the state change.

Interrupt lock When the Session Control Thread changes the state of a session it must not continue its work before other threads has acted upon this statechange. Therefore it must release the state semaphore. Before it can continue it must acquire the state semaphore again, but it can not be done before we know that other threads have acted upon the state. Therefore we will acquire the interrupt semaphore. This semaphore must be taken by a thread that wants to block on a call like send or recv. The Session Control Thread will interrupt any such function when it changes the state. The thread whom called the function will notice this, acquire the state semaphore, act upon the state and release the interrupt and state semaphores. This will allow the Session Control Thread to continue.

Session waitqueue This is a waitqueue for user threads executing in the Session Layer. Whenever an application calls send or receive on a session there is a chance that the call will block. The reason might be that there is currently no network that can deliver the message or that the application on the other side has suspended the session. To achieve this, functions like send and receive are built around loops that check the state of the session every iteration.

If the state changes from ACTIVE the user thread will put itself on the waitqueue. The user thread will not leave this queue until someone activates it. This way we can change the state from the session control thread and interrupt what the user thread is doing. When the user thread will check its state it will se that it is not active. When the session control thread notices that it is possible to activate the user thread again, it will do so by changing the state to ACTIVE and activating the waitqueue.

Hash table locks We decided to make the hash tables thread safe. Therefore all functions in the hash table API will lock the hash table before using it.

4.2.2 Receiving bytes

When we rollback to a checkpoint we might have given the user bytes that we will now receive again from the other host. Therefore we must keep track of how many bytes we have received since the last checkpoint. This will be done by increasing a variable every time we receive some data. This variable will be set to zero every time we perform a rollback. This way we know that as long as we have received less bytes since the last checkpoint than we have given to the user, e.g. is in the buffer, we must drop any incoming data. Doing this we can assure that no duplicate data goes through to the application.

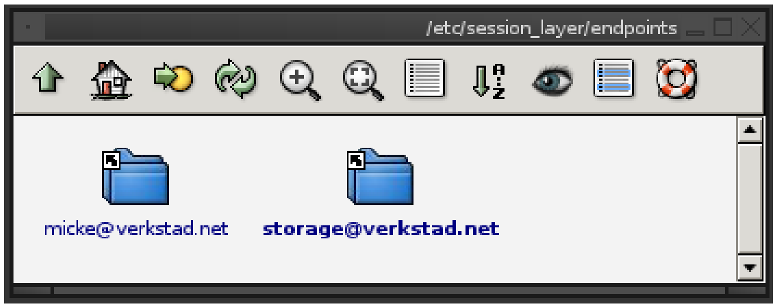


Figure 17: Endpoint directories

4.3 Disk based endpoints

4.3.1 Directory based endpoints

A directory on the local computer is a convenient way for the user to perceive the endpoint. For our demonstration, we have created a directory (“/etc/session_layer/endpoints/”) on every Session Layer enabled computer. This directory always contains subdirectories with names of the endpoints that currently reside at this computer. Figure 17 shows the endpoints directory of a computer.

Every time an endpoint is attached to the local computer, it will show up in the endpoints directory. For example, a user connects a USB memory stick to the computer. This memory stick contains a directory that can be identified as an endpoint. A link to the endpoint is created in the computer’s endpoints directory. This is seen in figure 17, where the endpoint “storage@verkstad.net” resides on a USB memory currently connected to the computer.

How to identify an endpoint directory There are two things that identify a directory as an endpoint. The first thing is that the directory has a name that fits the definition of an endpoint name. That means it can have a string consisting of the characters a-z, ’-’ and ’.’ followed by an ’@’ character, which is followed by a valid domain name. The second requirement on an endpoint directory is that it contains a file with the name “.endpoint”.

4.3.2 Saving and loading endpoints on disk

Our implementation includes functionality that enables a user to save down an endpoint with all its sessions to disk. The state of every program that has an open session will also be saved. The reason why we save the state of programs is that it is meaningless to resume a session unless we can bring back the program to the state where we suspended the session. Imagine that we save a session used by a file transfer application. Then we turn off the computer. When we later load the session again and resume it, the session will know where to continue transferring data, but the application will not know unless it can load its previous state from a file.

We have implemented two system calls for saving and loading endpoints in the kernel. They are *endpoint_save_to_buffer* and *endpoint_load_from_buffer*. The first system call will take an endpoint name and a user space buffer. When calling this system call it will save all session structures associated with the endpoint to the user space buffer. In the same way, the second system call will load an endpoint from a user space buffer. An endpoint will be created, and all sessions in the user space buffer will be copied into structures and associated with the new endpoint.

These system calls will be called from user space programs that will make sure the endpoint is saved and loaded from the right filename. The ".endpoint" file in an endpoint directory is normally empty. If the endpoint is to be saved to disk for later loading, the states of all the sessions will be saved into the ".endpoint" file. If a new endpoint directory is inserted into the endpoints directory, a user space program will look for the ".endpoint" file. If it is not empty, the endpoint's sessions will be loaded from this file.

4.3.3 Saving application states to disk

So how and when do we save and load application states on disk? The *endpoint_save_to_buffer* system call will go through all sessions associated with the endpoint that are to be saved, and make sure any current and future calls that have to do with the sessions will return the *-EMOVED* error. This will inform the user space application that it must save its state to a file since the endpoint will move.

The user space application will now save a file with name of the form *.state.application_name.x* in the endpoint directory, where *application_name* is the name of the application and *x* is an integer. The application will try to save to *.state.application_name.0*, if that file already exists it will try *.state.application-_name.1* etc. until it finds a non-existent filename.

After the user space application has saved its own state, it will close the session and then exit. This close will inform the Session Layer that this application is ready with saving its state. At this stage we have the complete endpoint, with all its sessions, services and application states saved in the endpoint directory. We may now move that directory between computers in any way we want. We could for example e-mail it or move to a portable disk drive.

4.3.4 Loading application states from disk

When we want to load an endpoint from disk, we will have a user space program called *loadep* read the *.endpoints* file in the endpoint directory, and then call the *endpoint_load_from_buffer* system call. The endpoint and all its sessions will be loaded into the kernel's Session Layer. After that, the *loadep* application will go through the files in the endpoint directory which have names starting with *.state* and start the corresponding applications with the state file as parameter.

The started applications are then responsible for restoring their state from the state files.

4.4 Session Layer API

In this section we will describe the Session Layer API towards the application. This is a library of functions that will be called by the application developer. Many of these functions are just simple wrappers around the corresponding system calls. These functions just exist in case we would want to add some user space functionality to these operations later.

4.4.1 Endpoint related functions

endpoint_get(struct endpoint **ep)
This function will set the ep pointer to the default endpoint of the local computer. In our implementation this local endpoint name is extracted from the host name and loaded at boot time.

endpoint_load_from_file(char *dir, struct sockaddr *sa)
This function will load an endpoint from a file and change its entry in the SNS to the address specified. It will also load all sessions associated with this endpoint.

endpoint_save_to_file(struct endpoint *ep)
This function will save an endpoint to a file. It will also save all sessions associated with this endpoint.

4.4.2 Service related functions

service_bind(struct endpoint *ep, int port, struct service **s)
This function will open a new socket on a random port and make an entry in the SNS mapping the service name to this port. This service may later be used to accept incoming connections.

service_unbind(struct service *s)
This function will release the bound port and remove the service from the SNS.

service_accept(struct service *s, struct session **sn)
This function will block until a connection is done to the service specified. It will then create a new session and return it.

4.4.3 Session related functions

session_connect(struct endpoint *localEndpoint, char *remoteEndpoint, unsigned short int port, struct session **sn)
This function will try to connect to the service specified. If it succeeds a new session will be created. This session can later be used for communication.

session_close(struct session *sn)
: This will close a previously created socket.

session_send(char *buf,int length,struct session *sn)
: This function will send an amount of bytes to the other side of this session.

session_recv(char *buf,int max_length,struct session *sn)
: This function will receive at maximum "length" number of bytes from the other side of this session.

session_suspend(struct session *sn)
: This will suspend this session, stopping all communication.

session_resume(struct session *sn)
: This will resume a previously suspended connection.

All functions may return negative error values. Those should be checked as they are bound to happen when e.g. an endpoint is moved (error value -EMOVED).

4.5 Naming

4.5.1 The server

The naming system we have developed is very simple. It is a standalone single threaded server. Data is stored in two hash tables. One contains endpoints and one contains services. When a message arrives we check the type of the message, hash the name of the endpoint/service and set or get the address/port.

4.5.2 The client library

We have implemented a kernel space client library which connects to the SNS to get information about endpoints and services. The client library is very simple. It consists of eight functions:

session_sns_add_endpoint(char *ep_name, __u32 address)
: Adds an endpoint entry in the SNS. Address should be the current IP address of this endpoint.

session_sns_change_endpoint(char *ep_name, __u32 address)
: Changes an entry in the SNS. Address should be the current IP address of this endpoint.

int session_sns_delete_endpoint(char *ep_name)
: Deletes an entry in the SNS.

session_sns_get_endpoint(char *ep_name, __u32 *address)
: Fills address with address information about the specified endpoint.

session_sns_add_service(char *ep_name, __u16 port)
: Adds a service to an endpoint in the SNS. ep_name should be of the format service.object@doomain. port is the port number that the service is currently bound to.

session_sns_change_service(char *ep_name, __u16 port)
: Changes a service at an endpoint in the SNS. ep_name should be of the format service.object@doomain. port is the portnumber that the service is currently bound to.

session_sns_delete_service(char *ep_name)
: Deletes a service. ep_name should be of the format service.object@doomain.

session_sns_get_service(char *ep_name, __u16 *port)
: Get information about the portnumber of a service. ep_name should be of the format service.object@doomain. port will be filled with the information.

All these functions will return a negative error value if they fail.

4.6 Synchronization

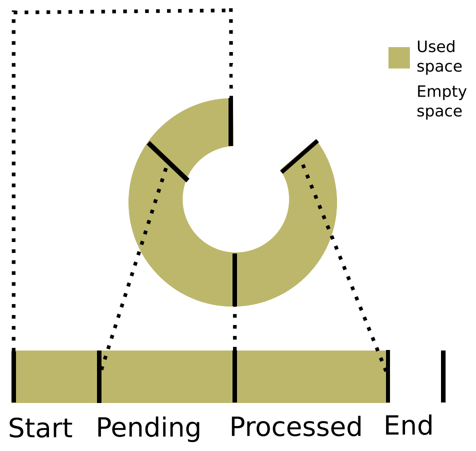
4.6.1 Buffer

To have a flexible buffer we decided to implement a circular buffer. For anyone not familiar with the concept of a circular buffer it is very simple. When data is added that goes beyond the end of the buffer it will be copied to the start of the buffer. The advantage with the circular buffer is that we can redefine the starting point of the buffer easily, when we reach the physical end of the buffer it will just wrap around to the physical start of the buffer. For our buffers we need several pointers, which are explained below.

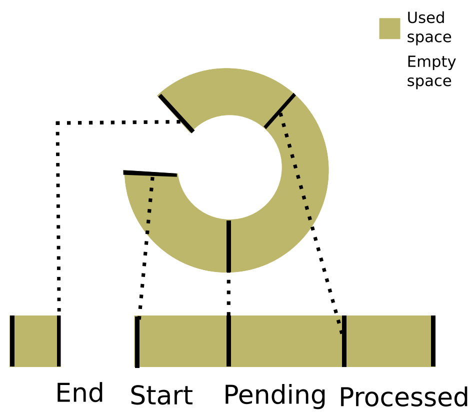
- start - start of the buffer and also the start of the last acknowledged checkpoint.
- pending - start of the pending checkpoint.
- processed - the point to which data has been processed. Processed data is either sent over the network or given to the user.
- end - marks the last byte in the buffer.

See figure 18(a) and figure 18(b) for a view of the normal and the wraparound case of the buffer.

Distance between different pointers In the buffer it is often interesting to calculate the distance between different pointers. This may sound like a trivial task, but as the buffer is circular, there is always a need to distinguish between two cases. For example, when calculating the distance between start and end, it should be calculated differently if end is greater or smaller than start. One could look upon this as a normal case and a wrap around case. The following functions for calculating distances is available in the session buffer.



(a) Circular buffer without wraparound.



(b) Circular buffer with wraparound.

Figure 18: Circular buffers in the checkpoint implementation.

session_buf_get_pending_end_size
: Returns the distance between pending and end, which is the number of bytes sent since the pending checkpoint.

session_buf_get_pending_processed_size
: Returns the distance between pending and processed, which is the number of bytes sent or forwarded to the user since the pending checkpoint.

session_buf_get_start_end_size
: Returns the distance between start and end, which is the number of bytes allocated in the buffer.

session_buf_get_start_pending_size
: Returns the distance between start and pending, which is the number of bytes sent or forwarded to the user since the last acknowledged checkpoint.

session_buf_free_space
: Returns the difference between the size of the buffer and the size of the data, e.g. the free space left in the buffer.

session_buf_capacity
: Returns the current capacity of the buffer, this is the maximum size of the data that can be stored in the buffer.

Buffers The Session Layer needs to buffer both received and sent data for every session. Therefore we have introduced a buffer abstraction that we call *buffers* which simply contains two buffers. We have defined a set of functions that works on two buffers instead of one. This is useful as all checkpoints are changed on both buffers simultaneously. When looking at the buffers it is important to know the difference between processed bytes for the send and the receive buffer. In the receive buffer, processed means that those bytes were given to the application. In the send buffer it means that those bytes were sent out on the network. This gives that the processed pointer for the receive buffer should never be reset as we do not want to give the application any duplicated data. On the other hand, the processed pointer of the send buffer has to be reset every time we want to restart from a checkpoint in order to resend all data associated with that checkpoint.

session_buffers_create
: Creates a new instance of the *buffers* data structure and initializes it.

session_buffers_add_send_data
: Adds a number of bytes to the send buffer. This will move the end pointer of the send buffer forward.

session_buffers_add_recv_data
: Adds a number of bytes to the receive buffer. This will move the end pointer of the receive buffer forward.

session_buffers_process_send_data
: Fetches a number of bytes from the send buffer. This will move the processed pointer in the send buffer forward so that we know that those bytes have been sent.

session_buffers_process_recv_data
: Fetches a number of bytes from the receive buffer. This will move the processed pointer in the receive buffer forward so that we know that those bytes have been given to the application.

session_buffers_new_cp
: This function will create a new pending checkpoint and make the pending checkpoint acknowledged. This means that we will move the start pointer to pending and the pending pointer to the value specified in the call to this function. This will be done for both send and receive buffers. Notice that by moving the start pointer forward we have freed all data belonging to the previous acknowledged checkpoint, thus making room for new data.

session_buffers_rollback
: This function will reset both buffers to the acknowledged checkpoint. This is done by setting pending to start for both buffers and setting processed to start for the send buffer.

4.6.2 Creating checkpoints

Sending checkpoint When the buffer is beginning to fill up, there is a need to create a checkpoint. In our implementation we will start to create checkpoints when 50% of the buffer contains data. This value is in no way based on any theory or experiment. It is just a value. When the buffer reaches this level, which is determined by calculating the distances between start and end, the send function will send a CHECKPOINT message. This checkpoint will have values derived from the buffers in the following fashion.

- Checkpoint number - This number is the identity number that is currently unused. E.g. if our acknowledged has 1 and pending 2 the new will have 0.
- Send position - This position is derived from the distance between the processed pointer and the pending checkpoint of the send buffer. This is the amount of data that we have sent since the pending checkpoint.
- Receive position - This position should equal the end pointer of the receive buffer, as this is the amount of bytes we received from the other host.

We must also set a variable containing the checkpoint number of the CHECKPOINT message just sent. It is important to know that we have sent a CHECKPOINT message when we do a rollback or receive a CHECKPOINT message.

Receiving checkpoint When a CHECKPOINT message is received we know that the acknowledged checkpoint can safely be dropped. Therefore we will put the new checkpoint received as our pending checkpoint, moving the pending checkpoint to acknowledged. If we did not send a checkpoint before we should send it now, following the same steps as if the buffer was nearing full. This checkpoint should have the same number as the received one. To only respond with a checkpoint if we did not send one prevents the two hosts from triggering each other every time they receive a checkpoint.

4.6.3 Resuming checkpoint

When we want to restart communication from a checkpoint, we should send a resume message to the other host. This resume message should contain the number of the *acknowledged* checkpoint. It the other host does not have this checkpoint it will send the number of our pending checkpoint in the RESUME_OK message. As the protocol assures that we always have a common checkpoint this will always work.

4.7 Session Layer Daemon

The Session Layer Daemon will be a quite big application that needs a lot of system dependent code. Therefore we have decided to extend an existing application to create a prototype SLD to be used for testing and proof of concept. We have extended netplugd[9] which is a simple application to get real time information about cables being plugged in or out of network interfaces, and Wireless LAN cards being within or out of range.

4.7.1 Netlink sockets

Netlink is a system in the Linux kernel that gives kernel space applications the ability to communicate with user space applications through BSD sockets. This is heavily used when forwarding networking and routing information from the kernel to user space and also within the kernel. A netlink socket is created in the same way as a normal BSD socket, but the address family AF_NETLINK must be specified. Netlink has a system of special messages and headers, which can be created and extracted using specific functions.

4.7.2 Our extensions

The original functionality of the netplugd application is that it uses netlink sockets to retrieve information about Ethernet cables being plugged in or out, and wireless interfaces becoming available and unavailable. Every kind of event will trigger a script which the user will specify.

[9] `http://people.debian.org/~enrico/netplugd.html`

We modified the netplugd application by removing the calling of these scripts. Instead, we do the following. When an interface goes down, we call endpoint_suspend() for all endpoints that reside on this device. This will suspend all endpoints that are attached to the interface which went down.

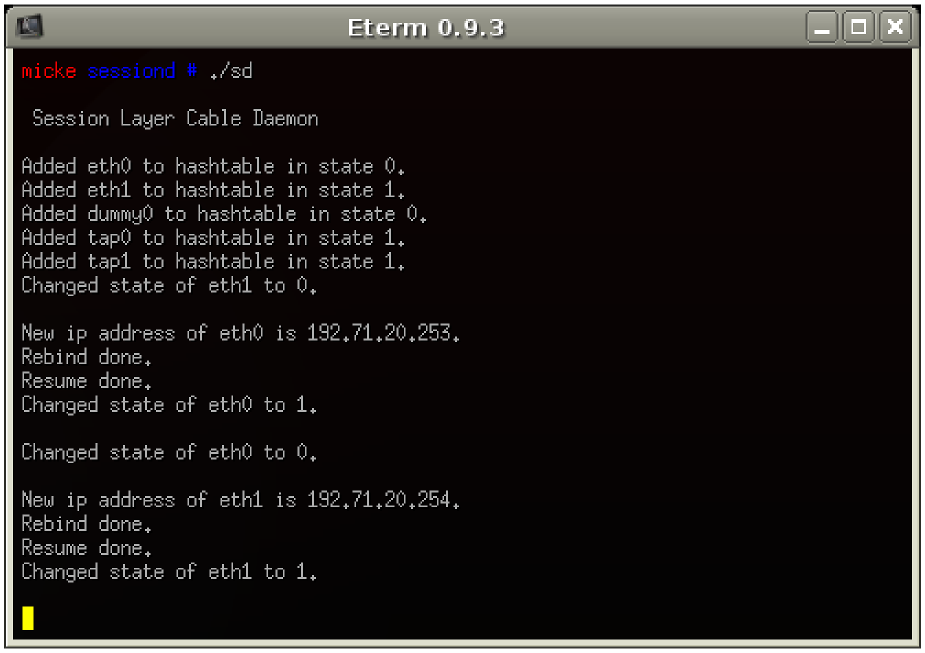
When an interface comes up, we execute the following.

1. Call *dhcpcd* from within netplugd to attain an IP address for the network interface.

2. Call *ifconfig* from within netplugd, parse out and store the IP address for the interface.

3. Rebind all endpoints to this new IP address.

4. Call endpoint_resume() for all endpoints.



```
micke sessiond # ./sd

 Session Layer Cable Daemon

Added eth0 to hashtable in state 0.
Added eth1 to hashtable in state 1.
Added dummy0 to hashtable in state 0.
Added tap0 to hashtable in state 1.
Added tap1 to hashtable in state 1.
Changed state of eth1 to 0.

New ip address of eth0 is 192.71.20.253.
Rebind done.
Resume done.
Changed state of eth0 to 1.

Changed state of eth0 to 0.

New ip address of eth1 is 192.71.20.254.
Rebind done.
Resume done.
Changed state of eth1 to 1.
```

Figure 19: Screen shot of the implemented Session Layer Daemon.

Figure 19 shows a sample run of the SLD. The steps executed in this test is to remove an Ethernet cable from the interface *eth1*, insert it into the other interface *eth0*, then pulling it out of eth0 and inserting it back into eth1. The SLD makes sure to suspend, rebind and resume the sessions appropriately.

The information about interfaces being added to a hash table in the beginning of the printout refers to a hash table within the SLD which holds all current network interface states. Whenever we get new state information we will update this hash table, in which the information is hashed on the name of the interface. We need this hash table to see what state we change *from* when a new event comes in on the netlink socket, in order to see the whole picture of what has really changed, and what is unchanged since before the event appeared.

4.7.3 Mobile objects

The Session Layer Daemon is supposed to handle events from mobile objects. As mentioned before we will use udev for this. Udev is very flexible and we have written some simple rules that will intercept events and start a script that will act upon these events.

Udev rules Udev rules are stored in "/etc/udev/rules.d/" it will parse all files in this directory that has the suffix ".rules". Files are parsed in the order they occur in the directory. Standard rules are named "50-udev.rules", we want to intercept events before those rules are activated so we named our file "10-usb.rules". This file contains the following rules.

```
KERNEL=="sd[b-z]", NAME="%k", SYMLINK+="usb%k", GROUP="users",\\
OPTIONS="last_rule"

ACTION=="add", KERNEL=="sd[b-z][0-9]", SYMLINK+="usb%k",\\
 GROUP="users", NAME="%k"

ACTION=="add", KERNEL=="sd[b-z][0-9]", RUN+="/bin/mkdir -p /mnt/usb%k"

ACTION=="add", KERNEL=="sd[b-z][0-9]", \\
RUN+="/etc/session_layer/session_script.sh /dev/%k /mnt/usb%k",\\
OPTIONS="last_rule"

ACTION=="remove", KERNEL=="sd[b-z][0-9]", RUN+="/bin/rmdir /mnt/usb%k",\\
OPTIONS="last_rule"
```

A brief description of those rules follow:

1. This rule will intercept the event that a new disk device is attached. It will create a symbolic link to this device that is named usb concatenated with the device name of this device, e.g. "usbsda".
2. This rule will intercept events about different partitions on the attached usb device. It will create a symbolic link to this device that is named in the same manner as above, adding only the device number, e.g. "usbsda1".
3. This rule will create a directory in "/mnt/" for each partition on this device. It will share the name with the partition, e.g. "/mnt/usbsda1".
4. This rule will run a script that will search this mobile device for endpoints. It will give it the name of the device and the directory it created in "/mnt/" as arguments to the script.

5. This rule will remove the directory created in "/mnt" when the device is removed.

Searching for endpoints We have written a simple BASH[10] script that will mount mobile devices and search them for endpoints.

```
#!/bin/bash
logger -t "Session Layer" "Found usb device! $1 $2"
logger -t "Session Layer" "Mounting..."
/bin/mount -t auto -o rw $1 $2
logger -t "Session Layer" "done."
logger -t "Session Layer" "Searching $2 for endpoints and iterating over them:"
endpoints=‘ls $2 -1a | grep \@‘
for i in $endpoints;
  do
  /etc/session_layer/endpoint_create.sh $2/$i;
done
logger -t "Session Layer" "Done with this usb device."
```

First this script will mount the device. Then it will search the root of the filesystem at this device for endpoint directories. To find an endpoint directory we will search for directories containing "@". Whenever an endpoint directory is found it will call an endpoint creation script and pass the absolute path to the directory as an argument.

Endpoint creation When an endpoint directory was found by the previous script it will call an endpoint creation script. This script will load the endpoint into the kernel and start any suspended programs. This is done by iterating over all the state files that are stored within this endpoint directory.

```
 #!/bin/bash
logger -t "Session Layer" "Creating endpoint from mobile directory $1"

# Load endpoint from file into system
/usr/local/bin/loadep $1

# Start every application specifying its state file
# e.g. /app/path/appname --statefile /mnt/usb1/object@domain/.state.appname.0
cd $1
app_path=/usr/local/bin/
ep_path=‘echo $1 | sed s/\\\/$// | sed s/.*\\\///‘
export XAUTHORITY="/root/.Xauthority"
statefiles=‘ls -1a | grep ".state."‘
for f in ‘ls -1a | grep ".state."‘; do
    /usr/bin/aterm -display :0.0 -e ${app_path}‘echo $f \\
    | sed s/\.state\.// | \\
    sed s/.[0-9]//‘ --ep ${ep_path} --statefile $1/${f} &
done;
```

First of all this script will run the "loadep" program to load the endpoint with all sessions into the kernel. Then it will search the endpoint directory for state files and execute application associated with those files. This will restore all session in the state where they were saved.

[10]http://www.gnu.org/software/bash/

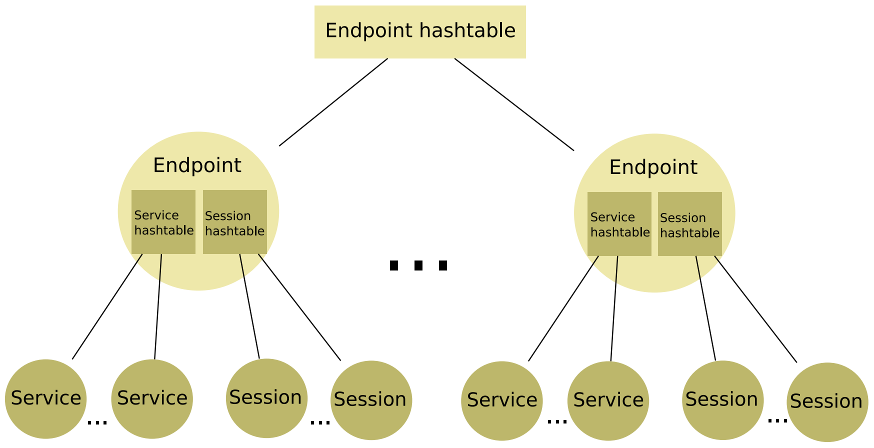


Figure 20: Session data structure.

4.7.4 Differences between design and implementation

After doing this, we have a very simplified version of the Session Layer Daemon. It will automatically change to new network interfaces when they are made available to us. It breaks our design principles somewhat, because it performs the suspend, resume and rebind commands itself. Our design states that the SLD should only *inform* the Session Layer about interface changes, but in our test implementation we carry out the rebinding of endpoints and sessions in the Session Layer Daemon.

For our testing purposes, we think this is satisfactory. We can see no major functional difference between executing the rebinding from the SLD or from the Session Layer itself. Since we have not implemented any sophisticated policy manager for the Session Layer, we might aswell perform these simple rebinding related tasks directly in the SLD.

4.8 Session layer states

In this section we will describe how the Session Layer state machine was implemented. The main controller of the state machine is a kernel thread which we call *Session Control Thread* (SCT). This thread is always active, and it is constantly listening for incoming events. The events are incoming messages on the session control channel (e.g. RESUME, SUSPEND, RESUME_OK, RESUME_DENIED).

4.8.1 Session data structures

The Session Layer contains a lot of data. Therefore it is important that the data is organized in a good way. The overall organization of the data is shown in figure 20.

Endpoint hash table This is a hash table which contains endpoints hashed on their network name, e.g. hd@verkstad.net. This makes it possible to have quick lookups of endpoints, by knowing only their name.

Endpoints Endpoints are named containers that hold the two functional entities that are needed for the Session Layer to work; services and sessions. This will associate them with the endpoint name that is used to identify their current network location. The endpoint data structure look like this:

- ep_name - This is the name of this endpoint, the same name as used for hashing.
- addr - This is the current address which is associated with this endpoint's name.
- sess_table - This is a hash table that holds sessions hashed on their session handle.
- serv_table - This is a hash table that holds services hashed on their service name, in this implementation this is the port number.

Service Services keep a listening socket open for incoming connections, they do not do much more. Therefore the data structure for a service is rather simple.

- name - The service's name, here a port number.
- socket - The listening socket that is used for accepting incoming connections.

Session The session data structure is used to hold all data that is important for the session to function properly. Why all those fields are needed will be motivated in more detail when we describe the implementation of the different functions that act upon the session. Below is a short description of all data fields in the session data structure.

- sid - This is the 128 bit long session handle, SH.
- peer_ep_name - This is the name of the endpoint at which the session we are connected to resides.
- local_ep_name - This is the name of the endpoint which our this session resides at.
- saddr - This is the IP address that our endpoint resides at.
- daddr - This is the IP address of the endpoint that the other end of this session resides at.
- state_sem - This is a semaphore associated with our state variable. This semaphore should be decreased when checking or changing the state of the session. This way we can assure that the state is always consistent.
- irq_sem - This semaphore should be held when we block on a recv call. This enables the control thread to notice this and interrupt us. It will then try to acquire this semaphore and therefore it will have to wait until we have acted upon the state change.

- socket - This is the mobile socket used for communication.
- buffer - This data structure contains a send and a receive buffer.
- sent_cp - This is the number of the checkpoint we have sent. It has to be -1 if we have not sent any CHECKPOINT message.
- acked_cp - This is the currently acknowledged checkpoint.
- pending_cp - This is the checkpoint that is in negotiation.
- recvd_bytes - This is the number of bytes received since last checkpoint. This value is compared with the amount of data given to the user since last checkpoint. Data will only be given to the user if this value is bigger or equal to the number given to user in the buffer. Otherwise we would duplicate data to the user.
- got_resume - If this is true it indicates that we got a RESUME message but we did not succeed to send an answer. This is an important knowledge about the state of the other host.
- initiator - This is set to true if this session was created from a connect call. Otherwise, if it was created from an accept call, this will be false. This is needed to distinguish the roles that hosts should have when conflicts occur.
- wq_head - This is the head of the wait queue associated with this session. When the session is in any other state then the ACTIVE state all ordinary calls, such as send or recv, will block on this queue. They will be activated by the session control thread when it is safe to proceed.

4.8.2 Session Control Thread

When we initialize the Session Layer, the kernel function kernel_thread() is called with a pointer to a function which we have defined. The contents of this function is what our Session Control Thread (SCT) will be executing. It is built around a while-loop which at the start of every iteration takes in a new message on the session control channel. When the message is received a new thread is spawned to handle this request. In this thread the endpoint which the message is destined to is looked up in the endpoint hash table. If no endpoint is found, the message is dropped.

When the endpoint which the message is destined to is found, the session which the message refers to is looked up in the endpoint's session hash table. We need the pointer to this session so that we can modify and read its state machine and other properties.

Next, we will switch on the message type, and for each message type we perform actions depending on which state the session's state machine is currently in. Below is a list of exactly what is done in each situation.

Incoming message of type RESUME: The resume message tells us that the other host wants to resume a session. The reasons for this could be many. For example that the other host previously lost network contact and now gained it back, or that the other host previously suspended the session and wants to resume it now. It could also be that the other host switched network interface and wants to update us with a new IP address.

ACTIVE
If we are in the ACTIVE state, we switch to READY_RESUME state, then we will interrupt any thread that is blocking on a recv or send call. We will then acquire the interrupt semaphore to assure that this thread has time to act upon this state change. After this we will drop through to the READY_RESUME case to do everything that is done there.

READY_RESUME
The first thing we need to do is to set the got_resume variable in the session to true. This should not be reset until we have successfully answered this message. Then we will extract the address the other host sent to us and change our destination address accordingly.

Then we will check the checkpoint that was specified. If it is our pending checkpoint we will move forward in the stream so that this will be our acknowledged. Now we will bind a socket to a port. This socket will alter be used to rebind this connection.

Now we will send a RESUME_OK message with our current address and the port number from the newly bound port. We will also specify our acknowledged checkpoint as the checkpoint to be used in this resume. If we fail to send this message we will go to the READY_RESUME state. If we succeed we will first reset the got_resume variable int the session. Then we will do an accept on the bound socket and wait two seconds for an incoming connection. If there is a connection we will close the session's old socket and give it this new one. Then we will activate any threads that were interrupted and move to the ACTIVE state. If the accept timed out we will go to READY_RESUME.

SUSPENDED
If we are in the suspended state, it means that we will not under any circumstances accept a resume request, since the local user of our session has not approved of it. We will send back a RESUME_DENIED message.

SENT_RESUME
Here we have a collision. Both sides want to resume at the same time. To avoid any deadlocks we have clear rules for what each host should do in this situation. If we are the initiator of this session, we will send a new RESUME message. If we are not the initiator, we will change state to READY_RESUME, so that we can take care of the incoming RESUME message that the other side will send.

Incoming message of type RESUME_OK: When we receive a RESUME_OK message we will act differently depending on the session's current state.

SENT_RESUME
This is the state we will most commonly be in when we receive a RESUME_OK message. We will first extract address information from the message and update our information, then we will check the checkpoint specified. If it is our pending checkpoint we will move forward in the streams so that it will be our acknowledged. This assures that both hosts are synchronized. If the we do not have the checkpoint specified we will terminate this session.

Now we will try to connect to the port specified in the message. If this succeeds we will change the state to ACTIVE, otherwise we will go to READY_RESUME.

All other states
Drop the message, since it does not make sense to receive a RESUME_OK message when we are not in the SENT_RESUME state.

Incoming message of type RESUME_DENIED: We will only handle the RESUME_DENIED message if we are in the SENT_RESUME state:

SENT_RESUME
Change state to READY_RESUME. Since we received a RESUME_DENIED the other side is in the SUSPENDED state, which means we will not resume the session until the other side wants it to be resumed.

All other states
Drop the message, since it does not make sense to receive a RESUME_DENIED if it is not in response to a resume request.

Incoming message of type SUSPEND: When we receive a RESUME_DENIED message we will act differently depending on the session's current state.

ACTIVE
Change state to READY_RESUME. The other side wants to suspend the session so we go to READY_RESUME.

All other states
If we are in any other state the message will be dropped.

4.8.3 Session system calls

To make it possible for the session API to communicate with the Session Layer we have defined a number of system calls. They are similar to the functions defined for the session API. We will therefore divide them into the same categories as the session API.

Endpoints

endpoint_create
This will create a new endpoint and add it to the endpoint hash table. This endpoint can later be used in subsequent calls to functions for sessions and services that we want to make use of this endpoint.

endpoint_close
This will close an endpoint and remove all sessions and services residing at this endpoint. This will of course kill any communication on those sessions.

endpoint_suspend
This will suspend all sessions on this endpoint. This is useful when we want to put all sessions into the READY_RESUME state when we e.g. lose our network connection. Note that this function will not put sessions associated with the endpoint in the SUSPEND state. That state can only be reached when the user *explicitly* suspends the session. Here sessions will be put into the READY_RESUME state.

endpoint_rebind
Rebinds an endpoint to a new interface by rebinding the source address of all session that reside at it. This is done by calling the session_rebind function for each session. This function is described in section 4.8.4. It will also update the SNS with new information about rebound endpoint.

endpoint_resume
When we e.g. have rebound to a new interface there is a need to resume all sessions associated with this endpoint. This will be done with a call to this function. It will send a RESUME message and change the state to SENT_RESUME for all sessions. The RESUME_OK message will then be handled by the SCT. If any of the sessions are not in the READY_RESUME then they will be ignored.

endpoint_save_to_buffer
Saves all session structures associated with the specified endpoint to the user space buffer provided by the caller. Will then remove the endpoint from the local computers Session Layer, and all sessions and services associated with it.

endpoint_load_from_buffer
Loads all session structures into the specified endpoint from the user space buffer provided by the caller. Will initiate and setup the endpoint and all sessions so that they are ready to be used.

Services

service_bind
This function will bind a service to a specific port number. This is done by creating a listening socket that is stored in the service data structure. The service name will also be associated with the port in the SNS.

service_accept
This function will use the previously bound socket in the service data structure to accept incoming connections. Those connections will be given to newly created sessions.

Sessions

session_connect
This function will connect to a service located at an endpoint. As the name resolution is done in the user space API this function actually gets an IP-address and a port. It will connect to this port, send a CONNECT message and wait for a CONNECT_OK. After this a session is created and given the socket that was used for the connection.

session_send
This function will send a number of bytes given to it to another application. This is a quite complex procedure as the Session Layer needs to take care of data integrity. The function is constructed around a main loop. This main loop may not be left before all the data from the user is in the buffer and sent over the network. The main loop will go through the following steps during every iteration.

- State: ACTIVE
  1. If there is room in the buffer add data from the user to the buffer.
  2. If we have sent a checkpoint and the buffer has room for more data, then we will make a nonblocking receive. This is to check if there is a checkpoint from the other host waiting to be received. If there is a checkpoint here we will process it. If the buffer did not have room for any more data we will make a blocking receive. This might receive checkpoints which can free room in the buffer.
  3. If we had not sent a checkpoint we will check if the buffer is more than 50% full. Then we will send a CHECKPOINT message.
  4. Now we will try to fetch data from the buffer. If we got it we will send it. Otherwise we will check if the data was put into the buffer. Then we will return to the user as we know that it was sent (because otherwise we would have got data from the buffer to send).
  5. Restart from step 1
- Any other state
  Block on wait queue of this session.

session_recv
This function receives at maximum a specified amount of bytes from another application. This function is a lot simpler than the send function but still it is not trivial. It is also built around a main loop. This main loop will return as soon as there is unprocessed data in the buffer. This data will be returned to the user. If the buffer have no unprocessed data this function will block on the recv_to_buffer call, see 4.8.4. Another possibility is that the state is not ACTIVE, then this function will block on the wait queue of this session.

session_suspend
This function will suspend the session by changing its state to SUSPENDED. There are two cases that must be taken care of. First the session might be in ACTIVE state. If that is the case we must send a SUSPEND message to the Session Layer at the other host. Second, if we are in READY_RESUME state we must not send the SUSPEND message. In both cases we should of course go to the SUSPENDED state.

session_resume
This function will resume the session if it has been suspended with the session_suspend function. Therefor it will first of all make sure that the state is SUSPENDED. Now it must check the got_resume flag. If it is set we have previously got a resume that we failed to answer. The correct thing to do in this situation is to first send a RESUME_DENIED message, to release the other host from the SENT_RESUME state. Then it will try to send a RESUME message. If this message is successful we will go to the SENT_RESUME state. If it fails we will go to the READY_RESUME state. This will force the session to retry to resume when network changes. After this we will return success to the user, if the resume was not sent, next call to recv or send could block.

session_close
This function will start by sending a CLOSE message. The close message means we will not send any more data to the other side, except for the case we need to perform a resume and we must rollback to an old checkpoint. After sending the CLOSE message, we will enter a receive loop where we take in packets and perform normal receive operations. We will exit the loop and return from the function once we get a CLOSE message back. If communication was somehow disrupted after we sent our first CLOSE message, and a resume is performed by the other host, we will issue a rollback and send out all data since last checkpoint followed by a new CLOSE message. This is done every time this happens.

4.8.4 Session auxiliary functions

This is a collection of functions used by the system calls when they need different tasks done. This can be things like sending data or suspending a session.

session_rollback

This function will rollback the session to the current checkpoint. This is done by setting the recvd_bytes variable to zero. There is also a need to reset the sent_cp variable as this information is not valid any more. After this we will call a buffer helper function that will reset the appropriate pointers in the buffers.

For the send buffer this means that the pending checkpoint is moved to the position of our acknowledged checkpoint, we will also move the processed pointer of this buffer to this position. This will assure that when we restart the session all data from the acknowledged checkpoint and forward will be resent.

For the receive buffer this is a little bit different. We will move the pending checkpoint to the position of the acknowledged, but we will not move the processed pointer. This pointer tells us how much we have copied to userspace, and as we do not want the user to get any duplicate data we do not want to reset this. Instead as we should reset the recvd_bytes variable, which was done earlier in this function.

session_rebind

This function will set our source or destination address according to the parameters given. It can also change the source or destination port.

send_session_ctrl_msg

This function will send a session control message over a TCP connection to another Session Layer. The message may vary between the different types this function can handle. This is the basic steps of the function.

- First we will create a session header and copy message type, flags and session handle to it.
- Now we will check if the message type is one of RESUME or RESUME_OK. If that is the case we will set the flags to contain our current checkpoint, that is the checkpoint we want to make a rollback to.
- After this we will try to create a TCP connection to the session control thread on the other host. If it fails we will return a negative error value to indicate that things have gone wrong.
- Now it is time to send the header and the bulk of this message. The bulk is the endpoint name of the receiver. This is always included so that any gateway in the network may do its job properly.
- Once again we check if the message is either RESUME or RESUME_OK. If that is the case we will now append the current address and port number of the socket used by the session for data communication. This is to inform the session control thread on the other host about eventual changes in our network connection.
- Now we will close the TCP connection.

send_session_data_ctrl_msg

This function will send a CONNECT, CONNECT_OK or CHECKPOINT message over the data channel. Step by step it works like this.

- First we will create a session header and copy message type, flags and session handle to it.
- If the message is of CONNECT or CONNECT_OK type we will append the name of the receiving endpoint and our own endpoint. This to assure that the message can pass through any gateways in the network.
- If the message is of the CHECKPOINT type we will set the header flags to contain the checkpoint we want to create and we will add a data structure with two integer values, one for the receive stream and one for the send steam.
- This function will return what send returns.

get_session_packet

This function makes it possible for the Session Layer to think that it always gets every packet as a complete packet. Not as a stream of bytes. This means that it will block until it has all data in a packet. The function also takes a filter of packet types so that we can assure that we only get packets that we want to have. This is the basic layout of this function.

- Check if the buffer is big enough to hold the header, otherwise return a negative error value.
- If the buffer was big enough, execute the following loop.
  1. Repeat receive calls until we have all of the header in the buffer.
  2. Now apply the filter on the header to see if it should be taken care of.
     - The packet is wanted.
       * Check the length of the payload and make sure it fits in the buffer.
       * If it does, repeat calls to receive until we have all of the payload in the buffer. Now return the packet. This is the normal exit of the loop.
       * If it does not, return a negative error value.
     - The packet is not wanted. Then we use the buffer as a intermediate storage and repeat receives until we have processed the entirely payload. Note that here the payload does not need to fit in the buffer. We overwrite the buffer every iteration, this is possible as we do not want the payload.
  3. Repeat from step 1.

create_session

This function creates a new instance of session data structure. To do this it will need an active socket, the endpoint name of this and the other host and a session handle. It will copy those parameters into the session data structure and it will create buffers and set different fields to their initial values.

recv_to_buffer
This function is called when we want to receive a packet on the data channel. To do this this function will start with a call to get_session_packet with a filter of DATA, CHECKPOINT and CLOSE. If this returns successfully we will switch on the message type.

- In the case of a DATA packet there are three cases.
  - We have received as many bytes since the last checkpoint as there are bytes in the buffer. This is the normal case when we just add the received data to the buffer. We must of course also increase the amount of data we have received since the last checkpoint.
  - We have received less bytes than there are in the buffer, but if we add the number of bytes received now we will overlap the buffer. In this case we have finished a resume so we need to copy the bytes in the packet that are after the end of the buffer to the buffer. We must also increase the count of bytes received with the same amount.
  - If none of the above statements were true we already have the data received in the buffer. Therefore it is safe to drop it, but we still need to increase the byte count.
- In the case of a CHECKPOINT the first thing we check is if we have an unanswered checkpoint request.
  - If that is the case we will: If the received checkpoint has the same identifier as our pending we will set unset the variable that indicates that we have an outstanding request. This since this message can be seen as an answer to the previously sent request. If the received checkpoint differs from our pending then this is a protocol violation and we will terminate the session.
  - If that is not the case this is a request for a new checkpoint. We must therefore establish this checkpoint as our pending checkpoint. After this we must send a CHECKPOINT message as an answer.
- In the case of a CLOSE packet we need to inform the user that the session was terminated by the other host. This is done by returning a negative error value. Moreover we need to respond with a CLOSE packet. The session will also be moved to the closed state.

4.9 Session Layer Demonstration

To show the Session Layer and all its components in action, we have created a demonstration setup. In this demonstration, we have implemented a simple file server and client. The server will transfer a file to the client. The server is bound to the endpoint “storage@verkstad.net” listening on the service “file.storage-@verkstad.net”. The client is bound to the endpoint “client@verkstad.net”.

Both the storage and client endpoints reside on their own USB memories. The directory for the storage endpoint contains the file to be transferred to the client. The file that the client receives will be stored in its endpoint directory. The demonstration setup shows how users can move the endpoint USB memories between different computers, and simultaneously changing which network interfaces and IP addresses they use, without breaking any sessions or services.

5 Analysis

5.1 Introduction

In this chapter we will explain how we have tested our work and evaluated the results. We will also discuss the results we have gotten from the testing.

5.2 Testing and evaluation

5.2.1 Method

What we want to evaluate The goal with this thesis project has been to design and implement the Session Layer. The main reason for doing this is to show that it is possible to realize the goals stated in section 1.1.3, and that it can be implemented as a reliable, easy to use software system. Our work can be divided into a few different categories which we want to evaluate.

Functionality testing
It is important that the functionality that is described in sections 1.1.3 and 3.2.2 is implemented correctly. We have tested this by running the use cases using our implementation, and then checking if we got the expected results.

Reliability testing
It would be quite easy to design and implement a software system that fulfills the requirements stated in our use cases, while being very unstable and unreliable. So we need to separately run stressful tests that will create situations that are unusual but possible. Additionally we want to run these tests for extended periods of time, to increase the probability that we have covered all situations that may arise when using the Session Layer.

Formal verification
Even if we run very comprehensive tests on our Session Layer implementation, we can never be totally sure that we have covered every situation that may arise. Therefore the verification of the correctness of our Session Layer design and implementation will continue with formal verification work carried out by Palmskog [7].

5.2.2 Hardware and software

In this section we will briefly describe what hardware and software we have used when we performed the tests.

Hardware In all our tests, we have used three computers. The first two computers were used as Session Layer client and server. They are equipped with 3 GHz Intel Pentium 4 CPUs, 1024 MB memory and 40 GB hard disk drives. The third computer was used as a Session Name Server. It is equipped with an 1 GHz Intel Pentium 3 CPU, 512 MB memory and a 40 GB hard disk drive. 100 MBit Ethernet network interface cards were used in all computers.

Software All test computers have a Gentoo Linux installation. We have implemented the Session Layer in the Linux kernel, version 2.6.15. As compiler we have used GCC version 3.3.6.

5.2.3 Functionality testing

To verify that we have implemented the required functionality in our Session Layer prototype, we have performed every use case described in section 3.2.2 to see if our prototype acts as expected. In this section we will go through a quite typical life cycle of an endpoint and some sessions that reside at it, and in the description of our test we will explain how all of our use cases are satisfied.

1. We have two hosts, A and B, which both have their own endpoints called a@verkstad.net and b@verkstad.net respectively. Those endpoints were created earlier using a system call, therefore **use case 7; create endpoint** is satisfied. We have another host, which we call C, that runs our Session Name Server.

2. Host A creates a service called *s*. We can see at the SNS output that an endpoint which can be reached through the name *s.a@verkstad.net* has been inserted into the SNS service table. The service has started listening on host A, and thus we have fulfilled the requirements of **use case 10; create service**.

3. Host B uses its client program to connect to the new service at host A. As we can see from the system log that the Session Layer writes to, the session ID is negotiated correctly between the two endpoints, and the session is now ready to be used for communication. Therefore **use case 9; create session**, and **use case 11; accept incoming session establishment request**, are now both shown to be working correctly.

4. In our simple test setup, the server currently running on host A will start sending a file to the client running on host B.

5. The application at host B will now suspend the previously created session. All communication stops, and both hosts now block on their send and receive calls. **Use case 5; application performs suspend**, has now been proven to do what it should.

6. The application at host B resumes the session after a while, and communication starts again. No data corruption is detected. **Use case 6; application performs resume**, is satisfied.

7. The user at host A now pulls out the host's Ethernet cable. This cable is its only connection with the network. Communication stops and we can

see that the send and receive calls that the server and client have made are now blocking. **Use case 1; unexpected disconnection**, is now satisfied.

8. Host A's user inserts the cable again, but into another interface at host A. After a resume is automatically carried out, communication is working again. No data corruption has occurred. **Use case 2; new network interface, no previous interface exists**, is satisfied.

9. The user at host B inserts a new Ethernet cable into a previously unused interface on the computer. Due to our simple endpoint policy that states that we should always change interface when a new one appears, that is exactly what will happen. Communication continues on the new interface after performing a resume. No data has been corrupted. **Use case 3; new network interface, previous interface exists**, is satisfied.

10. Now both users pull out their Ethernet cables and switch to another interface. When the cables are plugged back into new interfaces, neither host's information about the other endpoint's position is correct. They will ask the name server and perform a resume. Communication continues and no data has been corrupted. Therefore, **use case 4; both hosts simultaneously rebind**, is satisfied.

11. The file transfer has now completed. Both hosts close their sessions. Communication stops and the session is destroyed. **Use case 12; close session**, is satisfied.

12. The server at host A will close its service and endpoint. They are now both gone in SNS endpoint and service tables. **Use case 8; close endpoint**, works as intended.

5.2.4 Reliability testing

In this section we will describe the tests used to verify that the implementation can handle unlikely situations and conflicts. The tests that we describe here have been used through the whole implementation phase and they have helped us to understand how the session's state machine act when unlikely situations occur. Some errors found with those tests have forced us to change the design of the state machine. We will describe three tests and during the implementation we have used them together in different fashion. The final test was to combine them all.

Suspend and resume testing In this test both the server and the client calls the suspend and then the resume library function. Those functions are called with a probability of $\frac{1}{128}$ after each send and receive call. This causes a lot of conflicts and synchronization issues in the session state machine.

Network change testing In this test we both plug and unplug network cables as fast as we can. This stresses the Session Layer Daemon and the SNS. A problem is that the test speed is reduced by the "human factor".

Device movement testing This test is similar to the test above. We both plug and unplug devices containing endpoint directories, in our case USB Mass Storage devices, as fast as we can. This test is quite simple but when combined with the other tests the complexity increases.

When the implementation was finished, we could run all these tests simultaneously without any problems.

5.3 Conclusions

As shown in section 5.2 the implementation fulfills all use cases. The reliabilty testing shows that the implementation was reliable enough to show that the design is satisfactory.

6 Conclusions and future work

6.1 Summary

In this thesis, we presented design and implementation of a session layer based system for endpoint mobility. This section very briefly summarizes the work that we have presented in this thesis.

Related work There have been previous attempts to solve different parts of the problems that appear in mobile networks. We did not find that any of the work solved the problem stated in the beginning of this paper. At the same time we found that it existed good solutions to most parts of the bigger problem we tried to address. Therefore we felt assured that it was possible to design a system that solved the problem we wanted to solve.

Design and implementation The first part of the design work was coming up with the fundamental parts of our approach to mobility, namely Endpoints, Services and Sessions. When we had defined the basic building blocks, we designed a naming system used to name mobile objects, or as we call them, Endpoints. The scheme decided upon was mobileobject@domain, an example could be usbhd@example.com. As one of our main goals with the project was to have a disconnection tolerant system, we also designed a sub protocol used to assure that data transferred over a session is consistent even if the transport connection is corrupted. Here we decided to use a checkpoint based protocol. Then we designed the core of the Session Layer, the state diagram and the Session Management Protocol. Together they are used to keep track of witch action that should be taken when different events occur. Parallel with the design work we implemented a prototype of the Session Layer on top of the network stack inside the Linux kernel.

Testing We think the tests we have designed and carried out for our prototype have good coverage of problems that may arise. However, to really ensure the correctness of some aspects of our design, formal verification is required. That work will be carried out by Palmskog [7]. The results of our tests are satisfactory, we have not found any major flaws in our design or implementation.

6.2 Conclusions

Network mobility Network mobility is already possible with a lot of different systems, e.g. Mobile IP. Our creation of another system which is capable of this is not very revolutionary. This said, our design still has some advantages over other solutions such as Mobile IP. We do not have to deal with the problem of triangle routing, which can slow down communication. The only drawback that we can see with our design, is the need to change applications to support the new API. As previously argued this is needed to move beyond traditional mobility.

Device mobility We have shown that device mobility is possible and that our endpoint directories make it possible to name any device or data in a convenient way. We think that the possibilities here are unlimited. You could for example give a web camera an endpoint and name it. Whenever you plug it in to any computer or other networked device other users could connect to a service provided that will show the video stream from this camera. This would be a very easy extension to the implementation we have done and the same is probably true for a lot of other devices.

Temporal mobility Temporal mobility is one of the forgotten, but maybe the most useful, types of mobility. To be disconnected for long periods is normal for most mobile devices, yet very few designs take this into account. A TCP connection will die after around 15 minutes of inactivity, regardless of if the user use Mobile IP or not. We have shown that temporal mobility is possible with our design. Sessions will not die when the connection on the transport layer dies. Sessions are persistent and can even be stored to non-volatile memory which could be saved for years before they are resumed.

The design Our design shows that it is possible to combine, if not all, a lot of different mobility types in one common design.

6.3 Future work

In this section we have identified and described a few areas where further work is needed.

Naming The syntax of the naming system need to be thought through so that it does not conflict with any existing syntax, such as the URI.

Security To enhance security there should be a thought through public key solution to the session handle.

Session Gateway A session gateway that can route sessions between different networks should be designed. For example it should route sessions between private and public IP networks as well as different IP versions (e.g. 4 and 6).

User mobility It would be interesting to integrate the Session Layer into the user workspace, which could make it possible for a user to move his running software between different workstations.

Context aware mobility The Session Daemon should be improved in a way such that it can choose between different networks in an intelligent manner. It should be possible to have policies regarding as many aspects of different connections as possible. This could be anything from bandwidth to the cost for every byte that is sent.

References

[1] Ekwall, Richard, Urbán, Péter, Schniper, André, Robust TCP Connections for Fault Tolerant Computing, ICPADS '02, 2002

[2] Handley, M., Schulzrinne, H., Schooler, E., Rosenberg, J., SIP: Session Initiation Protocol, RFC 2543, March 1999

[3] ISO standard 7498-1, "Information Processing Systems - OSI Reference Model. The Basic Model", http://www.acm.org/sigcomm/standards/iso_stds/OSI_MODEL/ISO_IEC_7498-1.TXT

[4] Landfeldt, Björn, Larsson, Tomas, Ismailov, Yuri, Seneviratne, Aruna, SLM, A Framework for Session Layer Mobility Management, Eight International Conference on Computer Communications and Networks, 1999. Proceedings.

[5] Maltz, David A., Bhagwat, Pravin, "MSOCKS: An architecture for transport layer mobility", IEEE INFOCOM 1998 - The Conference on Computer Communications, no. 1, April 1998 pp. 1037-1045

[6] Nikander, P., Ylitalo, J., Wall, J., Integrating security, mobility, and multihoming in a hip way, Proceedings of Network and Distributed Systems Security Symposium (NDSS'03), pages 87–99. Ericsson Research NomadicLab, February 2003. 7

[7] Palmskog, Karl, Verification of the Session Management Protocol, 2006. To appear.

[8] Perkins, Ed. C., IP Mobility Support for IPv4, RFC 3344, August 2002

[9] Qu, X., Yu, J.X., Brent R.P., A Mobile TCP Socket, International Conference on Software Engineering (SE '97), San Francisco, CA, USA, November 1997.

[10] Schulzrinne, H. et. al., RTP: A Transport Protocol for Real-Time Applications, RFC 3550, July 2003

[11] Schulzrinne, H. et. al., RTP Profile for Audio and Video Conferences with Minimal Control, RFC 1890, July 2003

[12] SDP: Session Description Protocol, RFC 2327, April 1998, http://www.ietf.org/rfc/rfc2327.txt?number=2327

[13] Snoeren, Alex C., Balakrishnan, Hari, Kaashoek, M. Frans, Reconsidering Internet Mobility, HotOS-VIII

[14] Stocia, Ion, Adkins, Daniel, Zhuang, Shelley, Schenker, Scott, Surana, Sonesh, Internet Indirection Infrastructure, IEEE/ACM Transactions on Networking, Volume 12 , Issue 2, April 2004

[15] VoIP Protocols and Standards, June 30th 2006, `http://www.protocols.com/pbook/VoIP.htm`

[16] Wang, Yaoshang, Mobility Support for Networked Applications built in the TCP/IP Stack, Stockholm, April 2006

[17] Wesly, M., Eddy, At What Layer Does Mobility Belong?, IEEE Communications Magazine, October 2004

[18] Zandy, Victor C., Miller, Barton P., Reliable Network Connections, Proceedings of the 8th annual international conference on Mobile computing and networking, 2002

A Segment oriented data integrity

This design works like a simplified version of TCP. Instead of keeping the data consistent on the byte level it keeps it consistent on the segment level. A segment is a part of the data stream that has a segment number associated with it. The idea is to maintain a buffer of sent segments, and as we receive acknowledgments from the other side, we drop segments from this buffer.

The design There are two values to be decided upon. Segment size and how many outstanding segments we want to send before we need an ACK. The values chosen here will have a great impact on performance and should probably be different for different networks, as the optimal frequency of acknowledgments is dependent on the bandwidth and the latency of the network connection. To help explain this design we will pretend that these values are set to the following.

```
SEGMENT_SIZE = 200
NUMBER_OF_SEGMENTS = 2
```

This means that our buffer consists of two segments of 200 bytes each. We also need two pointers, one that points to the segment where we currently should put data and one that points to the first segment that needs an ACK. See figure 21.

Every time data is sent to the other host, it is added to the current segment of the local session buffer. We start at segment 0, and continue to fill all available segments. Every time we have sent some data over the network, we will check if the other side has sent any acknowledgments. In that case, we will note that the segment specified in the ACK has been received on the other side, and we can free that segment and reuse it.

If all segments in the session buffer become full (that means they are all unacknowledged), it means that there probably is some problem with the connection. In this case we will stop sending data. In a future implementation, this will mean that we suspend the current session until it can be resumed again. If a TCP connection is resumed, data transmission will start from the first unacknowledged segment.

Problems with this implementation

- It is impossible to choose a segment size and number of segments to use that is advantageous for all kinds of networks.
- This mechanism does not conform to the very much similar mechanism described in the OSI specifications for the Session Layer.
- Too high complexity for something solving a simple task.

These three problems make this design undesirable as the data integrity mechanism for the Session Layer. Therefore we have designed another mechanism for consistency which is described in the next section.

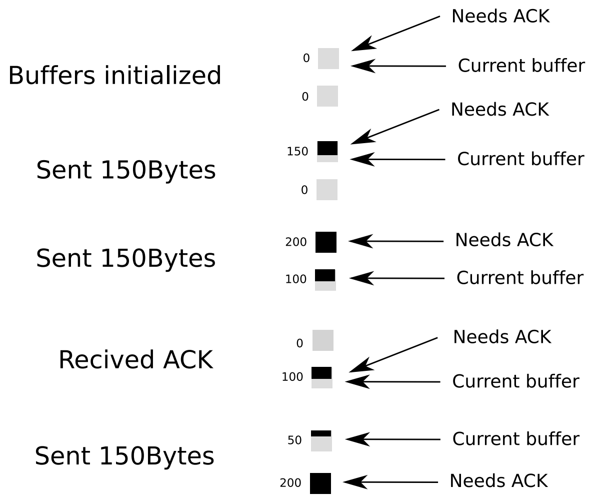


Figure 21: A sample usage of the session segment data integrity.