Implementing a DRM-Preserving Digital Content Redistribution System

The first Paradiso DRM prototype

Bsc. R. Gerrits
Vrije universiteit, April t/m October 2006

Abstract

This paper focuses on the implementation of a new kind of DRM proposed by Nair et. al. [1] and extended in [2]. Traditionally the process of online digital content distribution has involved a limited number of centralized distributors. This traditional model was extended by Nair et. al. by introducing a security scheme that enables DRM preserving digital content redistribution. Essentially consumers can now also buy the rights to redistribute content to other customers. In this paper the design and implementation of the Paradiso DRM is described in detail.

Table of contents

Introduction

Nowadays there is a widespread use of portable devices capable of rendering digital content, some of them also have peer-to-peer communication facilities. People are searching for ways to obtain digital content for their devices. Due to this, large worldwide peer-to-peer file sharing applications arise. These enabled consumers to easily obtain digital content without paying for content rights. This triggered content providers to start looking for new and secure ways for distributing their content while in the same time preserving the copyright and other digital rights. The most important requirement is to enforce the Digital Rights Management policies distributors set to protect digital content from being illegally copied or distributed without authorization.

The biggest challenge in DRM is to enforce DRM policies after contents have been distributed to consumers. The current approach is to limit the customer in its use of the content. The content is only distributed from an authorized provider to so-called compliant devices, which by construction are guaranteed to always observe the DRM policies associated with the digital contents rendered. In the traditional DRM the customer is only allowed to render the content. Due to this limitation the DRM schemes had only limited success.

Nair et. al. proposed in [1] and later extended in [2] to extend the business model by allowing distributors to sell not only the rights to render content, but also the rights to redistribute or resell content in a controlled manner. The resulting system is a network of peer-to-peer independent devices, each of them potential consumer, authorized distributor, but also an attacker. It is technically very challenging to implement the proposed protocols to enforce this new business model.

Building a prototype will give us more insight in the pros and cons so we can enhance the protocol. The unique concept of DRM-preserving content redistribution makes it harder to enforce the DRM policies. In the traditional DRM scheme there was only need for security when the content was transferred from a provider to the compliant device. Now we also need to guarantee rights preserving during device-device communication. This is where new problems arise. We need to be able to determine if the other device is trusted, and we need to enforce the DRM policies while at the other hand it is possible to redistribute content. Most of these problems are theoretically studied in the above mentioned papers. Please refer to them for detailed theoretical treatment of the problem. In this report the emphasis is on the design and implementation of a working prototype system.

An implementation point of view on the provider to customer (P2C), customer to customer protocol (C2C) and restore protocol (P2C') will be given in chapter one. Chapter two is an introduction to the platform on which the prototype was carried out. Chapter three covers the implementation of the proposed protocols. In chapter four the proposed architecture for the prototype is explained in detail. The implementation of the system gave us more insight in some practical aspects not handled in the original papers. These considerations result in some changes being proposed to enhance the security and efficiency of the system. They are described in the fifth and concluding chapter of this report.

Chapter 1

Paradiso DRM in an implementation point of view

1.1 The P2C and C2C protocol.

First take a look at the P2C protocol. In contrary to the mentioned papers the customer is called C instead of D and the rights are called ® instead of R.

1. owner(C) → C : P, h(M), ®
2. C → P : C, nC
3. P → C : $\{nP, nC, C\}_{SK(P)}$
4. C → P : $\{nC, nP, h(M), ®, P\}_{SK(C)}$
5. P → C : $\{M\}_{K'}$, $\{K'\}_{PK(C)}$, $\{®, nC\}_{SK(P)}$

The first thing to be noticed is that the reseller is not aware of what content and rights the customer wants until the 4th step is finished. It is also worth mentioning that the 4th step contains the payment message. So from an implementation point of view not only a failed communication can break the transmission (and enforce the restore protocol), but also when the content isn't available, corrupted on disk, or the rights can't be accepted. The design explicitly states that content and rights should be negotiated on beforehand. But failure is still possible if something happens in between. For now this should be kept in mind for the implementation.
In the papers [1,2] the content key is always denoted by K in both the P2C and C2C protocols. But there is however a difference between those keys. Each time content is requested at a provider it is re-encoded and a new content key is released. To emphasize this difference the content key will be called K' in the P2C protocol.
The C2C protocol is invoked when content is resold by a customer to another customer. So for convenience the seller of the content is called reseller, denoted by R. The rights are marked with an ' to emphasize that they differ from the rights obtained in the P2C protocol.

1. owner(C) → C : R, h(M), ®'
2. C → R : C, nC
3. R → C : $\{nR, nC, C\}_{SK(R)}$
4. C → R : $\{nC, nR, h(M), ®', R\}_{SK(C)}$
5. R → C : $\{M\}_{K}$, $\{K\}_{PK(C)}$, $\{®', nC\}_{SK(R)}$

When the P2C protocol and the C2C protocol are compared it is apparent that they are almost the same. The only difference is the key K. In the C2C protocol the content isn't re-encoded every time it is requested. The same key as the key received in the P2C protocol is transmitted. We assume the device hasn't enough CPU and/or battery power to re-encode content every time. So this is an important aspect to keep in mind during implementation. Only one small sub-routine should differ in the C2C protocol to get the P2C protocol.

1.2 The P2C' protocol

The recovery sub-protocol was introduced in [2]. The purpose of the recovery is to assure that the customer can get his content even if the 5th step of the C2C protocol failed. Just for convenience we will abbreviate the protocol to: P2C'.

1. owner(C) → C : resolves(C)
2. C → P : C, n'C
3. P → C : $\{nP, n'C, C\}_{SK(P)}$
4. C → P : $\{n'C, nP, \langle nC, nR, h(M), ®', R\rangle, P\}_{SK(C)}$
5. P → C : $\{M\}_{K}$, $\{K\}_{PK(C)}$, $\{®', nC\}_{SK(P)}$

The customer should assure the chosen nonce differs from the nonce chosen in the C2C protocol. The payment message from the failed communication is sent along with the 4th message. This combination of a new payment message and the old payment message is called a restored payment message.

1.3 Round up

If we look closely at all three protocols it is evident that the steps 2, 3, and 5 never differ. Only the values of the parameters change. The 4th step only differs between the R2C/C2C and P2C' protocols, but is the same between the R2C/C2C protocols. The complete message sent in the 4th step of R2C/C2C is, except for the signature, repeated in the 4th step of P2C'. This is also something useful to note for an implementation. There are plenty of possibilities to reuse several methods between the protocols.

The internal details of the protocol, like at which point exactly the payment message is stored, and what checks should be made at which point, are underspecified in [1,2]. Those details are however important for an implementation of the protocol. So they will be covered in the design of the implementation in chapter 3 and 4. But first the platform on which we tested the prototype will be discussed.

Chapter 2

Choice of platform

A prototype enables us to demonstrate the feasibility of Paradiso DRM. This chapter describes our search for a feasible platform to implement the prototype on. Some details about the platform we eventually decided to use will be discussed here. To be able to demonstrate content distribution and peer-to-peer communication, one of the prerequisites was that the platform should support unencrypted wireless communication. Technically speaking infrastructure mode should be enough, but ad-hoc mode is preferable. The first part of the section will focus at the iPod [19] and the last section will discuss the platform on which the prototype was tested eventually.

2.1 The iPod

The main reason for choosing the iPod is its availability. It would certainly demonstrate the usability of Paradiso DRM. Theoretically everyone could then choose between iTunes and Paradiso DRM. The disadvantage however is that the iPod is a closed system and that it is very hard to write custom software for it. There are however projects aimed at opening the iPod. There is for example RockBox [3]. This project is based on a custom written firmware for the iPod to enable running of the RockBox sofware. There is also a project called iPodLinux [4] which is partly based on the same custom written firmware for the iPod and enables the user to run Linux [20] on the iPod. It is expectable that this eases communication between 2 iPods.

2.1.1 USB Host driver

The USB driver support in these projects however is very limited. It only supports removable storage, which is one of the simplest forms of USB communication. To connect a wireless dongle or other network peripheral to the iPod USB Host is needed. The iPod chipset [16] is capable of USB On The Go [17]. USB OTG is a negotiation protocol to decide which of the two peers will become the host. There are also products available which prove that the iPod is capable of Host support. So basically the problem boils down to three sub-problems:

i. Get a physical plug to insert in the iPod to connect a USB device to it
ii. Get USB Host support working
iii. Get the connected device (wireless dongle for example) working

The first problem arises because the iPod is physically small. It doesn't have a standardized connecter. The cable delivered with the iPod has a normal USB connector. The data sheet of this USB cable could be examined in order to change the connector to a Host connector. One iPod could then be programmed to act as a host and the other iPod as a client. By doing this, there is no need to program the advanced USB OTG protocol.
The second problem is the hardest. Even though Apple firmware has built in Host support, providing us with a good starting point to implement our own custom firmware, the firmware is completely closed source. This leaves three possible solutions for this problem:

1. Reverse engineer the Apple USB Host firmware
2. Get datasheets from processor vendors
3. Ask for the original USB Host firmware at Apple

The first solution requires more time and resources then available for this project. Reverse engineering the firmware costs a lot of time and is a very complex task. It also requires some experience with reverse engineering techniques and knowledge of ARM assembler. So this was just not an option for this project. What should also be considered are the legal implications. Reverse engineering is not always legal so if no precaution is taken, then this could result in a lawsuit.
If we could get some datasheets from the vendor it would ease the work. The main task then is reverse engineering the system board of the iPod. For this to work an iPod should probably sacrificed, but it becomes much easier then to port an existing USB Host driver to fit the iPod architecture. This project sounds very reasonable, there is however a chance that the processor vendors will not release the data sheets of the processors. And the same goes for this solution as for the first solution: it will cost a lot of time.
The last solution is asking Apple for the original USB Host firmware. Since it was felt asking for such proprietary information would not meet with success, this line was not pursued.
When the USB Host driver is finally working another quite hard sub-problem arises; porting the USB wireless dongle driver. Most device drivers are based upon X86 architecture so depending on the driver's internal structure several parts should be changed. It is very uncommon for drivers to be written specifically for the ARM architecture.

So the overall conclusion is that USB Host support on the iPod is very hard and hence it was not attempted to accomplish this.

2.1.2 IP over FireWire

There is however another way to get two iPods to communicate. This is done by IP [20] over FireWire [5]. FireWire is supported natively by iPod firmware. It doesn't involve wireless communication, but it is peer-to-peer.

It is possible to use this, but several problems still remain. The iPod is a closed system so writing custom software for it is still a hard problem. If communication over FireWire fails then our project doesn't have any alternative, so it fails. Its also still unclear if there are any legal issues involved for using the iPod for such an experiment. In short it was decided not to use iPod as our development platform.

2.2 The Neuros

Fortunately an alternative is available: the Neuros DM320 Development Platform. It includes a feature-complete embedded Linux environment. There is a toolkit available which incorporates everything needed for user interaction, multimedia playback, and hardware support. At the moment of researching, a working USB Host driver was available and the device itself was still in a development stage. The development board is a prototype for a whole new range of products. It however doesn't have any screen or internal power supply at this point, but all programs written for the development board will be able to run on the other products in the product lineup. It is not stand-alone because it is externally powered and a separate device is still needed to sent commands to it. There however are a lot of advantages:

1. Completely open source, writing software for it is easy
2. USB Host driver present and working properly
3. A working Ethernet port is available
4. Active community which is willing to provide support
5. Documentation available about how to setup a development environment

The only thing work needed to get wireless working would be to port a wireless dongle driver to use with the device. No one has ever attempted this so we would be the first. Porting itself shouldn't be too hard, but it all depends upon the chosen dongle and driver availability. It's trivial, but the chosen USB wireless dongle should have an open source driver available.

2.3 Round up

The overall conclusion was that choosing Neuros gives a much more realistic future prospect compared to the iPod. If porting the wireless dongle fails then there is still the normal ethernet port to realize peer-to-peer communication. Furthermore there is a much more active and supportive community. The iPodlinux community consists mainly of people who are trying out Linux on the iPod while the Neuros community consists of people who are planning to bring a professional and serious product to the market and make profit from it. So we decided to go for the Neuros.

A detailed report on the process of getting the wireless driver to work with Neuros is provided as Appendix A.

Chapter 3

Architecture of the Paradiso implementation

The aim was to create a modular program where each component implements a certain task of the protocol and maximize reusability. Another goal was to write the program as structured as possible. This lowers the chance for programming errors in the security parts of code and also eases third party changes or enhancements to the code. In contrary to what would be expected, the design is based on a two-tier architecture. So there basically is one data + application layer and one interface layer. This ensures us that data can only be accessed through the so-named Security Manager. By using a three-tier architecture we would have allowed the data and application/security layer to run on a different platform with the implication that the data layer wouldn't be protected. In theory this isn't a problem because all critical data in the data layer is stored encrypted. Even when one wants to play or view the content it would be decrypted by a hardware chip, on-the-fly. It was however a design decision to use the more secure two-tier architecture. In the prototype four modules can be distinguished.

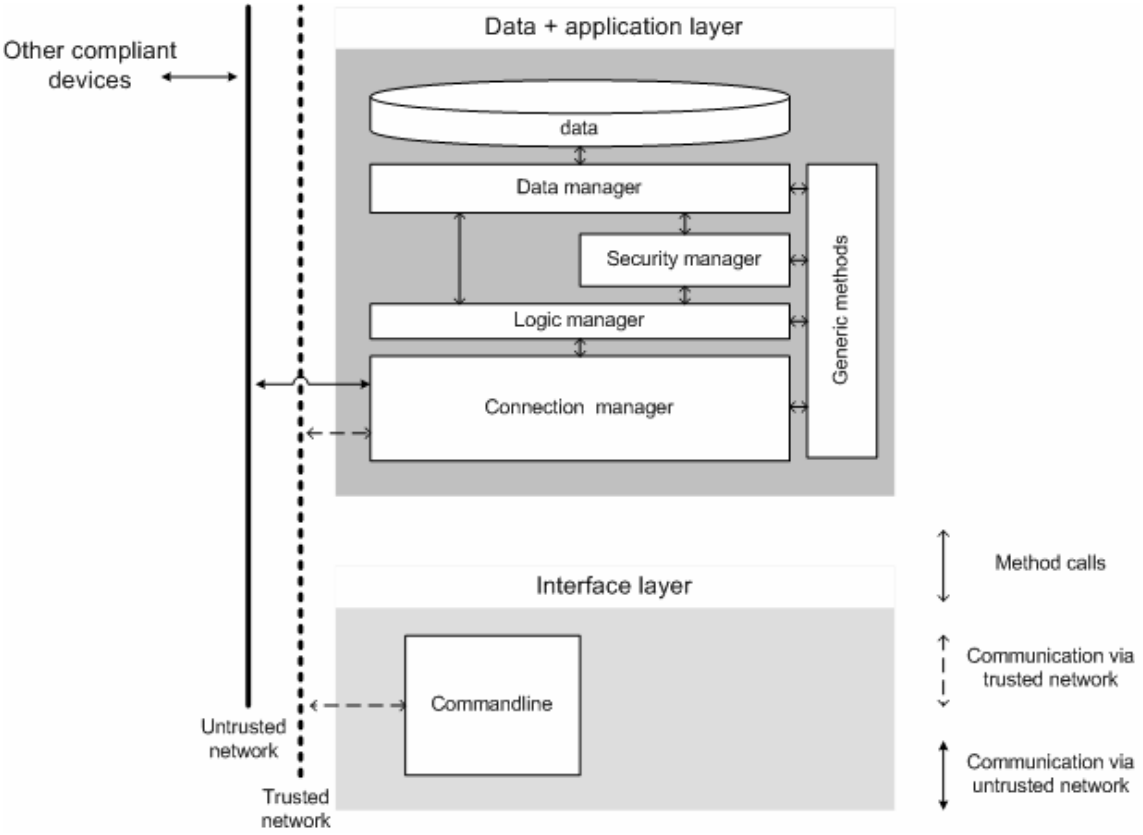

Fig. 1. Global view of Paradiso DRM architecture

Module 1: Connection manager (CM)
This part is responsible for handling all communication to the outside world. Every incoming new connection is handled by a separate newly forked process. In this way, there is no need for complex mapping between connections and data, especially sessions, belonging to them. It really simplifies the structure of the program. Synchronization between those processes is handled using semaphores.

Module 2: Logic manager
This part of the system is responsible for maintaining a correct logical flow of the system. It makes decisions based on the response from certain actions performed by the security manager and data received from the interface layer or connection manager. This part cannot be integrated with the security manager because of reusability and structure reasons. The code in the logic manager gives a perfect view of the data flow.

Module 3: Security manager (SM)
This is the most crucial part of the system. It is responsible to ensure that it is impossible to perform illegal actions on the data. It is session and state based. The security manager alters sessions supplied to it and responds to them in the correct manner based on the current state of the sessions. The sessions itself aren't protected so the security manager assumes they cannot be tampered with.
In order to get a compliant device a Trusted Platform Module (TPM) specification [3] compliant piece of hardware should be used. The hardware is tamper-resistant and in short eventually secure the program from tampering. The security manager is the only module allowed to communicate with the embedded TPM compliant hardware module. It is responsible for validating the signatures, signing messages, decrypting and encrypting messages. An asymmetric key pair is used from which the private key is stored in the TPM compliant hardware.

Module 4: Data manager (DM)
The data manager handles all disk operations. Because the prototype doesn't use a database management system, the data manager is also responsible for keeping track of the files. The data manager is used by both the logic manager and security manager. Most calls however involve security checks which can only be performed by the security manager. So almost all calls to the data manager originate from the security manager.

And finally there are the *generic methods*. This isn't a separate module, but just a collection of several methods used by more than one module of the program. There are methods to handle error messages, handle semaphores, read and write data to sockets, initialize and listen to sockets (both for udp and tcp). All these methods are completely stand-alone and aren't aware of data structures belonging to Paradiso DRM itself. These methods can be reused in other programs without modifying them.

Do note that communication between the commandline and the device, and communication between devices, is over TCP [22] sockets. A separate piece of code is responsible for network scanning using UDP[23]. As seen in figure 1 the connection between the commandline and device is over a trusted network. Connections between compliant devices is over an untrusted network. The commandline implementing the interface layer could theoretically be located on the device itself. However due to the absence of screen and keyboard peripherals for the Neuros, the commandline is located on an external device.

3.1 The connection manager
The connection manager handles all socket communication. It is fully struct/data aware so it also handles the host to network and network to host conversions. It also adds a completely abstract error handling layer to the communication. Each time some struct, integer, or long is written to the socket it is preceded by a 2 byte value to indicate if normal data can be expected. If a struct indicating an error is written to the socket the value is set to indicate that an error struct should be expected. So when something is read from the socket the connection

manager can determine what he can expect to receive. It allows the following program structure:

```
if( signature is correct )
    sent information struct
else
    sent error struct
```

and at the corresponding receiving side:

```
if( receive information struct )
    process information struct
else
    indicate error
```

Thus this simple error layer, added by the connection manager, really simplifies things for both the sender and receiver.

3.2 The logic manager

The logic manager is divided in three client/server modules, all responsible for separate individual tasks.

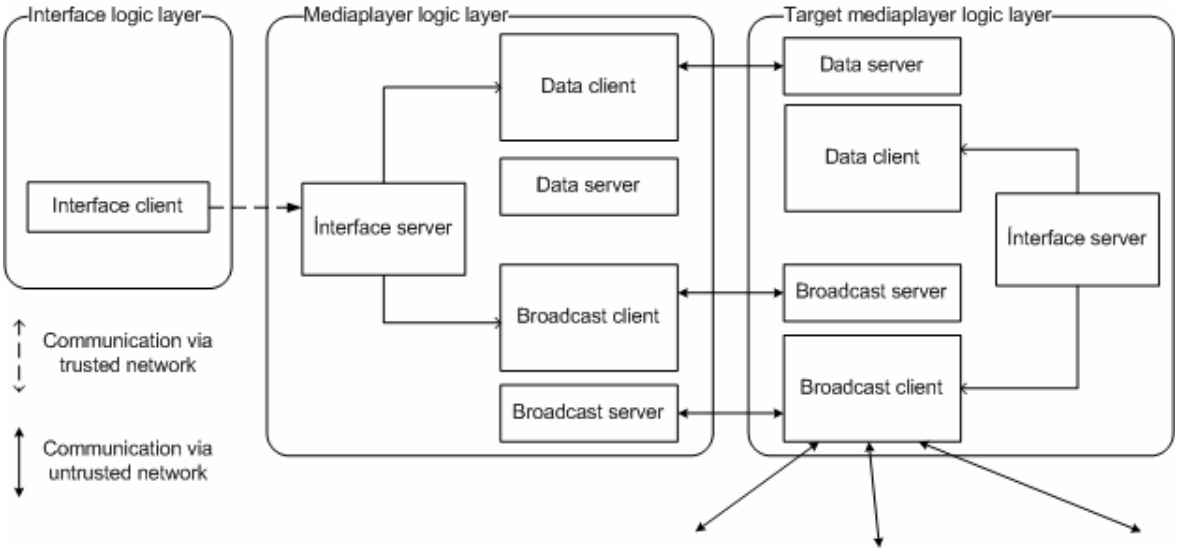

Fig. 2. Logic manager communication overview

3.2.1 Interface server (IS) and commandline

The interface server listens to the interface socket for incoming requests from the interface client. When a request is received, all necessary data, depending on the type of request, is read from the interface socket. The request is then processed directly or, if needed, forwarded to the data or broadcast client. The interface server is capable of handling the following requests:

- Get content (forwarded to data client)
- Get broken sessions
- Restore session (forwarded to data client)
- Request list (forwarded to data client)
- Request update (forwarded to data client)

- Request cash (forwarded to data client)
- Request scan (forwarded to broadcast client)
- Play content
- Request public key

Some commands, which can only be sent to a provider, are as follows:

- Add new content
- Revoke device

The requests which aren't forwarded to other client/server modules will be briefly discussed here. The "Get broken sessions" request reads all open sessions from disk. It generates a list of all the important information in the session and writes it to the interface socket. The interface client can decide what to do with this information. It could for example, request a restore session for each of these sessions.
The play content first reads the content hash from the socket. It forwards the hash to the security manager to search for any content matching this hash. If the content is found the security manager will theoretically send a command to the TPM to decode the content and forward the unencrypted data directly to the hardware to play the content. In the prototype, due to absence of secure hardware support, a temporary unencrypted file is created in the unsecured storage.
The interface client, from here on referred to as commandline, is used to sent commands to the interface server. The communication between the commandline and interface server is kept as simple as possible. The commandline should be seen as completely independent from the Paradiso DRM prototype. In the actual implementation the connection manager module is however reused in the commandline. Most of the generic methods are reused in the commandline also.

3.2.2 Data client/server (DC/DS)

The data client receives its commands from the interface server. It then connects to a data server located on a second mediaplayer. For most commands, it receives data from that data server, checks it at the security manager and then requests data from the security manager. It then sends this data over the socket and starts listening for the next message. The data client and server are responsible for handling the following commands:

- Get content (P2C and C2C protocol)
- Restore session (P2C'protocol)
- Request list
- Request update
- Request cash

The last three commands are part of the extensions to the original Paradiso DRM. These will be discussed in section 4.2.

3.2.3 Broadcast client/server (BC/BS)

The broadcast server listens for UDP messages at a specific port. When a messages arrives, it immediately replies to the message with the public key chain and the name of the device. The broadcast client on the other hand is capable of broadcasting an UDP message over the network. After the broadcast, it starts waiting for the replies (sent by the broadcast server). Each incoming public key chain is immediately checked and disposed if it can't be trusted for

whatever reason. So for example, if the public key of a device is listed on the DRL, then it won't be able to see any other device.

3.3 The security manager and TPM

The security manager is the key component of the prototype it not only protects the data layer, but also leads the data server and client in the right direction. It also guarantees that no illegal actions are performed by the TPM. For example, reading some specific content should not be possible when the content will be sent to an untrusted device or when we haven't received a correct payment yet. The security manager guarantees us that the payment is stored to disk before the content is requested. A security manager session can only be used once to obtain the content. After that, the only resolution for the customer should be using the restore protocol. The security manager guarantees this with a session and state based architecture. The state based approach guarantees that certain actions are only possible when certain conditions are met, and the session based approach enables us to save a security manager session to disk so it can be restored at a later point.

The security manager is also the only module allowed to communicate with the TPM directly. So all actions that require any kind of encryption/decryption/signing or hashing will be processed via the security manager. For this reason, there is a distinction between the use of encryption for internal usage and for external usage. By internal usage we mean for example, the provider who is updating his internal DRL or the interface server who is requesting the local public key. As soon as a second device is involved, we need to ensure this device is trusted. So a normal session can only be created if the supplied public key of the second device is trusted. Each incoming public key will always be checked. For example, when a broadcasted scan message is received at the broadcast server the corresponding public key is first fed to the security manager to start a session. Only when this key is verified and trusted, can the broadcast server get the local public key and reply with this key to the scanning device.

All state transitions are recorded in the following state transition diagram. The green letters indicate provider only methods. These methods are only compiled into the code when a provider version of Paradiso DRM is built.

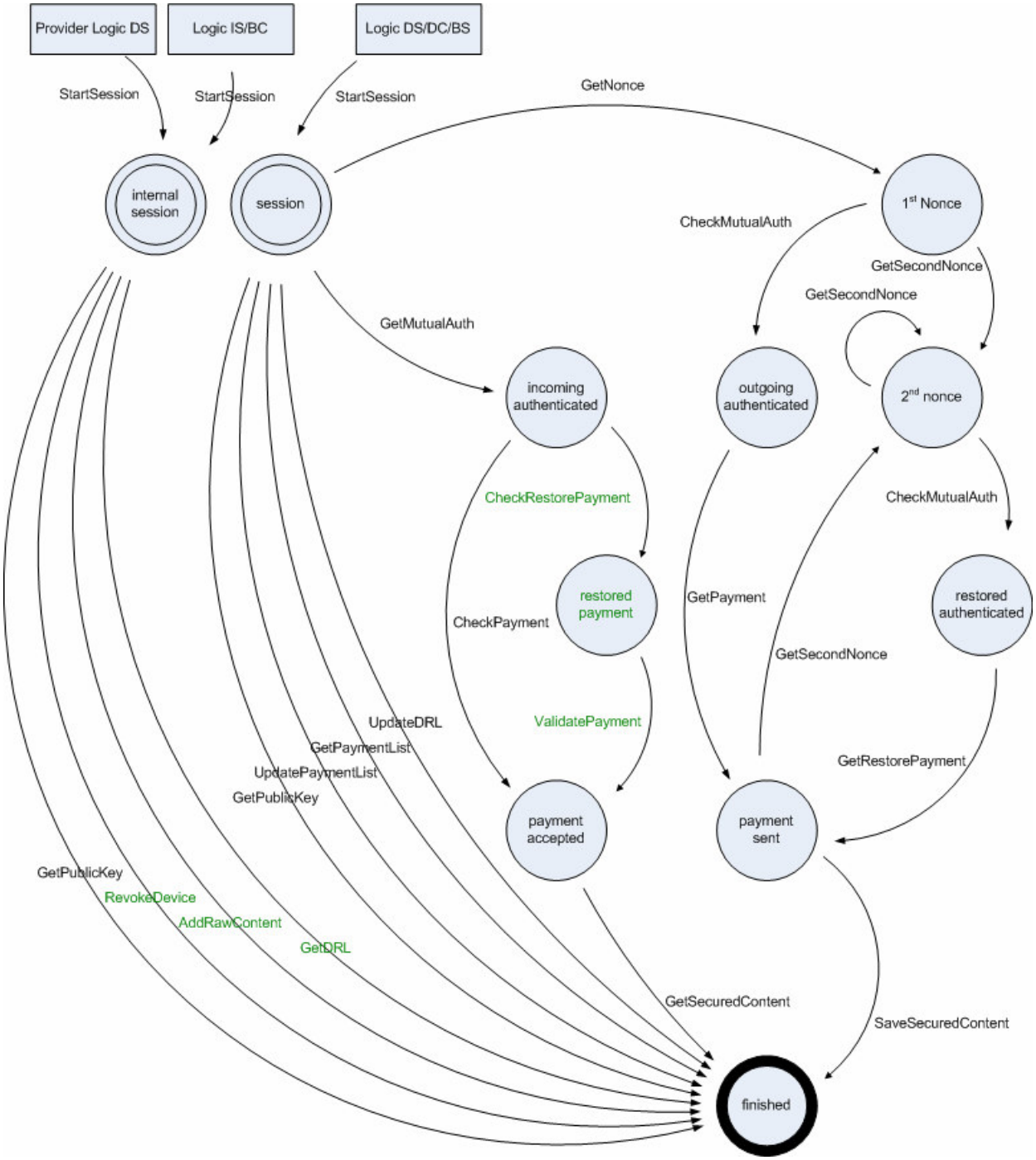

Fig. 3. State transition diagram of a security manager session

3.3.1 TPM emulation

The chosen platform (the Neuros) does not support secure hardware. For this reason the TPM functionality is emulated in software. The software implementation emulates a secured scratch memory, a non-volatile secured memory block and storage of the public license organization RSA key. The non-volatile secured memory is emulated with a write-through cache. As soon as the Paradiso DRM is started the secured memory and licensing organization public key are read from disk and stored in memory. Any writing to the secured memory will result in a disk write of the changes.

The OpenSSL library [24] was used for implementing the cryptographic functionality. SHA-1 is used for secure hashing, AES-256 for data encryption and RSA as public key infrastructure.

3.3.2 Content license files

The prototype also implements the Content License Files (CLF) completely. The CLFs provide the mechanism to guarantee that it is not possible to manipulate with the content stored on the device. All critical information, like the AES key of the encrypted content, are stored in these CLFs. The easiest way would have been to store these CLFs directly in the secured memory of the TPM. The size of the secured memory however, is restricted and CLFs are quite large. The solution for this problem is already presented in [1] section 7.1. The CLFs should be stored in unsecured memory and a SHA-1 hash of the CLF should be stored in secured storage. Each time a CLF is used, the corresponding hash in the secured storage should be checked to verify that the CLF hasn't been tampered with. Furthermore the CLF should be kept in the secured scratch memory when it's changed. In this way any tampering with the CLF during content exchange can be prevented. In [1] it is estimated that the CLF, including metadata, will use 536 bytes of data. In the prototype the CLF uses 585 bytes of data, consisting of:

content_info struct:	291 bytes consisting of meta data, original content size, the content hash and rights
AES key:	256 bytes containing the RSA-public-key-encrypted AES-key
AES iv:	32 bytes containing the initialization vector needed for the AES decryption
index:	2 bytes fileindex needed by the data manager to locate the file
encrypted_size:	4 bytes to indicate the encrypted size of the data

The content info struct contains the following information:

hash:	20 bytes containing the SHA-1 hash of the unencrypted content
type:	1 byte to indicate the type of content {audio, video, image, text, software}
title:	128 bytes to store the content title
author:	128 bytes to store the author or artist of the content
rights:	10 bytes of current rights to this content {resell count, resell depth, resell total, price}
size:	4 bytes containing the original, unencrypted, size of the content

In the current implementation the rights are kept as simple as possible. It consists of three integers and one long, representing the resell count, maximum resell depth, resell total and price. In [1] it was proposed to use authorization and policy languages like XACML or XrML. These languages are however very complex and there was no time to study and implement these within this project. Besides, the full implementation of those languages may be an overhead that is best to avoid. For this reason the rights are kept very simple. The resell count, resell depth and resell total are something new introduced in this project. The resell count indicates the maximum number of times the content can be resold. If a customer request a resell count of X then the resell count at the reseller is decremented by X plus 1. It namely resells the content to the customer once, and the customer gets an additional X times to resell the content himself. So a customer who doesn't want resell requests X=0. The resell depth is given an initial value of zero or higher. If a customer obtains content then he gets a resell depth decremented by one. When a reseller has some content of which the resell depth is almost depleted (=1) then a customer is not allowed to request a resell count higher than 0. So

the customer will not be able to buy content for resell purposes. This is very useful when the provider wants to control the number of resellers of some content. He can set the resell depth to, say two, giving him the possibility to control which resellers can and which cannot obtain the content. The resell total is used to guarantee atomicity. The counter is incremented by one each time the content associated with the rights is sold.

3.3.3 RSA key chain

As described in the papers [1,2] there are several actors involved in determining if a device is trusted. The complete chain of trust is implemented in the prototype. The struct is called public_key and contains the following:

```
device_name              the name of the device chosen by owner
type                     the device type, can be RESELLER or PROVIDER
device_pkey              the RSA public key of the device
signature                manufacturer his signature on device public key
manufacturer_pkey        the RSA public key of the manufacturer
manufacturer_signature   licensing organization his signature on the
                          manufacturer's public key
```

The RSA public key of the licensing organization is embedded in each device.

3.3.4 Atomicity guarantee

To prevent inconsistencies in the system, and to prevent anyone from obtaining the same content twice by paying once, or to manipulate the content rights at free will, the security manager has to guarantee atomicity of operations. There are three cases when the need for atomicity is critical. The first case is when the provider adds new content to the system. The second case is when a payment message is received and we update our local CLF. And the third case is when secured content is received and we create a new CLF and store both of them to disk. We accomplish this atomicity by performing actions in a predefined sequence. The sequence allows us to detect failures between each of the atomic steps. When an inconsistency is detected it can be determined exactly what happened and some roll-back functionality can be triggered or the remaining actions can be completed.

Case 1: New content is added to the system

Guaranteeing atomicity for this case is the easiest. We identify the following atomic operations:

1- get the file index
2- generate and store hash of CLF
3- store the CLF to disk
4- save the content to disk

In the first step the location where the CLF and content files should be stored on disk is determined. Details about it are in section 3.4, for now it is sufficient to know that the file index is used as location for files. In the second step the hash of the CLF belonging to the new content is calculated and stored to the secured memory. The location of the hash in secured memory is the same as the file index. In the third and fourth step the CLF and content are stored to disk respectively. The following overview addresses failures between any of these steps:

failure between 1/2

This is not problem, there is no inconsistency. The file index is just calculated and stored in memory.

failure between 2/3

Contrary to what one could expect, this isn't a problem either. The hash which was written to the secured storage will just remain unused. The hash is only used if there is a corresponding CLF stored to disk at the earlier determined file index location. Since action three wasn't performed, no CLF exists on the disk.

failure between 3/4

This creates a simple inconsistency between the content and CLF directory. The content matching the CLF isn't stored. Because the content wasn't stored the CLF is completely useless, so just removing the CLF from disk restores the system consistency.

Case 2: Payment message is received and local CLF is updated

Guaranteeing atomicity for this case is the hardest and most crucial.

pre-condition: customer always requests a resell total equal to the stored resell total

1- determine file index of payment message
2- store the payment message
3- store the hash of the changed CLF
4- store the CLF

If the pre-condition is not met the payment message will be rejected before any of the steps are performed. After this check is made the file index can be determined for the payment message so it can be stored to disk in the second step. It contains essential information to detect and recover from a failure later in the process. In the third step, the CLF hash is calculated and stored to secure storage. The resell count of the CLF rights is decremented by the requested resell count X + 1 as described earlier. Furthermore the resell total is incremented by one and the resell depth and price are left alone. In the final step the CLF is stored to disk.

failure between 1/2

No problem, file index only resides in memory.

failure between 2/3

During initialization of the device, all the stored payment messages are checked in order to detect this failure. In each payment message the resell total is stored and should be lower then the current resell total in the CLF. If there is a payment message which has a resell total that is equal to the CLF then a failure between 2/3 or 3/4 occurred. If the CLF hash in the secured memory matches the CLF on disk then it is a 2/3 failure, otherwise a 3/4 failure. This is guaranteed because a content buyer is obliged to request the same resell total. Recovering from this failure is easy, actions 3 and 4 can be performed once the error is detected without any side-effects.

failure between 3/4

In this case the CLF hash stored in secured storage doesn't match the CLF stored in normal storage. This failure can indicate 2 things: 1. someone has tampered with the CLF in normal storage, 2. The hash was updated, but the CLF on disk wasn't. Detecting this failure is once again done by scanning the payment messages for matching resell totals. If a resell total matches and the CLF hash doesn't, as described above, then a 3/4 failure could have occurred. The only thing left to do now is verifying if the CLF stored to normal storage isn't tampered with. The rights that should have been applied to the CLF are stored in the payment message. So the CLF is changed according to these rights and then the hash is calculated and compared with the hash stored in secured memory. If it matches then the CLF isn't tampered with and step 4 can be completed afterwards. Otherwise the CLF has been tampered with.

Case 3: Save secured content and update the CLF

When new content comes in a new CLF should be created and both the CLF and content should be stored to disk. This case corresponds a lot with the first case. Please keep in mind that the method where this atomicity is guaranteed can be used by both the P2C/C2C and P2C' protocols.

1- get the file index for the CLF and content
2- save security manager session (including content key) to disk
3- generate and store hash of CLF
4- store the CLF to disk
5- save the content to disk
6- remove the session

failure between 1/2

No problem, file index only resides in memory.

failure between 2/3

No problem either. The session is stored to disk so the customer is able to invoke the P2C' protocol and restore the session. The existence of a session in the session directory can however indicate this failure or a failure between 3/4, 4/5 or 5/6.

failure between 3/4

In this case an unused hash is stored in the secured memory. This isn't a problem because there is no corresponding CLF stored to disk.

failure between 4/5

This failure creates an inconsistency between the content and CLF directory. The content matching the CLF isn't stored so the CLF is useless and can be removed without problem.

failure between 5/6

The session is still stored on disk which enables the customer to start the restore protocol at the provider and obtain the content twice by paying once. To be able to detect this error the content key is stored in the security manager session. At device startup, and when the restore protocol is started, the device will search if there is a CLF stored on disk with the same content key as stored in a restorable session. If a match is found the session is removed.

The recovery procedure of the last step creates a new problem. Imagine a reseller who has obtained some content with a high resell count and resell depth. At a certain moment the reseller runs out of resell counts. He contacts a nearby reseller and tries to obtain the content, but the connection fails. The reseller he is obtaining the content from has, by coincidence, obtained the content within the same redistribution tree (so the content key is the same). Now when he tries to invoke the P2C' protocol the session is recognized as a 5/6 failure and removed.
This isn't an issue for the current prototype because it doesn't allow, during the P2C/C2C protocol, to obtain the same content twice. This identification is done by looking at the content hash. The content hash is used to uniquely identify content. When someone triggers the C2C protocol it only sends the content hash to identify the content he is requesting. So if this content was stored to disk twice then there should be any way to decide what content should be used. In order to prevent this situation no double content is allowed on the device. If this requirement is removed however the above problem arises. In order to solve this problem some value should be added to the CLF to uniquely identify between contents. Another, less neat, solution could be to check for corresponding content keys in advance. Then the system could revoke buying content multiple within the same redistribution tree.

3.4 Implementation of the data manager

The data manager is used for all disk access. No module is allowed to access the disk on its own. During installation a directory called *data* is created. All data needed for Paradiso DRM to run correctly is located in this directory. It contains the following subdirectories:

```
clf
content
payments
respayments
sessions
keys
temp
tpm
```

And the revocation list, called drl.data, is located in the root directory. The first five directories make use of file indexes for file organization. File indexes are a simple solution for file organization. Each file in the directory gets an incremental number: the file index. The filename is equal to this file index (padded with zeros at the beginning to a total length of 4 characters).
The files in the *clf* and *content* directory are linked to each other. So each CLF has one corresponding file in the *content* directory with the same file index. The payments, *respayments* (restored payment messages) and *sessions* directories are independent of each other.
Beside the file index format, there is also another file format called custom format. In the custom format the data stored in the file is preceded by the size of this data. When the data manager reads a custom format file it first reads the size, then allocates memory the data in the file and then reads all the data.
There is one problem when using file indexes and multiple processes. A new file index is calculated by reading the directory contents. The total number of files are counted and the largest file index currently in use is determined. This information is checked for consistency and then the new file index is determined. But if multiple processes are involved it is possible that two processes determine the same file index and start writing at that location. For this reason process synchronization is needed. The prototype uses semaphores to accomplish this.

Finding a certain content file is realized by searching the *clf* directory. Each CLF starts with the hash from the corresponding content. So each CLF is compared with the hash of the content where we are searching for. As soon as the CLF has been found we also know the file index of the content because all content/CLF file indexes correspond.
The *keys* directory contains all needed non-private keys of the device. In fact the whole public RSA key chain is stored in this directory. The *temp* directory is used to temporarily store decrypted data during playback. This playback functionality was only added for demonstration purposes, so the *temp* directory itself is also temporarily. The *tpm* directory contains all private information needed by the TPM.

3.5 Internal error handling

The prototype provides a lot of internal error handling which is worth a brief overview. There are several security checks that can fail and having a good error handling structure eases the resolution of programming errors. Each module of the prototype has its own set of error packages. So The error packages are declared in the header file of each module. There is one generic method, called quit, which is able to print the error message and the name of the module, or sometimes called layer, where the error code originated from. A programmer can determine the correct module name by looking at the error code. All error codes are organized as follows:

```
Error code layout: XXCC

XX denodes the layer and CC the error code
XX=00   generic errors
        CC=0D   generic errors
        CC=1D   socket related errors
        CC=2D   server related errors
        CC=3D   semaphore related errors
XX=10   connection manager errors
XX=11   provider connection manager errors

XX=20   logic data server errors
XX=21     "   interface server errors
XX=22     "   data client errors
XX=23     "   broadcast server errors
XX=24     "   broadcast client errors

XX=30   security manager errors
XX=31   security manager TPM errors

XX=40   data manager errors
```

So for example, the error code -2117 originates from the logic layer interface server. From the interface server header file, it can be seen that it indicates that the interface server failed to read the device name from the interface socket. When an error occurs, each layer prints its corresponding error message. This makes it easy to trace an error:

```
1375> security manager tpm :-3139: Signature fed to TPM appears to be
incorrect.
      Errno: 10 - No child processes
OPENSSL error: error:0407006A:rsa
routines:RSA_padding_check_PKCS1_type_1:block type is not 01
OPENSSL error: error:04067072:rsa routines:RSA_EAY_PUBLIC_DECRYPT:padding
check failed
1375> security manager :-3013: Mutual authentication signature incorrect.
      Errno: 10 - No child processes
```

```
1375> logic data client :-2204: Verify mutual authentication failed.
      Errno: 10 - No child processes
1375> logic interface server :-2109: Data client failed to process restore
request.
      Errno: 10 - No child processes
```

In this trace, the signature check on the mutual authentication message has failed. The signature check itself is handled by the TPM, which doesn't know where the signature is used. The security manager however does this usage and prints the corresponding error message. This trace makes life easier when debugging. In the prototype all error messages are printed to standard output and there is no distinction between warnings / critical errors / input errors etcetera. Distinction between error messages could be used to set an error reporting level during compile time. This could be improved in the future.
Declaring the error packages in several header files creates some difficulties during linking/compilation. Error packages should be defined only in the corresponding header files to keep it manageable. These header files are included by several parts of the program. So what we want is only declarations in the header files and the definition in some other header file which is only used once during linking. But by doing this one single file is create in which all the error packages reside, which makes it less manageable. The Minix source code [10] uses a neat trick to solve this problem. This solution is used also for the prototype. One single file, called def.h, includes all the available header files. In each header file the error packages are declared as follows:

```
DEFERROR(E_ERROR_ID, "Error string", -1000);
```

And the DEFERROR compiler macro is declared as follows:

```
#ifdef _DEFINE
 #define DEFERROR(name, message, code) error_package name = {message, code}
#else /* DECLARE */
 #define DEFERROR(name, message, code) extern error_package name
#endif
```

So at top of the def.h file the _DEFINE compiler variable is set. This triggers all the header files to start defining the error packages instead of only declaring them. The same trick is used for global variables.

3.6 Prototype directory layout

The prototype consists of six directories:

```
> data_manager          the source code for the data manager
> security_manager      the source code for the security manager
                         and the TPM
> logic                 the source code for the broadcast client/server,
                         data client/server and the interface server
> connection_manager    the source code for the connection manager
> setupfiles            all data needed to install the program, should
                         normally only be available at the manufacturer
> tools                 some tools needed to be able to install the program
```

A seventh directory *data* is created after installation. The contents of this directory are already described in section 3.4.

Several files are located in the root directory:

```
data.h
interface.h
paradiso.h
paradiso.c
commandline.c
Makefile
INSTALLING
```

The data header file contains all the structs used for communication between the data server and client. The interface header file contains all structs pertaining to communication between the interface server and client. The interface header file however also contains several structs which are used also by the data server and client for communication, or part of other structs. The most important structs, like the public_key struct, are stored in the interface header file, but used all over the program. The interface header file contains exactly all the structs to create an interface client. The paradiso header file makes all the needed includes for the main program located in paradiso.c. The only task the main program performs is initializing the security manager and TPM, setting the correct signal handlers to be able to properly shutdown the program and fork off the interface server, data server and broadcast server. Then the main program begins to wait for someone to cancel the program. If this command comes in it properly shuts down the child processes, de-initializes the security manager and TPM, and then exits. The commandline C-file contains the interface client. Finally, the Makefile is needed to compile and link the program, and the INSTALL file where the usage of the Makefile is explained.

Chapter 4

Implementation of the protocols

4.1 Implementation of P2C and C2C

The implementation of the P2C/C2C protocol is modeled using two sequence diagrams. The first diagram, figure 5, represents the sequence of methods called at the customer side, and the second diagram, figure 6, represents the reseller or provider side. Both diagrams focus at the data client and data server modules. In the customer view diagram details about the interface client/server communication are omitted. In this diagram you can only see the internal communication between the interface server and data client. As the communication between the interface server/client is simple it is not modeled using sequence diagrams. Furthermore communication between the security manager and the TPM are also omitted.

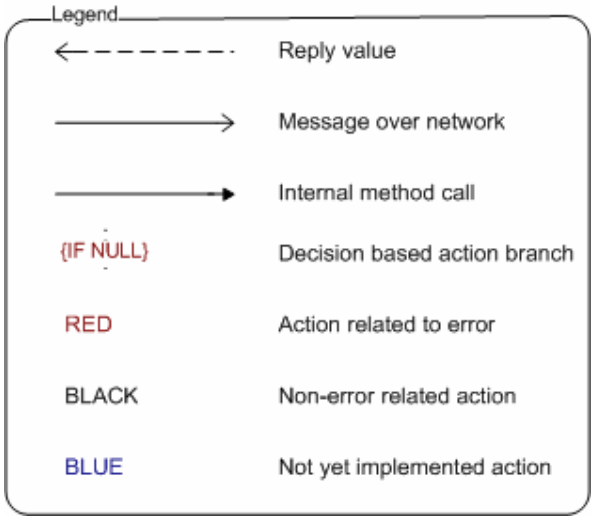

Fig. 4. Sequence diagrams legend

The sequence diagrams does not use pure UML notations [25] so the legend to the right would be very useful to understand the diagram. The decision based action branch was needed to be able to indicate when and where errors can occur. All actions related to errors are subsequently colored red. It is also possible to read from the sequence diagrams which communication is internal and which happens over the network. It is also possible to deduce this information from the positioning of the several methods, but being able to indicate this explicitly eases reading of the diagrams.

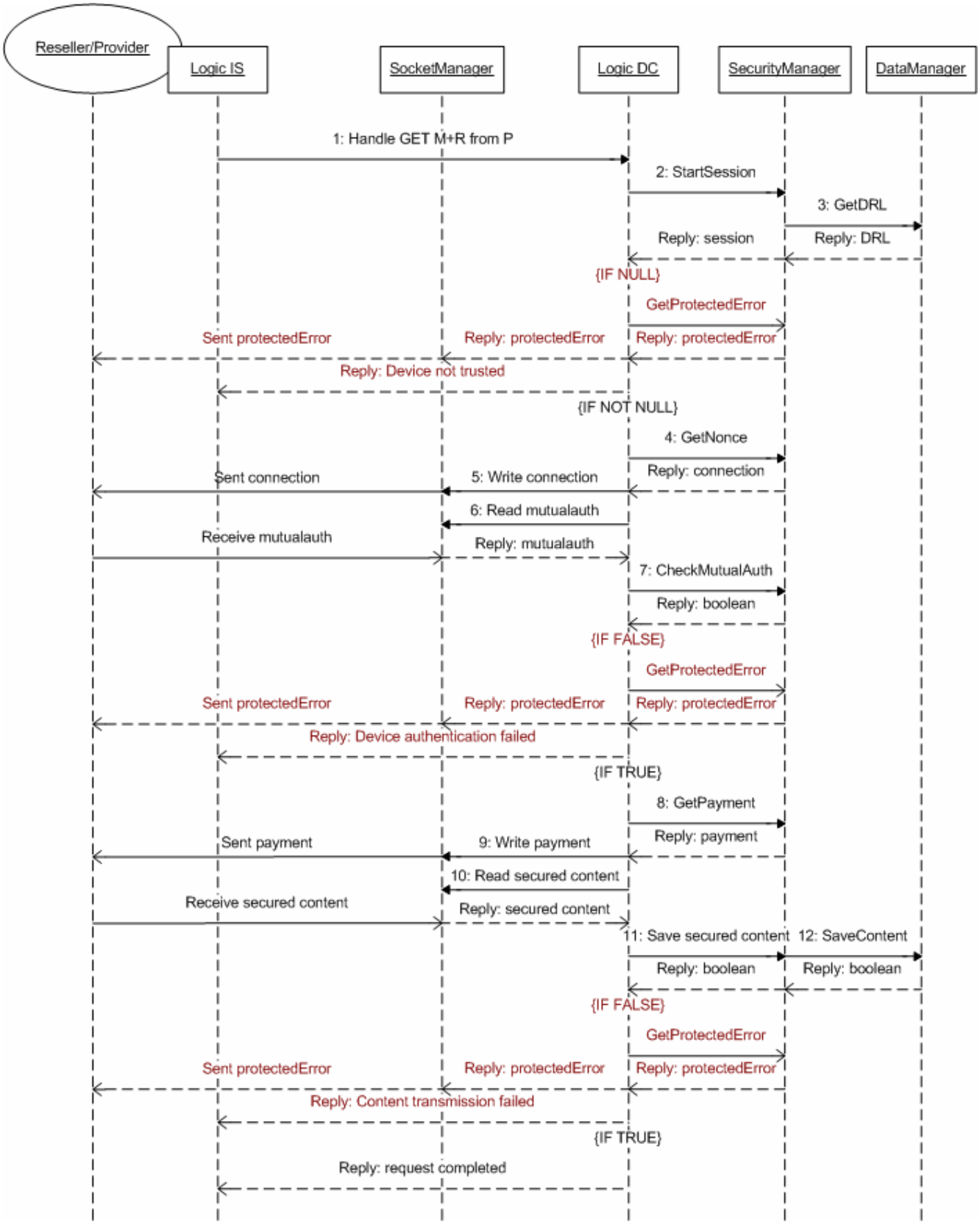

Fig. 5. P2C and C2C protocol from a customer point of view

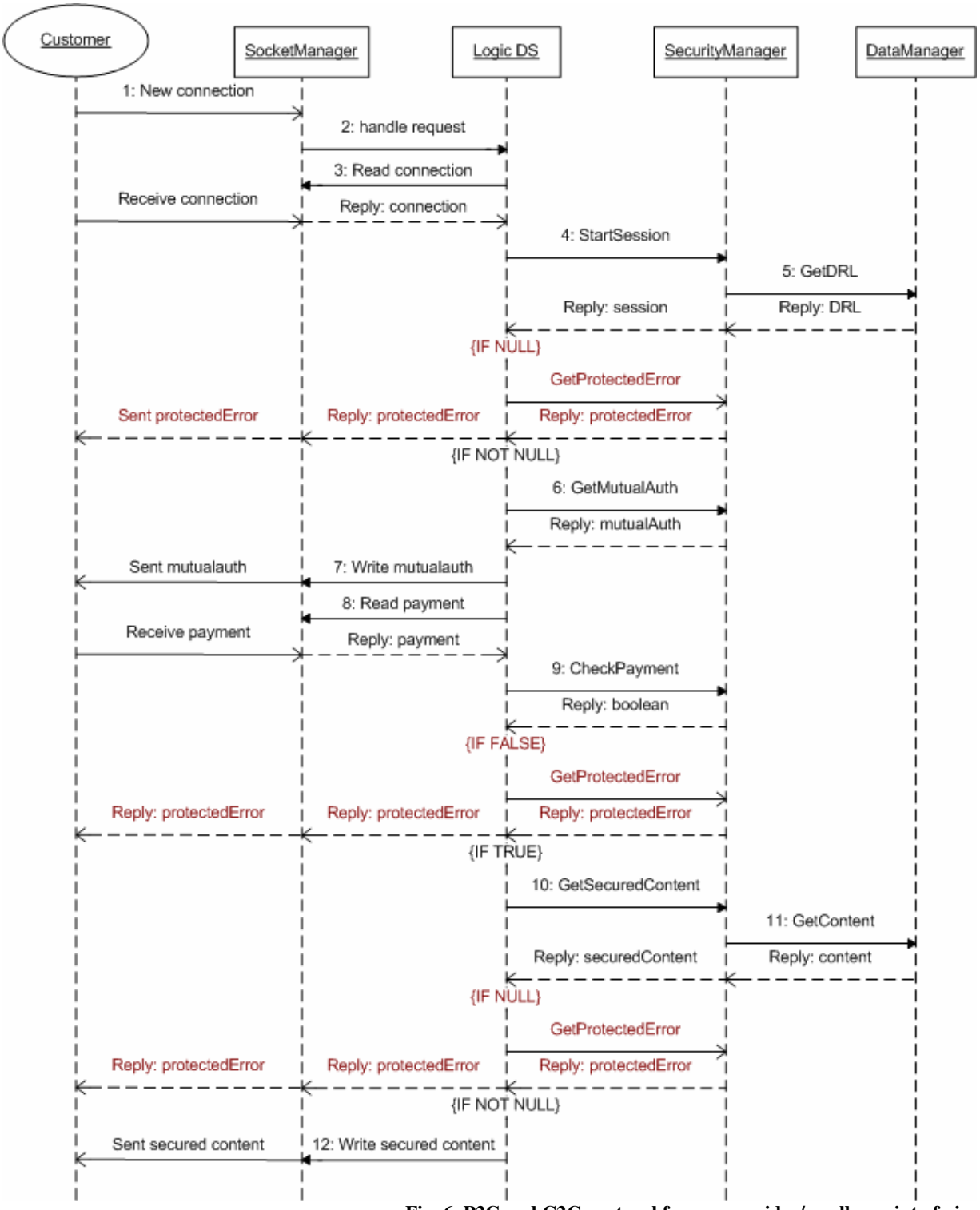

Fig. 6. P2C and C2C protocol from a provider/reseller point of view

Note the usage of error messages in both figures. In figure 5, simple, unprotected, replies can be returned to the interface server which initiated the get request. The communication between the data client and server should however be protected. These special error messages are missing in the original description of the protocol and have been specified in this project. This should not be seen as an extension to the original protocol. The real problem was that the

original protocol didn't state anything about error handling. So it was in fact underspecified and the protected error message specified in this project fills this gap. The protected error message is nothing more than the 2 nonces, an integer error code the device public key chain and a signature of all this information. In terms of the protocol: $\{nR, nC, E, C\}_{SK(C)}$

4.2 Implementation of P2C'

The same kind of sequence diagrams are used for representing the restore protocol.

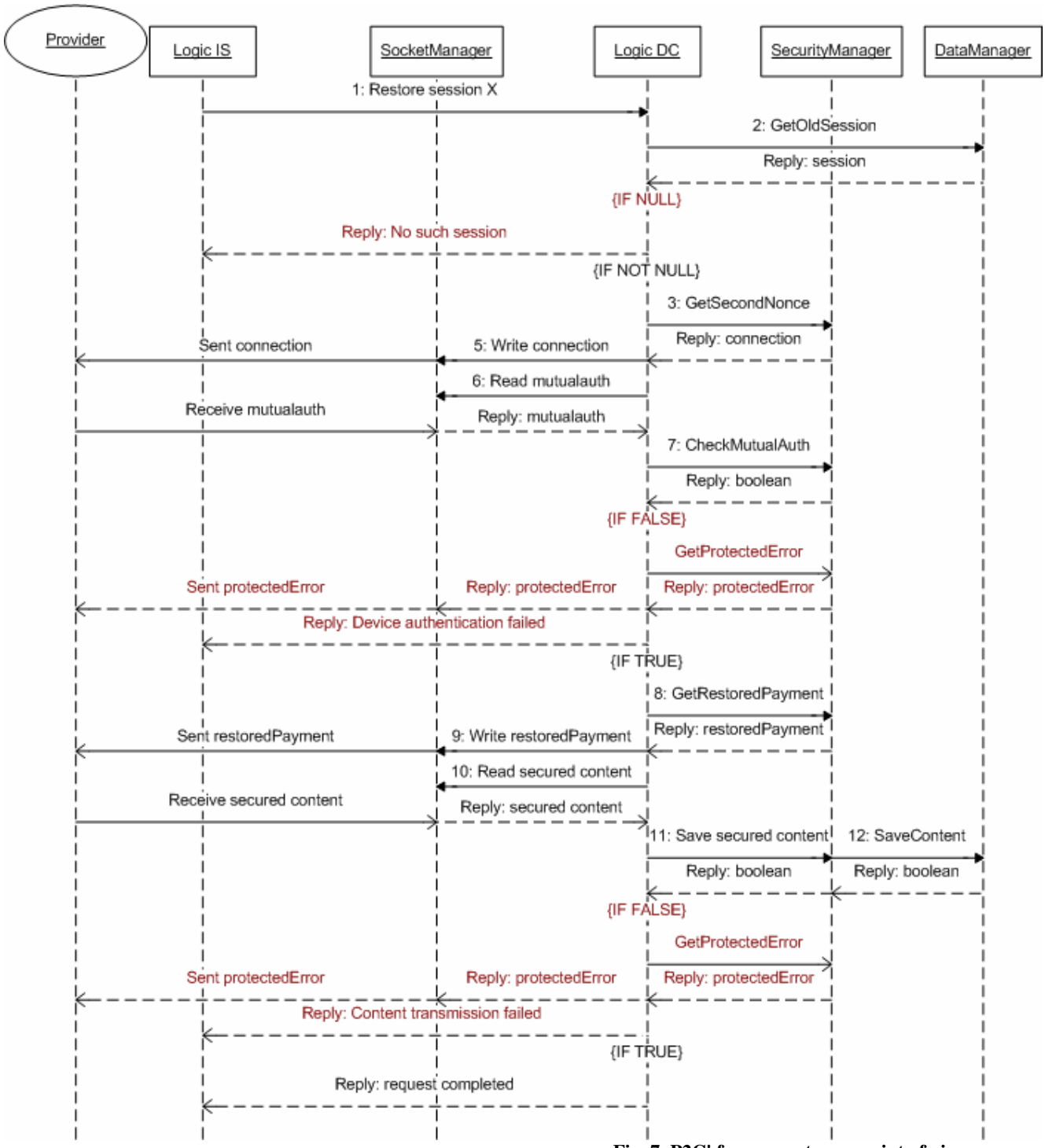

Fig. 7. P2C' from a customer point of view

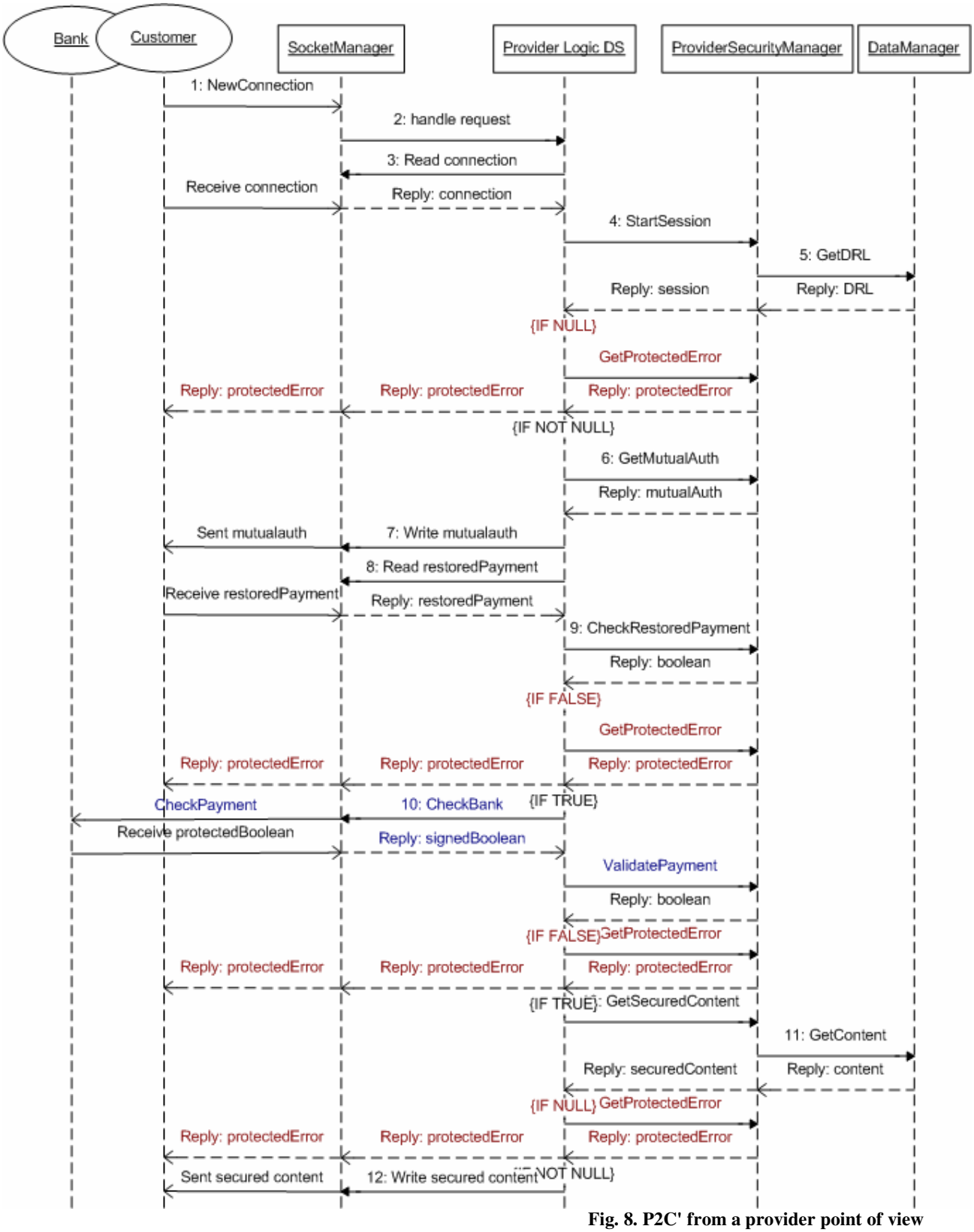

Fig. 8. P2C' from a provider point of view

Some of the methods in the provider view would only be available at the provider version of Paradiso DRM. Most of the methods however are shared by normal devices and providers. If looked closely at, it is apparent from the schemes that the provider is only an expansion of a normal customer device. Methods like startSession and getMutualAuth are invoked by both provider and customer devices while methods like checkRestoredPayment and validatePayment are only invoked at the provider side. We concluded already that the P2C and C2C protocols are basically the same except for the re-encoding of the content. The only thing that should differ is the method called by the security manager during the call of getSecuredContent. A request for content at a provider should result in a re-encoded version of the content while request for content at a reseller should not result in a re-encoded version.
In the prototype these differences in C code are created with a compiler variable called _PROVIDER_. It can be defined during compilation and triggers all these differences in C code with the use of conditional compiler directives. Most of these compiler directives can be found in the logic layer, at the location where new requests are processed. For example:

```
#ifdef _PROVIDER_
            return ds_handle_restore(request_socket);
#else
            return ds_generate_error(NULL, request_socket,
&E_DS_NO_PROVIDER);
#endif /* _PROVIDER_ */
```

The provider-only code is stored in separate C files. For example, the method ds_handle_restore is stored in the C file data_server_provider.c. During compilation this C-file will only be included if the _PROVIDER_ variable is set.

4.3 Implementation of protocol extensions

The proposed protocols only consist of a get request at a provider/reseller and a restore request at a provider. There is however no specification for listing content at a provider or reseller, manipulating the revocation list, to cash your money or scan the network for other devices. The reason for this is probably that there is no need for security in these protocols. It doesn't matter if someone eavesdrops on the content list, the payment messages or the network scan. The content list consists of per definition freely accountable information. Everyone who has a compliant device can list the content. The content list is no use for someone who has a non-compliant device. The integrity of the payment message is protected with a signature from the sender. There is no way for the swindler to manipulate the payment message to make a profit. In the scan protocol there is no need for protection either.
Finally the revocation list also has to be handled. There is support for revocation lists in this prototype, but its very limited. The revocation list used in this prototype consists of one linear list of revoked devices signed by one provider. The provider can issue new revocation lists whenever it wants. There is no version management so the provider can easily remove and add devices from or to the list without problem. Any device (even providers!) can request the revocation list stored at any other device. If the signature is accepted the current stored revocation list is replaced with the requested one. So a newer version can easily be overwritten with an older revocation list. Evidently there are several flaws in this design. The goal was however to create basic rudimentary support for revocation lists, and not to add sophisticated revocation list support.
One flaw in this design is that each single provider can release its own revocation list. A trusted device however can only hold one single revocation list. So there is no way to manage those multiple revocation lists. Another flaw in this design is that it is very hard to manage a large revocation list. There is nothing available to efficiently search for a device listed in the

list except for linear search. And finally if the list grows large it becomes even more unmanageable. The list will use a significant amount of disk space and searching it will take forever. So this prototype implementation of the revocation list is just there to show that its possible. Further research needs to be done in this area.

Take a look at the sequence diagrams. The protocols are simple enough that the customer point of view is enough.

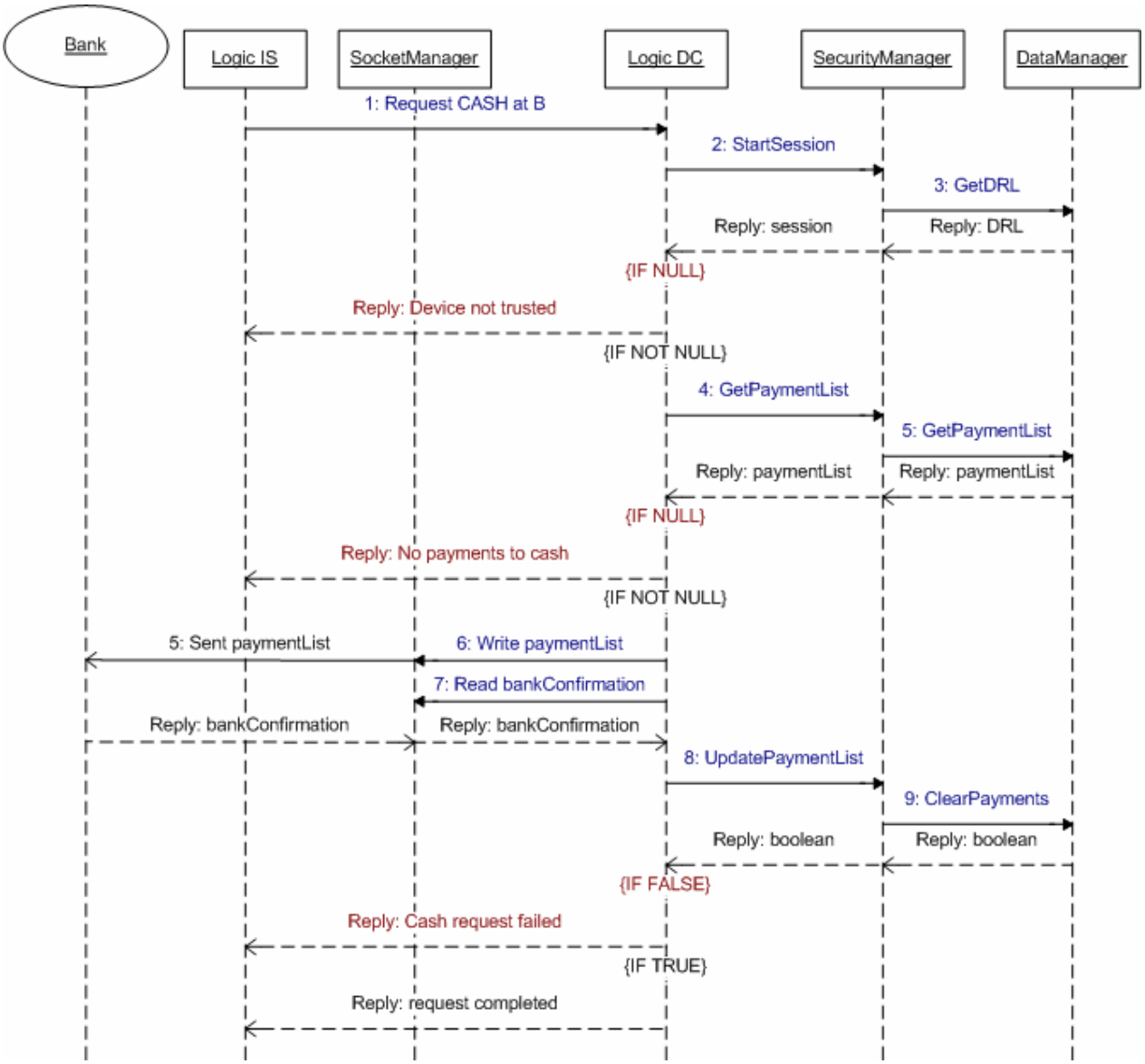

Fig. 9. Cash protocol from a customer point of view

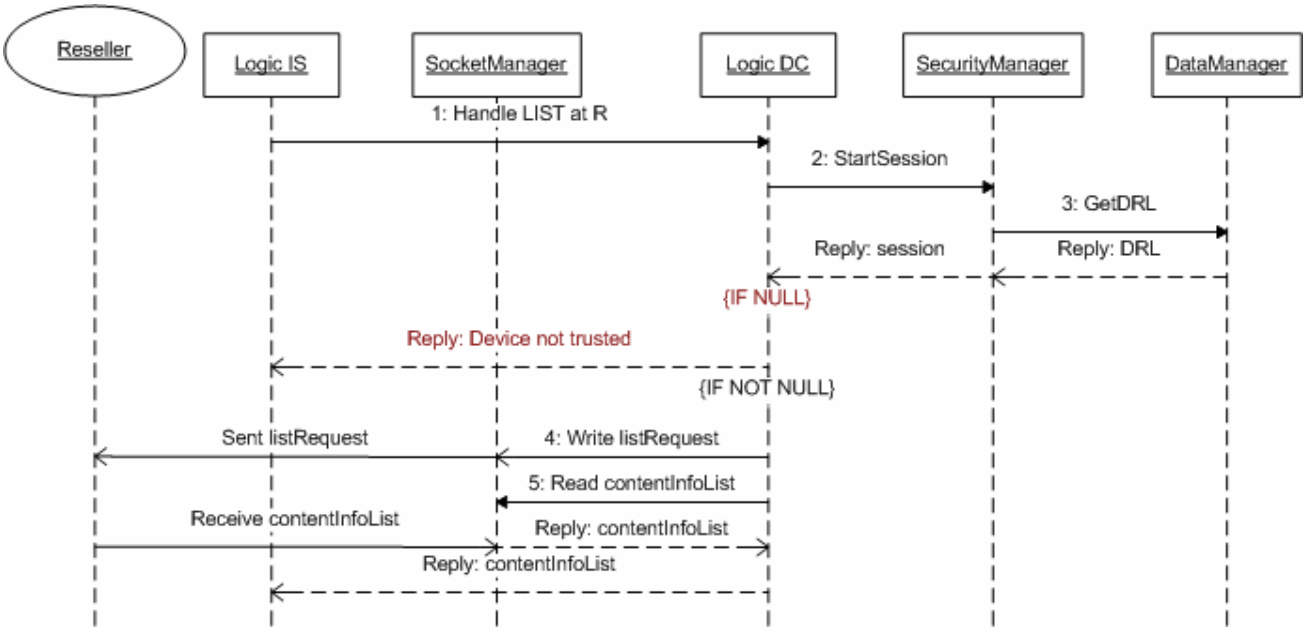

Fig. 10. List content protocol from a customer point of view

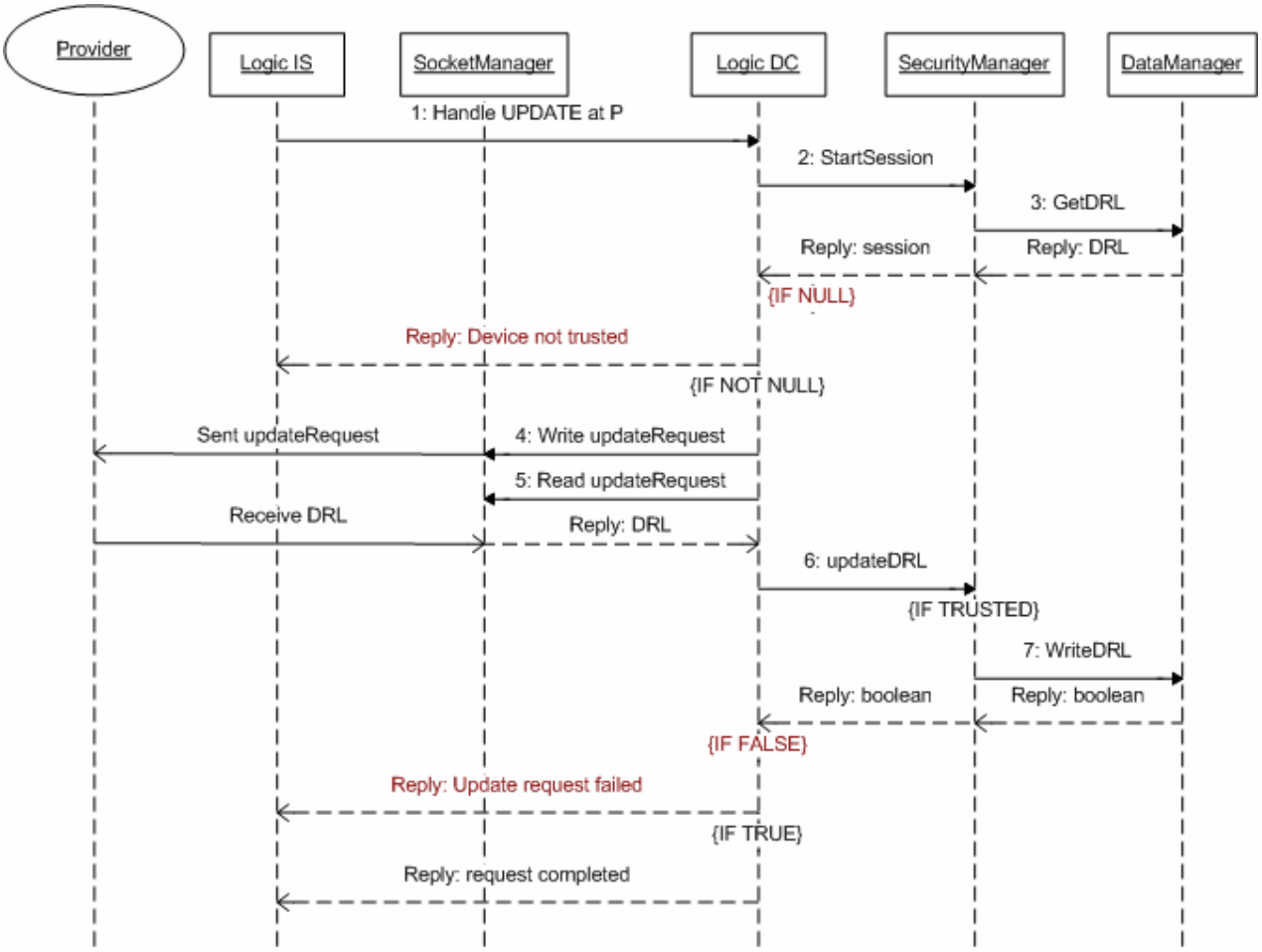

Fig. 11. Update DRL from a customer point of view

4.4 Implementation of interface client/server protocol

There is no need to represent any sequence diagrams for the interface client/server communication because a description is sufficient to understand it. Most of the commands first send an integer over the socket to indicate the command type. Then it sends the data required for the command. The data is always organized in structs. The interface server replies with an integer indicating whether the command was accepted or not. Some commands, like listing content at a device, result in information being sent to the interface client. The information is needed so the user can make a choice on what to do next. This information is sent by the interface server just after the integer. If the integer indicates an error then the interface client should not expect any data.

Chapter 5

Future improvements and concluding remarks

This report concludes with suggestions on future work to improve the prototype as well as the protocol. Creating a prototype of the Paradiso DRM protocol design gave more insight in the problems involved. Studying this prototype makes it easier to come up with new improvements to the current design. How feasible is the design of Paradiso DRM? And how can we enhance the feasibility? This chapter is divided in two sections, the first section addresses all future improvements to the Paradiso DRM prototype and the second section addresses possible improvements to the protocol design.

5.1 Improvements to the Paradiso DRM prototype

5.1.1 Connection manager retrying

As of now the connection manager isn't capable of performing any retries.. It should however be very convenient if the connection manager could do this. If for example, a message is received from which the signature is rejected, the whole connection is rejected. This creates the opportunity for an attacker to deliberately send malformed message to the data server for connections to be rejected. This could be a real threat for providers accessible in the public domain. It would be better if the connection manager would be able to just ignore this malformed message and continue listening for the next trusted message.

5.1.2 Protected error messages

The retry limitation also has some consequences for the protected error messages. They aren't implemented completely as it should be. When a protected error message arrives, the signature should be checked at the security manager and, depending upon the result, the message should be accepted or another try for the original request should be attempted. Because there is no retry support there is no need to check the protected error message. So the prototype always accepts any incoming error message and aborts the associated connection.

5.1.3 Organize dependencies and rewrite makefile

The dependencies of the makefile to build the prototype aren't organized that well. Some time could be spent to examine all dependencies and rewrite the makefile.

5.1.4 No TPM emulation

The Neuros does not have TPM compliant hardware. It would however be very interesting to reprogram the TPM part of the prototype to use a real TPM.
During the research another interesting platform was encountered. It offers a general purpose application development environment for embedded devices along with options for embedded Linux [20], Windows XP embedded and windows CE, has connectors for VGA CRT or Flat panels, has two 10/100 Ethernet ports, a PS/2 keyboard port, also an USB 2.0 Host port (fully supported), an IDE connector, and above all an Atmel AT97SC3201 TPM [11]. The vendor sells complete development kits to setup your own development environment. And there is also a tested, working and running flawlessly wireless dongle driver available [26].

5.1.5 Rights based upon XACML or XrML

In the papers [1,2] it is proposed to use an authorization and access policy language like XACML or XrML to represent the rights on the content. Due to the time limitation of this project a study and implementation based on these languages was not attempted.

5.1.6 Detailed error information between interface client/server

When an error occurs in any of the program layers, only the error returned by the lowest layer is returned to the interface client. This makes it very hard to determine what is going wrong if there was no access to the prototype main program to read the complete error trace. This problem could be solved by returning the complete error trace to the interface client.

5.1.7 Make use of ID3 tags for content information

It would be very convenient for the user if the interface client would make use of ID3 tags to determine the title and author/artist of a piece of content. At this moment the user has to type this information manually even if it's already stored in the ID3 tag of the mp3. There are open source solutions available like id3lib [12] which can be easily integrated in the interface client.

5.1.8 Updating content rights

In the papers [1,2] there nothing is stated about what to do if someone wants to obtain extra rights for a piece of content he already owns. In the prototype it is just not possible to request the same content for a second time, even if your rights are depleted. When the requested content is already present on the device the invocation of the P2C/C2C protocol is rejected. It depends upon the rights implementation how this could be improved. Furthermore it is important to once again guarantee atomicity of actions when the rights are being updated. Beware that new problems would arise in implementing atomicity if the requested content is alreadt present on the device. Details about this are described in section 3.3.4.

5.1.9 Improvements to the revocation list

As already noted in section 4.3 there is an obvious need to improve the revocation list. The current support is only there to demonstrate the capability of a revocation list. The implementation however is not scalable at all. Research should be done to come up with a new revocation list scheme. This new scheme should cover the automatic distribution of revoked device details and probably some way to distinguish between device domains in order to limit the size of the revocation list. At first sight a hash table looks like the perfect solution for searching the revocation list for a revoked device. The public key can be used as an index to the hash table in order to find out if the key is listed.

5.2 Improvements to the protocol design

In the papers [1,2] it is somewhat implicit assumed that tampering with the device is very hard or even impossible. Care should be taken when making such implicit assumptions. To emphasize this, please look at the history of the Playstation portable [14,15]. Although it uses a sophisticated system to protect the UMD discs from being copied and played illegally, the device has been hacked. Even a large company with lots of DRM experience failed to design a well protected product. This really stresses out that we should not only put a lot of effort in protecting the system from being tampered with, but an even amount of effort should be spent in how to protect the system from misuse, and limit the damage caused, by compromised devices.

Tamper-resistant hardware would be used in the actual implementation, but any number of hardware or software errors could make tampering with the device easy. These problems may seem less important than the protocol itself, and even though devices will eventually be revoked by the revocation list, the system design should be capable to deal with attacks so that the protocol won't break entirely, and that the impact of such situations is limited.

5.2.1 Introduction of a redistribution counter

The current protocol design enables a consumer to get content rights at least twice by paying once. In order to do this the consumer has to tamper with his device. First he would request for the content at a reseller just like he normally would do. He can then easily get a copy of the payment message by eavesdropping on the connection. In our prototype he then has to create a fake session file in the sessions directory. He should make sure that the content key stored in the session differs from the content he already obtained (using a 0 value for the content key is probably sufficient). At this moment he has tricked the device into thinking that there is a restorable session. So he can simply connect to any nearby provider and ask them for the content. This last step isn't possible in our prototype however because the prototype isn't able to properly process a request for content that is already stored on the device. For more details about this problem consult section 3.3. If the providers do not use some sort of communication channel to identify the used nonces then it would also be possible for the customer to obtain the content at more than one provider. This would however also make it very easy to detect that the customer tampered with his device.

So the consumer can only get the content more than once on his own device because his identity is embedded in the device. This small flaw however could become a larger problem if obtaining the private key becomes easy due to some hardware or software error. Collaborating consumers could then eavesdrop on the communication between the reseller and consumer and then pretend to be the other consumer and obtain a copy of the content at the provider.

Note that the problem is also there from a protocol point of view. It is not a design flaw of the prototype. In the protocol the payment is sent before the content is delivered. The reseller can never verify if the content has been delivered, and for this reason the restore protocol was proposed. So in fact, we can never prevent this situation from happening because there are two protocols to obtain content! Some things could be changed in the protocol to make the tampering harder, but the restore protocol can always be invoked once when the C2C protocol was used legitimately.

If there doesn't exist any state at the provider then it would in theory be possible for one identity to get certain content more than 2 times at the same provider. So this problem should, without doubt, be tackled.

It is arguable that if a consumer has paid for content once that he should be allowed to obtain it again. This should however be done in respect to the rights he has to the content. To solve this the recovery sub-protocol could be changed to a restore protocol in which there is a fixed number of times in which the content can be redelivered. This redelivery limit has a minimum value of 2. This is the new proposed P2C' protocol:

1. owner(C) → C : R, h(M), ®'
2. C → R : C, nC
3. R → C : $\{nR, nC, C, N\}_{SK(R)}$
4. C → R : $\{nC, nR, h(M), ®', R, R', N\}_{SK(C)}$
5. R → C : $\{M\}_{K}$, $\{K\}_{PK(C)}$, $\{®', nC\}_{SK(R)}$

The redelivery counter (N) is tracked by the bank. The maximum number of times some content may be redelivered is stored in the rights associated with it. The old payment message is no longer needed to invoke the restore protocol. The only difference between the C2C and the restore protocol now is that the provider sends the number of times the content is obtained already in the third step. If this number exceeds the maximum number of redelivery then the consumer must pay again for the content. This is also the reason why the provider sends the counter in the third step. It enables the consumer to decide if he wants to sent his payment when he already reached the number of maximum redeliveries.

Keeping track of this number is easy. A schematic overview of it in respect to the payment procedure is shown below. Just as proposed in the recovery sub-protocol both the reseller and provider can cash the payment message at the bank. The bank will then set and keep track of the redistribution counter.

Normal	Reseller encashes first	SET N=1
Failure (1/2)	Provider encashes first	SET N=1
Failure (2/2)	Reseller encashes second → provider pays	SET N+1
Restore	Consumer restores	SET N+1

The following scenario needs special attention:

1. The consumer gets content at the reseller (N=0)
2. The consumer didn't receive the content and restores it at the provider (N=1)
3. Communication failed once more and the consumer tries to restore at the provider once more (N=2).
4. The reseller connects to the provider to encash his money (N=3).

So the counter exceeded the maximum value, but the customer wasn't warned about this properly. So the consumer would have to pay twice now. This scenario can be prevented by keeping track of the counter in the device itself. The internal counter should be incremented each time the consumer transmits a payment message. So the counter has the value one in the first step, two in the second etc. In step 3 the consumer will be warned that by proceeding there is a chance that he will be paying twice for the content.
Another change in respect to the old restore protocol is that the identity of the reseller is sent along in the 4^{th} step now (R'). Without this it becomes arguable that the new approach isn't really fair for the reseller. It enables the reseller to determine who should be rewarded for selling the content. The consumer could however misuse this by sending a fake reseller identity. Doing this could result in a collaborating reseller being rewarded for reselling the content. There is however a risk involved in doing this. If the reseller did receive the payment message then the consumer will be paying twice for the same content. It appears for the provider as if the consumer bought the content twice at two different places. So sending a fake reseller identity is only profitable if he knows for sure that the reseller did not receive his payment. He can only know this for sure when he is collaborating with the reseller and this basically boils down to a reseller who does not encash certain payment orders.
A problem with this new approach is that now there is chance for a consumer to pay for content without obtaining it. In the old protocol the consumer could invoke the restore protocol several times until the content was delivered. In the new approach the restore protocol can only be invoked once, given that the default limit is set to two. If the customer has real communication problems with his device then this is probably not enough to obtain the content. It is assumed that invoking the restore protocol once should be enough. One of these retries is at the side of the provider which could be seen as a connection with good quality. If the connection problems still exist then the consumer can be asked to deliver some physical evidence of the miscommunication. By, for example, sending his device to the trusted manufacturer for a check-up.

5.2.2 Completely removing the restore protocol

The proposed restore protocol in section 5.2.1 is capable of much more if looked at carefully. Solely using the newly proposed restore protocol is introduced as a possible option. If the restore protocol is the only protocol available to get content at another device some of the

problems mentioned in 5.2.1 are solved and new capabilities are introduced. It becomes, for example, possible to set the maximum redistribution value to one and restoring becomes possible at both a provider and reseller.
Determining the redistribution counter in the 3rd step is the first problem encountered. The solution for this problem is simple. Each payment message a device releases will, if it isn't lost during transmission, eventually end up at the bank. The bank will then decide, based upon the maximum redistribution of the content, if a payment message will be credited or not. So there is no need of a global awareness of the redistribution counter. The usage of the redistribution counter changes. It can be used now as an indicator of how many payment messages, from the customer, are stored at the reseller. Take a look at the following situation:

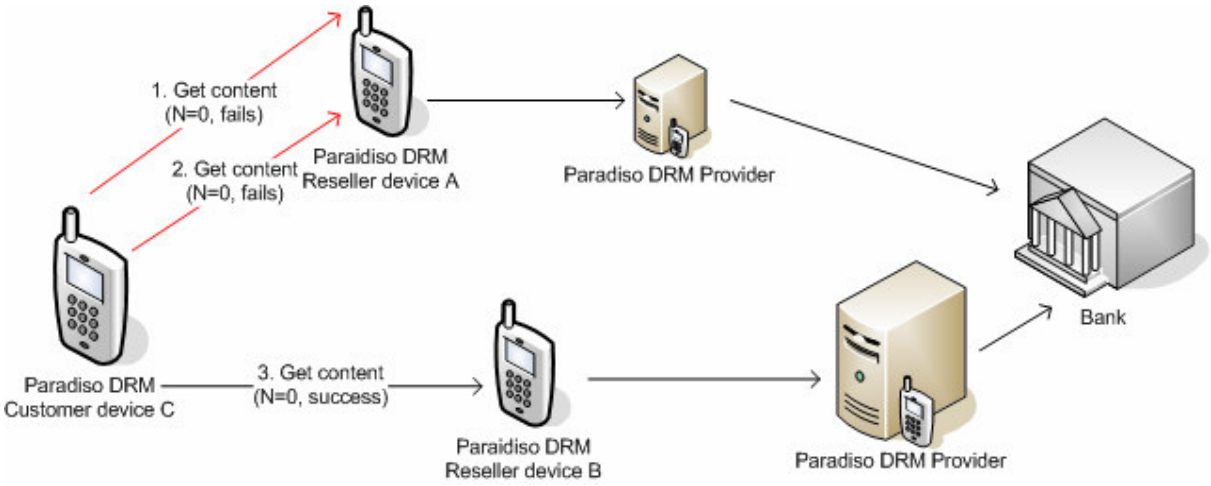

Fig. 12. Demonstration of redistribution counter and only one restore/get protocol

In step 1 the customer tries to obtain content at a reseller, but the communication fails. During the first step the reseller device states that it does not have a redistribution counter (N=0). The device responds with a payment message (4th step) which also indicates that it is the first time its obtaining this content. The customer does this by setting the R' field to equal R. So the customer device states that it wants content h(M) from reseller R which he already once obtained from R. As already described during the introduction of the redistribution counter in section 5.2.1 the R' field is used to determine who should eventually be rewarded for selling this piece of content.
If the communication failed in the original protocol then the customer would have to contact a provider. In the new situation the device can just contact the same reseller again. It then depends upon the maximum number of redistributions what the customer will be paying for the content. If it is set to 1 he should realize that there is a possibility that he is going to pay twice for the content. But assume, for this case, the redistribution counter is set to 2. Look at step 2 in figure 16. The device requests the content at the same reseller once again who then states N=0. This indicates that the reseller did not receive the payment message for this content. In step 2 the customer requests the same content, but fails again. This time the payment message is delivered to the reseller. The customer contacts that same reseller once more which then states N=1. After receiving this message, but before sending its third payment, the customer decides to cancel the protocol. The customer contacts another nearby reseller to get the content in step 3 of figure 12. In this case the customer sets the R'=A and R=B to indicate the revenue should go to A. This time the communication succeeds. Eventually when both payment messages reach the bank only one of them is credited from the customers account.

In order to use the protocol introduced in 5.2.1 as a get and restore protocol one change is needed. The reseller should be aware of the content the customer is requesting before the 3rd step in order to determine the redelivery counter. The hash and rights of the desired content should be sent along in the 2nd step to solve this problem. The new protocol then becomes:

1. owner(C) → C : R, h(M), ®'
2. C → R : C, nC, h(M), ®'
3. R → C : $\{nR, nC, C, N\}_{SK(R)}$
4. C → R : $\{nC, nR, h(M), ®', R, R', N\}_{SK(C)}$
5. R → C : $\{M\}_{K}$, $\{K\}_{PK(C)}$, $\{®', nC\}_{SK(R)}$

5.2.3 Alternative payment scheme

Another still existing problem is a reseller who never cashes payments. This threat is somewhat comparable with a reseller with a compromised device who sales the content more times then he is allowed. This thread is discussed in [1]. Counting of the total number of distributions is introduced as a non-perfect solution to this problem. When the provider counts more distributions then allowed by the given license then we know that the reseller has circumvented his device and it will be added to the revocation list. Central payments make it really easy to keep track of this number. The problem is harder if the reseller never cashes any payments. In this case the violation can only be detected by counting the content present on consumer devices which is not preferable because of privacy issues.

Fortunately there is another solution. An alternative payment scheme will (partly) solve this problem. First by introducing a consumer and reseller part in payments a reason for the consumer to contact the provider about a purchase is created. This can be seen as some sort of refund when the customer decides to 'register' his purchase. This can be accomplished without creating any privacy issues. The cost of a piece of content is incremented with a percentage embedded in the rights associated with the content. This can be seen as a 'security percentage'. When the consumer registers his purchase the security percentage is refunded. The provider credits this amount from the reseller's bank account. The reseller can only get this money back by cashing the payment he got from the consumer.

By setting the security percentage to 100% a very interesting scheme is created. The total cost of a song would then be: (cost of song x 2) + reseller commission. The consumer pays twice for his song, but the provider will refund once (the 100% security percentage) if he proves which reseller sent him the song. The refund will then be credited from the bank account of the reseller. So in this particular case, if the reseller decides not to cash his payment, he will be paying for the song himself. Now it becomes impossible for a reseller, who isn't collaborating with his customers, to offer free content to the public.

It also becomes easier to detect collaborating reseller and customer devices. If one of the customers by accident cashes his refund and we never receive, in a x amount of time, the associated reseller payment then the chances are high that the reseller is a fraud.

This payment scheme may seem like the ultimate solution. There is however a good argument against it. In a sense the usefulness of the protocol as a whole is degraded. If the consumer has to contact the provider each time content is obtained, why can't he then just buy the content at the provider? On the other hand, if connecting to the provider is very easy then there is no problem. Provider communication could be embedded in the docking station for example. When the consumer goes outdoors he can get his content from resellers, and when he is at home he puts his device in the docking station which then synchronizes with the provider.

In the new payment scheme it should be prevented at all cost that the reseller gets the payment message from the customer, but the customer doesn't get the corresponding refund message. In this case the customer will be paying twice for content he didn't receive. On the other hand it wouldn't be fair if the customer could get a refund on some content he did never

pay for. At first sight it is only possible to solve one of those problems. It all depends upon the moment that the refund message is sent:

```
1. Request data -> reseller
2. OK -> customer
3. Payment message -> reseller
4. Refund + Content message -> customer
```

Or:

```
1. Request data -> reseller
2. Refund message -> customer
3. Payment message -> reseller
4. Content message -> customer
```

In the first case there is chance that the customer will be paying twice for content he didn't receive. This situation occurs if there is a failure between message 3/4. In the second case however it becomes fairly easy for the customer to collect refund messages from resellers. There is no way for the reseller to proof that he didn't receive the corresponding payment message. But what if we enforce the customer to present a receipt of the content for the refund message to be valid. If the customer has this receipt then it is sure that he paid for the content. The protocol then looks like this:

```
1. Request data -> reseller
2. Refund message -> customer
3. Payment message -> reseller
4. Receipt + Content message -> customer
```

At first sight this may introduce the same problem as by the first solution. There is a chance that the customer will be paying twice for content he didn't receive because he now has to proof the he has a receipt of the content and in order for the customer to have this message step four should be completed without problems. The problem is solved by presenting two options for the customer at the bank. The first one is that the customer presents a refund message including the receipt. In this case the amount will be immediately transferred from the reseller account to the customer. In the other case the customer can present a refund message solely. Now the refund message is only valid if the reseller also presents his payment order. For this alternative scheme to work only one small change of the protocol proposed in section 5.2.2. In the third step the rights of the content should be added in order to create a valid refund message. A part of the last step ($\{®', nC\}_{SK(R)}$) could be used as receipt.

5.3 Concluding remarks

The first results of the prototype are very promising. Paradiso DRM has the potential to become a mayor player in a new still largely unrevealed market segment of DRM-preserving digital content redistribution. The prototype demonstrates its feasibility and helps interested parties to get a good feel of the problems involved. There is however enough work left to do.

References

[1] Srijith Krishnan Nair and Bogdan C. Popescu and Chanadana Gamage and Bruno Crispo and Andrew S. Tanenbaum; Enabling DRM-preserving Digital Content Redistribution, Vrije Universiteit, Faculty of Sciences, 2005

[2] H. Jonker, S. Krishnan Nair, M. Torabi Dashti; Nuovo DRM Paradiso: Formal specification and verification of a DRM protocol, Technische Universiteit Eindhoven, Vrije Universiteit Amsterdam, CWI Amsterdam, 2006

[3] Rockbox is an open source replacement firmware for mp3 players, www.rockbox.org

[4] iPodLinux is an open source venture into porting Linux onto the iPod, www.ipodlinux.org

[5] Any IP based service can be linked together using FireWire with this driver, http://developer.apple.com/hardwaredrivers/firewire/ip_over_firewire.html

[6] A stable and feature rich Linux driver for wireless 802.11b and 802.11g cards that are based on the Ralink rt2400 and rt2500 chipsets, http://rt2x00.serialmonkey.com/

[7] A driver supporting all devices based on the RT2500USB chipset, http://etudiants.insia.org/~jbobbio/ural-linux/

[8] Linux 2.6.14> patch for deprecated verify_area, http://www.colino.net/wordpress-1.5/archives/2005/10/29/fglrx-unknown-symbol-verify_area/

[9] Information and explanation of data structure alignment, http://en.wikipedia.org/wiki/Data_structure_alignment

[10] Andrew S. Tanenbaum, Albert S. Woodhull; Operating systems, second edition, ISBN0136386776, 1997

[11] The SBC-GX533 is a low profile, fan-less, RoHS compliant EBX form factor board, http://www.arcom.com/pc104-geode-gx533.htm

[12] an open-source, cross-platform software development library for reading, writing, and manipulating ID3v1 and ID3v2 tags, http://id3lib.sourceforge.net/index.html

[13] The AT97SC3201 is a fully integrated security module designed to be integrated into personal computers and other embedded systems, http://www.atmel.com/dyn/resources/prod_documents/2015s.pdf

[14] Sony Play Station Portable news, http://pspupdates.qj.net/

[15] PSP mod chip information, http://www.psproms.com/psp_mod-chips.htm

[16] The PP5002 SuperIntegration System-On-Chip used in the ipod http://www.portalplayer.com/products/documents/5002_brief_0108_Public.pdf

[17] Information about USB On-The-Go http://www.usb.org/developers/onthego/

[18] Apple iTunes, Apple Computers Inc, http://www.apple.com/nl/itunes/

[19] Apple iPod, Apple Computers Inc, http://www.apple.com/ipod/

[20] Linux, Linus Torvalds et. al, http://www.linux.org/

[21] Postel, J.; Internet Protocol, RFC 760, USC/Information Sciences Institute, January 1980, http://www.ietf.org/rfc/rfc0791.txt

[22] Postel, J.; Transmission Control Protocol, RFC 761, USC/Information Sciences Institute, January 1980, http://www.ietf.org/rfc/rfc793.txt

[23] Postel, J.; User Datagram Protocol, RFC 768, USC/Information Sciences Institute, August 1980, http://www.ietf.org/rfc/rfc0768.txt

[24] A full-strength general purpose cryptography library, http://www.openssl.org/

[25] Fowler, M., Scott, K.; UML Distilled: A Brief Guide to the Standard Object Modeling Language, second edition, ISBN020165783X

[26] OpenSource Linux Driver for Atmel AT76C5XXx-based Wireless Devices, http://atmelwlandriver.sourceforge.net/links.html

Appendix A

This appendix describes how wireless support was added to the neuros platform.

The wireless dongle

Research showed that a wireless dongle with a ralink chipset would be the best choice. It has 2 open source drivers available [6,7]. Both of these drivers are maintained actively and there are also known applications of them in ARM devices. Furthermore the Asus WL-167G uses this chipset and is low priced. All of this gives us a good base to get the wireless dongle working.

Porting the wireless dongle driver

It took about 6 weeks in total to port the wireless dongle driver. In this time both open source drivers were tested. Furthermore a lot of different versions of the kernel were compiled and tested with a version of the driver. At a certain moment a completely different wireless dongle was tested. In this chapter will however only describe what the process is, and why this should be done, to get the wireless driver for the Asus WL-167G working on the neuros platform. Following this procedure probably costs one day, but finding out this procedure cost a lot more time. As discovered during all this testing there are several requisites for the wireless driver to work:

1. Kernel support for wireless tools
2. Wireless tools for ARM architecture
3. A certain version of the neuros linux kernel
4. ARM compatible C code of driver

Wireless extension and tools

The wireless extension is a generic API allowing to configure and see statistics of the wireless driver in user space. A single set of tools supports whole ranges of WLANs regardless of their type. The only thing needed is wireless extensions added to the kernel and a set of tools, called wireless tools, to manipulate the driver configuration, request driver information and sent commands. All configuration parameters can be changed on the fly which eases wireless configuration.
Almost every linux WLAN driver available today supports wireless tools. There are only a few drivers, most of them are very old, who do not support wireless tools and come with their custom made set of tools. The driver
The default kernel that comes with the neuros doesn't have wireless extensions compiled in the kernel. So a custom kernel has to be made which includes the wireless extensions. After this the new custom kernel should be burned to the internal flash memory of the neuros. When this is accomplished the newest version of wireless tools should be downloaded. Fortunately the wireless tools are open source and the C code is ARM compatible. So the only thing to do at that point is compiling the wireless tools with the arm compiler supplied with the neuros. For this the Makefile of the wireless tools has to be changed on 2 points. At first it should include the makefile configuration directives matching the development environment. And second it should include the correct linux kernel files. Then the wireless tools will compile without problems.

The right kernel

At first the USB Host driver was running without problems. During the order of 2 neuros boards and the delivery of them however they migrated the board to the newest linux kernel, 2.6.5 -> 2.6.15. With this migration the USB Host support broke. This wasn't documented

very well so we first tried everything on the newest linux kernel until someone at neuros told that the USB Host driver was broken. During this testing with the newest linux kernel the USB driver appeared to be working quite well. It was reported that the hotplug support didn't work that well (so we had to replug-in the wireless dongle several times for the driver to detect it), but after that everything seemed to be working quite well. It wasn't apparent that the Host driver was broken. This forced us eventually to downgrade the board. The old version of the kernel however is very buggy, there isn't a list available with things that don't work instead they give a list with things that do work. Small parts of the kernel code had to be changed to get it compiling properly. After this the wireless dongle didn't work either.

Porting the wireless dongle driver

The driver that eventually worked was the one made available by the vendor. This is basically version 2.0.7.0 of the serialmonkey RT2570 Ralink driver. Asus doesn't present any notes about changes to the serialmonkey driver. The newest driver available at serialmonkey doesn't work in combination with the older 2.6.5 linux kernel. This is mainly caused by a method called verify_area which was removed somewhere between version 2.6.5 and 2.6.15 from the linux kernel. There is a module available [8] to fix this undefined symbol problem. But even with this fix the driver doesn't seem to be working very well. The driver is able to detect the wireless dongle and you can configure it without any problems using the wireless tools. However when you try to scan for networks the driver fails to detect any wireless network. So we changed a small part of this scanning procedure and discovered that it was able to see other wireless networks. They got rejected however due to a failed sanity check. There just wasn't any apparent reason why the check failed. So we disabled this sanity check to see what happened. The driver was able to connect to the network now! However no data transfers were possible and connection got lost after several seconds.
At a certain moment we gave upon the wireless network driver. There just wasn't anything left to test, and there weren't any leads on what could go wrong. At this point something important was discovered while programming at the prototype. The prototype of Paraidso DRM could without problems communicate between two x86 computers and between two neuros devices. When however a communication between a x86 computer and a neuros platform was tried weird things started to happen. After a lot of debugging is was discovered that the size of structs differed between the ARM and x86 compilation of the prototype. After some research we figured out that this was caused by data structure alignment. The x86 compiler has no alignment (or 1-byte alignment), but the ARM compiler has a default 4-byte alignment. There are many reasons available why the compiler has an default alignment of 4 bytes [9]:

- Extra transistors on the CPU are required to support accesses which are not word-aligned
- Reads not aligned to the width of the memory bus require two reads.
- Writes not aligned to the width of the memory bus require two reads and two writes.
- Accesses across cache-lines require evicting two cache-lines.
- Accesses across page boundaries can incur two TLB misses and could even require swapping in both pages from disk

The ARM architecture is based on the RISC philosophy. It is general for RISC processors to set a word boundary to increase performance. So all data should be stored or loaded to or from a word boundary. When however an address is accessed which is not on a word boundary the compiler can set a flag so a subroutine is called to figure out where write or load

the data exactly. The ARM processor probably uses 4 byte words so a default alignment of 4 bytes increases performance. A small example to clarify what is happening:

```
struct example{
 uint8_t type;
 uint8_t data;
};
```

Normally this struct would use 2 bytes of space. But it uses 4 bytes of space when compiled with the ARM compiler. It is quite easy to set the compiler alignment by using the #pragma directive. The GNU C compiler however only recognizes it, but does not support it. The pragma directive is mainly used for windows programming where alignment setting is more commonly seen. There is another directive available called packed structs. It works as follows:

```
struct example{
 uint8_t type;
 uint8_t data;
} __attribute__((__packed__));
```

Now when the struct is compiled using the neuros arm compiler it uses 2 bytes also. This immediately fixes the communication problems between ARM compiled and x86 compiled code. At this point we figured out that this was probably what all the weird problems with the wireless driver was causing. Something like this would certainly cause sanity checks to fail. Bearing this in mind we looked at the wireless driver again. The driver makes use of a lot of structs for its data management. The most remarkable was that it also made use of structs for hardware I/O. Something that is completely not-done for driver programming. So it was certainly worth a try to rewrite all those structs. The packed attribute can not be used in combination with typedef so a lot of rewriting was needed. So the moment of truth had come. The ported code was compiled and tested. It worked flawlessly now! Even WEP encryption was working without any problems.