ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS
Department of Informatics
Master of Science Program in Computer Science

Diploma Thesis

The Peer-to-Peer Wireless Network Confederation Protocol: Design Specification and Performance Analysis

Pantelis A. Frangoudis

Advisor : George C. Polyzos

Athens, June 2005

Abstract

The Peer-to-Peer Wireless Network Confederation (P2PWNC) is a peer-to-peer system that enables WLAN hotspot roaming based on service reciprocity, motivated by the wide spread of low cost wireless equipment and broadband access technologies. System users are organized into small teams (peers) and members of one team can freely access hotspots of other teams, provided that their team also offers service. The design specification, reference implementation and performance analysis of a protocol for the operation of the P2PWNC scheme is presented. Two P2PWNC modes are studied; a centralized mode, requiring a Trusted Central Authority responsible for keeping the history of service provisions in the system and a decentralized mode, in which team-local modules have replaced the central authority. Even in the centralized case, though, peers make autonomous decisions and, thus, the peer-to-peer nature of the system is not compromised.

Acknowledgements

I wish to thank Professor George Polyzos, advisor of my diploma thesis project, for his guidance during the development of the project. Also, I would like to thank Professor Costas Courcoubetis for accepting the role of the external advisor. Last but not least, I wish to thank Elias Efstathiou for his invaluable help in every step of the process of designing, specifying, implementing and evaluating the P2PWNC protocol.

Contents

1 Introduction **1**

2 The Peer-to-Peer Wireless Network Confederation **4**

2.1 Background 4
2.2 Overview 5
2.3 Entities 6
2.3.1 Teams 6
2.3.2 Team Members 7
2.3.3 Access Points 8
2.3.4 Receipts 9
2.3.5 Receipt Repository 9
2.3.6 Trusted Central Authority 10
2.3.7 Team Server 10
2.4 Architecture 11
2.4.1 Centralized Design 11
2.4.2 Decentralized Design 12

3 Protocol Specification **13**

3.1 P2PWNC Protocol Messages 13
3.1.1 General 13
3.1.2 Core Protocol Messages 14
3.1.3 Augmented BNF for the P2PWNC Protocol 16
3.2 Cryptography 22
3.2.1 Supported Cryptosystems 22
3.2.2 Key Generation 23
3.2.3 RSA Cryptosystem Parameters 23

3.2.4 Elliptic Curve Cryptosystem Parameters 23
3.3 Data Representation 24
3.3.1 Public and Private Keys 24
3.3.2 Digital Signatures 25
3.3.3 Digital Certificates 25
3.3.4 Receipts 26
3.4 Entities and their Behavior 27
3.4.1 System Parameters 27
3.4.2 Centralized Case 28
3.4.3 Decentralized Case 31
3.5 Message Sequences 35

4 Reference Implementation 38
4.1 Generic Modules 38
4.1.1 Protocol Module 38
4.1.2 Cryptographic Operations 43
4.1.3 Multithreading 44
4.2 Receipt Repository 46
4.2.1 The Role of the Module 46
4.2.2 Supported Operations 47
4.2.3 Module Architecture 48
4.2.4 Decision Algorithms 61
4.3 Access Point Agent 64
4.3.1 Architecture 64
4.3.2 Client Session Handling 65
4.3.3 Communication with the Receipt Repository 68
4.3.4 Traffic Measurements 68
4.3.5 Network Access Control 70
4.4 Mobile User Agent 72
4.4.1 Operation in the Centralized Case 72
4.4.2 Operation in the Decentralized Case 73
4.5 Trusted Central Authority 74
4.6 Team Server 75

5 Performance Evaluation **77**
5.1 System Testbed 77
5.2 Performance Metrics 78
5.2.1 Time Measurements 78
5.2.2 Space Measurements 80
5.3 Cryptographic Operations 81
5.3.1 Parameters 81
5.3.2 Measurements 82
5.4 Maximum Flow Algorithm Performance 85
5.4.1 Running Time Evaluation 86
5.4.2 Memory Utilization Evaluation 89

6 Future Work and Extensions **92**
6.1 Security Considerations 92
6.2 Implementation Issues 93
6.3 Deployment Issues 94
6.4 Evaluation Issues 95

7 Conclusion **96**

References **97**

List of Figures

2.1 P2PWNC centralized mode 11
2.2 P2PWNC decentralized mode 12

3.1 Client operation in centralized mode 28
3.2 Access point client session in centralized mode 30
3.3 Client operation in the decentralized operation mode 32
3.4 Client operation in the decentralized operation mode 33
3.5 Team server operation 34
3.6 Message sequence in the centralized operation mode 36
3.7 Message sequence in the decentralized operation mode 37
3.8 Receipt repository update operation 37

4.1 Message conversions 41
4.2 Compact representation of a receipt 42
4.3 Receipt repository architecture 49
4.4 Adjacency list representation of receipt graph 58
4.5 Receipt graph segment 63
4.6 Access point agent modules 65
4.7 TCA software architecture 75

5.1 Maximum flow running time on an AMD AthlonXP 2800 87
5.2 Maximum flow running time on Linksys WRT54GS 88
5.3 Maxflow memory usage on Linksys WRT54GS 91

List of Tables

3.1 Valid ECC curves in the P2PWNC protocol 24

5.1 Platform specifications 78
5.2 Cryptographic operation performance 83
5.3 Key size ratio 83
5.4 Cryptographic operations running time ratios 84

1 Introduction

Nowadays, wide deployment of Wireless LAN (WLAN) technologies is taking place. The cost of equipment needed to connect users in a WLAN has degraded rapidly and the effort necessary for setting up and configuring such a system is minimal. Also, WLAN technologies operate in unlicenced radio frequencies. Thus, the WLAN market share that residential users occupy has increased.

Although wireless coverage is increasing, there is no unified way of accessing these networks. The development of WLAN market has not followed the traditional marketing models of other telecommunication technologies, such as cellular telephony. Instead, it has grown in an unplanned manner and this is the reason why there is no established WLAN access model.

What is more, the spread of WLAN technology was not initiated by large telecom organizations. Rather, due to the low cost of equipment, individuals, universities and the open source community quickly adopted it and formed a large, uncontrolled by large operators, user base.

Together with the proliferation of broadband access technologies, such as xDSL, the spread of WLAN hotspots forms the ideal breeding ground for the provision of ubiquitous internet access. However, network heterogeneity and limitations imposed by the wireless network operators hinder the development of such services.

In order to provide ubiquitous internet access, a scheme providing seamless wireless LAN roaming is necessary. Despite the fact that the technology for sharing wireless internet is available, there is no established model providing such a service. Numerous solutions have been proposed addressing this issue. However, none of them has had global acceptance for various reasons. Perhaps, the most important of them was the lack of established WLAN roaming standards as well as the fact that the user base of WLAN technology has evolved in an unpredictable manner. The result is a fragmented market containing commercial hotspot operators, free citywide networks [1] [9], non-commercial operators such

as university campuses and individual household access points.

From the above discussion, it occurs that universal WLAN roaming capabilities are extremely limited, considering the advances in the area of wireless networking. Obviously, in none of the existing solutions is the potential of sharing household broadband connections through residential WLAN hotspots exploited.

Aiming at fueling the deployment of ubiquitous internet access, a novel approach was proposed in [18] [19]. The scheme that is to be described in this document is based on the peer-to-peer paradigm. The principles upon which its design is based are those of agent autonomy, and implementation and deployment simplicity. The players in this scheme join small teams, each of whom operates a number of public access points. The members of the teams are mobile users who roam within the coverage area of access points that belong to other teams. Each of these teams is a peer in our system. Roamers are granted internet access according to the following simple high-level rule:

Members of a team are granted access if it can be proved that their team also serves members of other teams

Thus, the system is built upon service reciprocity. These groups of WLAN users and operators build an exchange-based economy, in which the exchanged good is WLAN access.

The *Peer-to-Peer Wireless Network Confederation (P2PWNC)*, as the proposed scheme is called, does not impose on its users the overhead associated with the financial, legal and technical complexity that traditional roaming aggreements incur. Instead, management complexity is reduced by not using strict service accounting, offering agents autonomy of decisions and grouping users in teams. The P2PWNC attempts to achieve the goal of offering agents the right incentives that promote cooperation between them, considering that agents are autonomous, rational and self-interested. By cooperation, adherence to the reciprocity rule is implied. It is also implied that adherence to the rules of the system is not enforced by any means other than by offering the right incentives. For example, teams are encouraged to offer service with the promise of freely being served by their peers in the future. All system participants, though, take independent decisions. What is more, none of the system's software components is considered tamper-proof; a user might as well deviate from the specified behavior, if it is for her own interest. A peer may refuse to offer service without being directly punished. However, the threat of being excluded from the system and enjoy free roaming no more should prevent him from diverging from the system's rules.

Two possible deployment scenarios of the system have been studied and developed. First, there is a centralized design, where a Trusted Central Authority maintains the history of service provision in the system. Second, there is a decentralized one, where there is no globally trusted central agent; rather, the TCA is replaced by team-local entities that are only trusted by the members of the team to whom they belong.

The remaining document is organized as follows. In Section 2 the entities and the architecture of the P2PWNC scheme are presented. Section 3 deals with the specification of the protocol for the communication of the system entities and the generic agent behavior. The reference implementation of this protocol is presented in Section 4. A perfomance evaluation of the prototype system that has been imlemented is shown in Section 5. Finally, potential extensions and future work can be found in Section 6, while Section 7 concludes.

2 The Peer-to-Peer Wireless Network Confederation

2.1 Background

The widespread deployment of WLAN hotspots has caused the emergence of *wireless cities*. Household users and other hotspot operators set up wireless access points, each of them offering network coverage to mobile users in a range of few meters. Such wireless *freenets* [9] [1] aim at providing internet connectivity to their members and, sometimes, to passers by their antennas.

Other approaches, such as [5] and [10], aim at the residential wireless access point owners and offer a means of making a profit out of sharing their broadband internet connections. That is, visitors to other access points are charged for the service they enjoy. Residential WLAN and broadband connection owners may use such methods to compensate for the maintainance cost of those connections. However, such solutions suffer from management complexity issues. One the one hand, the companies that offer such services are actually involved in the payment mechanisms. That is, they mediate transactions by taking care of billing. On the other hand, there is overhead concerning economic, legal and technical issues for the service providing access points, which may be a discouraging burden, especially for household operators.

A proposal with the similar motivation as the P2PWNC is presented in [25]. The scheme involves a reputation-based mechanism for Wi-Fi roaming with the presence of a trusted central authority and employs a micropayment scheme. Another approach using an exchange-based system to promote resource sharing is [13]. In particular, a peer-to-peer file sharing system is described, which is based on indirect service exchange, in a similar manner as in the P2PWNC scheme. However, their ideas are not direclty applicable to the P2PWNC, since their assumption that both peers are providing service simultaneously may

seem restrictive for the case of WLANs.

2.2 Overview

The Peer-to-peer Wireless Network Confederation is a WLAN roaming scheme whose participants are rational, self-interested and autonomous agents. Agent behavior is driven by their tendency to maximize their profits, that is enjoy as much service as possible, having the least possible cost.

In the proposed scheme, users team up, joining small groups that will from now on be denoted as *teams*. Teams are the *peer* entities in the P2PWNC. A team has a number of wireless access points in its possession and a team's members are mobile users. That is, the role of a team is dual; seen as a whole, it acts simultaneously as a service provider (via their access points) and as a service consumer (via their roaming members). Roamers interact with foreign access points. The outcome of such a transaction is service provision on behalf of the access point and an unforgeable proof of service on behalf of the mobile user, that will be denoted as *receipt*. Cryptographic primitives ensure the validity of this proof of transaction.

History of prior transactions between teams is kept in a repository of *receipts*. As mentioned earlier, this repository may be shared by all teams (centralized design) or each team may keep its own private repository (decentralized design). The first case implies the existence of a *Trusted Central Authority (TCA)* acting as the central repository. However, even in this case, the maintenance of private team repositories is not out of the question. Decisions of service provision taken by access points are normally based on the transaction history.

Before delving into the details of the P2PWNC scheme, some basic assumptions and characteristics of the system need to be stated.

- Peers are identified by simple public/secret key pairs.

- Team members are identified by a certificate issued by the team they belong to.

- Accounting is *relaxed*, and so is agent identification information. That is, although the traffic that mobile users cause in visited hotspots is measured, service provision is not granted strictly according to the measured traffic volume; rather, a probabilistic module is used, as described in Section 4.2.4. Also, the only identification information are public keys. No user names or real-world identities need to be used.

- Each team operates a number of Access Points providing access to roamers belonging to other teams.
- Decision on whether a mobile user will be provided service are taken autonomously by the service provider. Normally, this decision will be based on transaction history information, but this is is not enforced, in accordance with the agent autonomy principle.
- Service is provided according to the following rule of reciprocity: *"Members of teams may be freely serviced by hotspots belonging to other teams if they can prove that their team also serves members of other teams."*
- All system agents are selfish and rational. Thus, it is assumed that the threat of exclusion from the system coupled with the promise of enjoying free wireless internet access can act as an incentive to cooperate and obey to the system's rule of reciprocity.
- There is no trust between peers, nor any cooperation between them is supposed. In the centralized design of the P2PWNC scheme, full trust to the TCA is assumed.
- There is total intra-team trust.

In the following section, the P2PWNC system entities, their roles, characteristics and relationships will be defined.

2.3 Entities

2.3.1 Teams

As described in the previous section, users of the P2PWNC form small *teams*. Teams are providers and consumers of service at the same time. They provide service via the hotspots they operate and their members consume service when visiting foreign access points. The necessity of grouping members in small teams instead of regarding each individual as a peer is apparent, considering the following advantages:

- Since the scheme proposed also aims at residential users and considering that each peer is a provider and a consumer at the same time, it would sometimes be unreasonable to demand that each individual sets up a hotspot of her own. That would be

the case for households that participate in this community; each household member would need to set up her own access point.

- Teams can include members who are willing to contribute but can only set up hotspots in areas where demand is limited. For example, a member residing at the outskirts of a city would rarely have the oppportunity to provide service (and thus gain credits so that she can enjoy WLAN roaming in the future). By joining a team which has hotspots in more crowded places, she can take advantage of the service provided by other team members in areas where there is more demand. In exchange, such team members offer their teams (and, as a consequence, the whole P2PWNC system) larger coverage.

- By grouping members in teams, the total number of peering entities is reduced. Especially in the case of a centralized design of the P2PWNC scheme, management complexity is reduced, since some managerial tasks are delegated to the teams.

Teams are expected to have a small number of members, usually of the order of a few tens. For example, members of a family may form a team. Another example are hotspot owners of a city neighborhood, who may pool their access points to improve their coverage. The case of single member teams is possible, since they are a degenerate case of the team model described.

A team is identified by a public/secret key pair. In the centralized case, these key pairs may be issued by a Trusted Central Authority. In the decentralized case, they are self-generated. Possibly, there will be a team "leader," who will be responsible for maintaining the team's keys and generating new member key pairs and certificates.

2.3.2 Team Members

Team members are roamers who subscribe with a team. They are identified by a public/secret key pair as well, which is issued by the team leader. Membership with a particular team is expressed via a certificate, which has the following format:

$$\textit{Member Certificate} = \{\textit{Team Public Key, Member Public Key}\}_{teamsignature}$$

As it seems, a member certificate associates a team member with the team she belongs to. The signature of the certificate is produced by the team leader using the team's private key. Since the team's public key is contained in the certificate, team membership can simply be checked by verifying the signature using the team's public key.

As stated in Section 2.2, a basic assumption of the system is complete intra-team trust. Namely, a team member is trusted by all others in the same team. It is assumed that a member acquires a certificate and key pair after a face-to-face contact (or similar procedure) and that the decision of accepting her in the team is taken unanimously by the existing team members, so that intra-team trust is guaranteed.

Considering the full trust between team members and the peer model of the P2PWNC, one may bring the necessity of member certificates in question. Why are certificates needed, since peer entities are teams (and not team members) and since the consumption of the members of a team is aggregated under the team's name? The team's public and private keys might be distributed to trusted individuals instead of issuing certificates.

The answer to this question lays in the very nature of teams; for reasons of intra-team trust management, it is important that the consumtion of service that a member does can be discovered. This can be achieved, since the proofs of service provision (*receipts*) contain the consuming member's certificate. Thus, an overconsuming member, that is someone who consumes service disproportionatelly to her colleagues can be discovered and, possibly, punished, via an intra-team procedure.

It should be noted that the relationships between members of a team are not specified in terms of the P2PWNC protocol. Thus, the trust model within a team, the member subscribing procedure, the punishment of over-consuming members, etc. are autonomously arranged by each team.

In the remainder of this document, the terms *roamer*, *team member*, *mobile user*, *mobile node* or, simply, *client* will be used interchangeably to denote the same thing.

2.3.3 Access Points

Each team needs to provide WLAN access via a number of access points that it manages. Access points (APs) are the means by which a team gathers credits so that its members can enjoy WLAN service by foreign teams. An AP's software agent is responsible for access control to the team's network. Also, it is responsible for collecting *receipts* from roaming members of other teams and forwarding them to the *receipt repository*. Normaly, in case a roamer wishes to have WLAN access to a visited AP, the AP will consult the receipt repository (which maintains transaction history) on whether access should be granted to the roamer, and act according to the receipt repository's suggestion. In the end of the WLAN session, the receipt of service provision should be forwarded to the repository.

2.3.4 Receipts

The Peer-to-Peer Wireless Network Confederation is a system based on service exchange among peers. The service unit in this system is a *receipt*. A receipt is a proof of prior transaction between a team member (service consumer) and a service providing team. It contains the roamer's certificate, the provider team's public key, the timestamp indicating when the transaction begun and the receipt *weight*, which is the amount of traffic forwarded by the access point on behalf of the roamer during the session. Finally, a receipt contains the signature of the service consumer on the above information.

That is, the roamer digitally signs the receipt, thus acknowledging the amount of service that she has enjoyed during a transaction with a visited access point. A receipt has the following format:

$$\textit{Receipt} = \{\textit{Consumer Cert., Provider PK, Session Timestamp, Weight}\}_{member\,signature}$$

The weight value of a receipt is measured by the service providing AP. Thus, by signing a receipt, the mobile user accepts the AP measurement. To avoid being exploited by malicious clients, who will consume resources and eventually refuse to sign a receipt (leaving the service unacknowledged), the AP will normally request receipts from a roamer periodically during a session. Thus, during a typical session, a number of receipts is generated, all of which share the same certificate, provider public key and timestamp. The weight value strictly increases as time goes by. However, only the last receipt is valid for a session. Therefore, the receipt characterizing a session can be uniquely identified by the following tuple (Duplicate receipts are not permitted - the ones with smaller weight value are ignored):

$$\textit{Receipt ID} = \{\textit{Consumer Cert., Provider Public Key, Timestamp}\}$$

Receipt validity can be checked by verifying the two signatures that are contained in it: First, the signature contained in the consumer certificate, so that the roamer's membership with the specified team is ensured, and, second, the receipt signature, which is verified using the member's public key (included in her certificate).

2.3.5 Receipt Repository

History of transactions is maintained in a *receipt repository*. Receipts are generated after roamer-access point sessions and are forwarded for storage to this receipt database. Such a repository must support for the following operations:

- Receipt insertion
- Receipt lookup
- Receipt deletion
- Decisions on service provision

The last operation needs to be further explained. Since the receipt repository keeps track of the system's history and since service provision decisions are based on this information, there must be an algorithm whose input are the identifiers of two teams (provider and consumer) as well as the transaction history and its output should be a decision on whether service should be provided to the potential consumer.

Since there are two possible deployment schemes for the P2PWNC presented in this document, one centralized and one decentralized, the receipt repository can appear in two forms. In the centralized case, there is a single receipt repository maintained by a trusted central authority. In the decentralized case, each team keeps its private view of the system's history in its own repository.

2.3.6 Trusted Central Authority

This entity appears only in the centralized P2PWNC design. As mentioned previously, it is responsible for maintaining the global transaction history of the system. Apart from maintaining the receipt repository, it acts as a central team registry. The TCA issues key pairs for teams.

However, it is designed so that it has the minimum possible intervention in the system. In the centralized case, the role of the TCA is advisory; it cannot enforce punishments. It can only suggest teams whether they should provide access to visitors or not.

2.3.7 Team Server

In the decentralized case, where there is no global history of transactions, the role of the TCA is played by *Team Servers*. A team server is a trusted entity only inside the boundaries of one team. Each team operates exactly one team server. The additional functionality that it offers over the TCA is that team members can directly communicate with it and send receipts or obtain an updated view of the receipt repository (that is, team members can retrieve receipts from the team server).

Team servers may also exist in the centralized case. In fact, considering agent autonomy, a hybrid system design where some teams follow the suggestions of a TCA and other teams ignore it in favor of their local team server is feasible.

2.4 Architecture

2.4.1 Centralized Design

In the centralized case of the P2PWNC there is one global receipt repository (RR) maintained by the TCA. Figure 2.1 depicts this architecture. As one can see, there are three teams whose members roam around the access points of each other. Mobile users communicate with visited access points over wireless links, while access points connect to the TCA over the Internet.

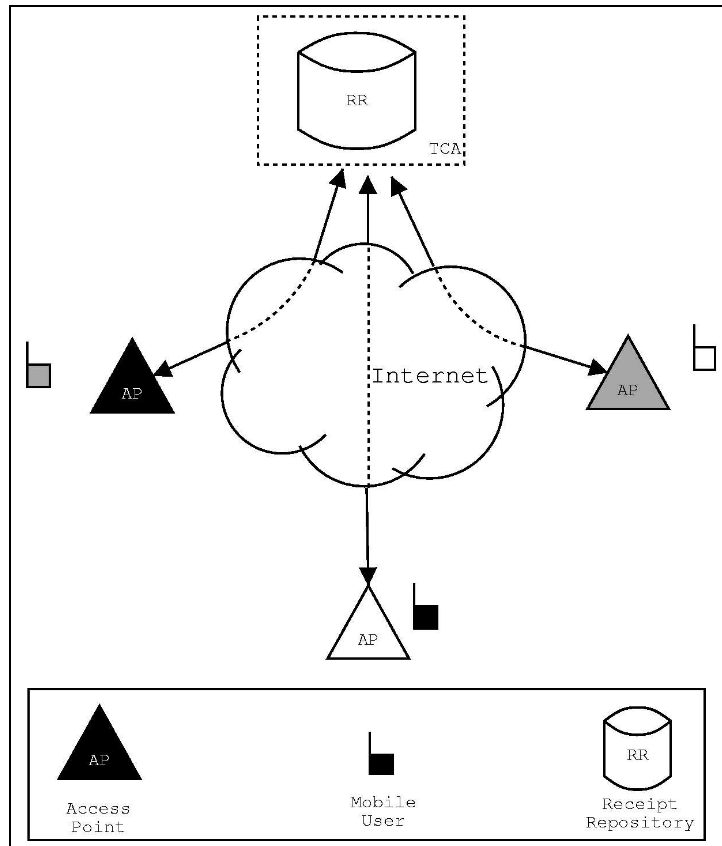

Figure 2.1: P2PWNC centralized mode

2.4.2 Decentralized Design

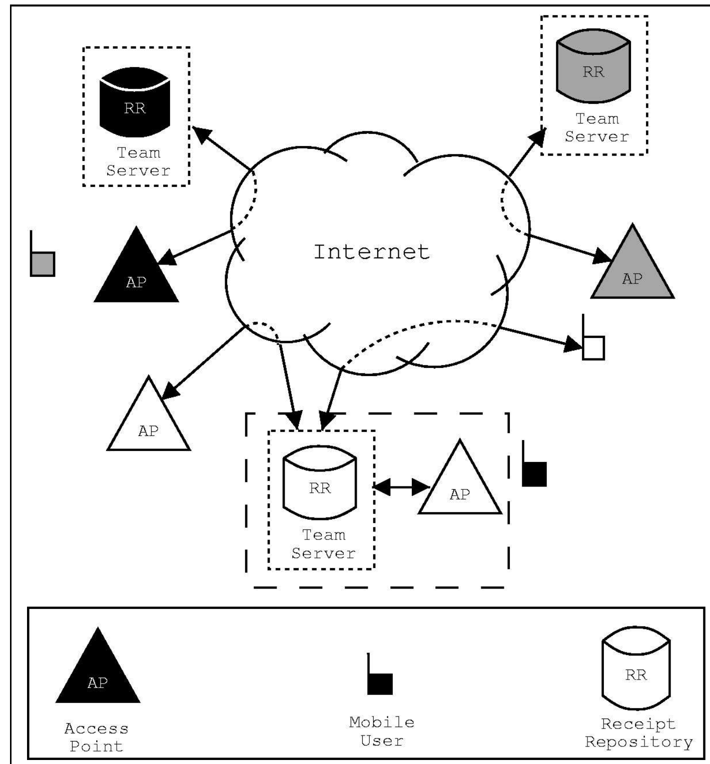

Figure 2.2: P2PWNC decentralized mode

As previously stated, in the decentralized case, each team has a local receipt repository which is managed by the team server. Team members can communicate with the team server to update their receipt repository, which they carry with them. This communication may be carried out over the Internet. Several security issues therefore arise. However, they are considered out of the scope of this document.

As shown in Figure 2.2, each team communicates with its own team server, which handles the team's repository. It should be noted that the team server might as well be collocated with an access point. In Figure 2.2 the white team server is physically collocated with one of the team's APs.

3 Protocol Specification

3.1 P2PWNC Protocol Messages

3.1.1 General

Communication between the entities of the P2PWNC system is carried out through a set of simple ASCII-based messages. These core messages are enough to specify communication between each pair of these entities, namely between the access point module and the client, the access point and the trusted central authority, a client and her team server (in the decentralized case) and the access point and its team server. This minimal set of messages can also be used for future extensions to the system or for operations that are not specified in our system and by the protocol. For example, a distributed receipt database that stores a team's receipts can be built on top of a team's access points making use of some of the existing messages. In the following paragraphs the set of protocol messages, their semantics and their usage are described in detail. Also, the ABNF [16] specification of our protocol is given.

The messages of the P2PWNC protocol are plain text. There were numerous reasons that lead to this choice. First, text messages are human-readable, which makes them more comprehensive. Second, they are more convenient when protocol debugging comes to question. Taking the speed of modern wireless network infrastructure (e.g. the 54Mbps 802.11g protocol) into consideration, the overhead incurred by text representation of data is negligible.

As to the format of the messages, they are composed of a four-charactered message header, which represents the message type. These four characters are followed by a mandatory "Content-length" header, which indicates the number of octets that follow the content length header. All line breaks are carriage-return line-feed (CRLF) sequences. The content length header is followed by other headers and data, which are message specific. Binary

data that are not human-readable are Base64 [24] encoded. Timestamps represent UTC time and are encoded according to the RFC 822 [15] format.

3.1.2 Core Protocol Messages

What follows is a list of these messages and a short description of each one of them.

CONN This message initiates a session between a roamer and a visited access point. It is sent by the roamer and contains her certificate, so that the visited access point can verify the roamer's membership with the certificate issuer team.

CACK This message indicates that a roamer has been admitted access to a visited wireless network. Normally, it is the access point's response to a CONN message. The CACK response includes the session's timestamp, namely the time that the session has started. All subsequent receipts that a client is to sent to the access point as an acknowledgment of the provided service during the session should include this timestamp.

RREQ Periodically, the access point requests that the roamer acknowledges that she has consumed a specific amount of service during the session. This request is performed by means of an RREQ message. This message includes the traffic that the roamer has initiated during the session, measured in bytes, and the provider team's public key. The roamer is supposed to immediately reply with an RCPT message, which is described below.

RCPT An RCPT message is the basic unit of information about service provisioning in the P2PWNC system. It represents a transaction between a roamer and the access point of another (service providing) team. An RCPT message contains the roamer's certificate (roamer's team public key, roaming member's public key, team signature), the service providing team's public key, the amount of traffic initiated by the mobile user during the session (receipt "weight," measured in bytes) and the session's timestamp. All the above are digitally signed by the roaming member using her private key. This signature is included in the RCPT message, thus acknowledging service consumption on behalf of the client. It should be noted that during each session, many receipts may be generated. All these receipts have the same timestamp (beginning of the session) but have increasing "weight" values, since they are client responses to the periodical RREQ messages sent by the access point. Eventually,

only the last receipt of a session is significant for the system, since it summarizes the amount of service the provider has offered to a roamer during the session in question. Being the system's means of measuring service provision/consumption, an RCPT message is also used with the exact same structure in many other cases in the P2PWNC protocol.

QUER This message is used by an access point to inquire whether access should be granted to a visiting roaming user. It contains the public key of the provider team and the public key of the team to which the roamer belongs. A QUER message is sent to a system module that is capable of deciding whether the requesting member (mobile user) should be granted admission to the provider team's wireless local area network. In the centralized case, such a module can be a Trusted Central Authority (TCA), while in the decentralized case, the decision about the clients admission is taken normally by the provider's Team Server (which might as well be collocated with the service provisioning access point).

QRSP A QRSP message is the reply to a QUER one. It is issued after the module responsible for deciding whether the client should be granted access has processed the QUER message. The header field "Action," which can take the values "Grant" or "Forbid" indicates the outcome of the decision function.

RRSP This message is the reply of the entity responsible for storing receipts (typically the TCA or a Team Server) when it is sent a receipt. In case of an error while handling the receipt, the RRSP message contains information about the reason of the error.

UPDT An UPDT message can be issued by a roamer so that she can refresh her receipt repository. It is sent to the roamer's home Team Server. This message contains a timestamp field. The Team Server that receives an UPDT message sends the roamer the receipts whose timestamp is more recent than the one specified by the roamer. Obviously, this functionality is only necessary in the distributed case. In the centralized one, there is no need for a mobile user to carry receipts, since all transactions are stored in the Trusted Central Authority's repository.

3.1.3 Augmented BNF for the P2PWNC Protocol

3.1.3.1 Generic Data Types

In this section, the syntax of some basic data types that are in use throughout the protocol specification is introduced. The set of these data types contains decimal integers, Base64 encoded data and timestamps. As mentioned before, timestamps are represented according to the RFC822 protocol [16]. The rule name that expands to a timestamp of this format is ⟨date-time⟩. The ABNF syntax specification for this rule is omitted. However, an example of such a timestamp follows:

```
Mon, 23 May 2005 16:13:39 +0000
```

For decimal integers, the syntax is presented below:

```
⟨DEC-INT⟩ ::== 1⟨NON-ZERO⟩ *DIGIT
⟨NON-ZERO⟩ ::== "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

As far as Base64-encoded data are concerned, the alphabet used are upper and lower case ASCII characters and the characters “+” and “-”. The character “=” is used for padding. In the Base64 encoding specification, the output character stream must be formatted in lines of no more than 76 characters each. In the P2PWNC protocol, the maximum length of such lines is limited to 64 characters. Lines are separated by a single CRLF sequence. The specification of the above is as follows:

```
⟨B64-CHAR⟩ ::== ALPHA | DIGIT | "+" | "/"
⟨B64-PAD⟩ ::== "="
⟨B64-TERM-LINE⟩ ::==
          64⟨B64-CHAR⟩
        | *63⟨B64-CHAR⟩ ⟨B64-PAD⟩
        | *62⟨B64-CHAR⟩ 2⟨B64-PAD⟩
⟨B64-NON-TERM-LINE⟩ ::== 64⟨B64-CHAR⟩ CRLF
⟨B64-DATA⟩ ::== *⟨B64-NON-TERM-LINE⟩ ⟨B64-TERM-LINE⟩
```

3.1.3.2 Generic Message Format

P2PWNC protocol messages are plain ASCII messages that conform to the following generic pattern:

```
⟨P2PWNC-MSG⟩ ::==
          ⟨MTYPE⟩
          ⟨CLEN-HDR⟩
```

```
          [⟨EXTRA-HDRS⟩]
          [CRLF ⟨EXTRA-DATA⟩]
```
where

```
⟨MTYPE⟩ ::== 4ALPHA CRLF
⟨CLEN-HDR⟩ ::== "Content-length:" SP ⟨DEC-INT⟩
⟨EXTRA-HDRS⟩ ::== *(CRLF ⟨HDR⟩)
⟨HDR⟩ ::== ⟨HDR-NAME⟩ ":" SP ⟨HDR-VAL⟩
⟨HDR-NAME⟩ ::== 1*CHAR
⟨HDR-VAL⟩ ::== 1*CHAR
⟨EXTRA-DATA⟩ ::== 1*CHAR
```

Therefore, the generic P2PWNC message format is described by a four-character message type field, followed by a mandatory content length header, whose value is the number of octets contained in the message after the first CRLF encountered (that is the CRLF at the end of the content length header). The "Content-length" header is then followed by a number of (optional) other headers with similar structure to that of the "Content-length" header and other optional message specific data, such as a Base64 encoded public key.

The order of the additional headers is important and it is message specific. All strings are case insensitive, apart from any Base64 encoded data, for which case is of importance. It should also be noted that whitespace following a header name should be limited to exactly one space. Therefore, the following headers are not equivalent; according to the P2PWNC protocol, the former is malformed:

```
Content-length:    100
Content-length: 100
```

However, the headers

```
CoNtEnT-leNGTh: 100
Content-length: 100
```

are both equivalent and well-formed.

3.1.3.3 Message-specific Headers

Having specified the generic format of message headers previously, the message-specific headers are going to be presented with the use of ABNF notation. These headers are used by various protocol messages. ⟨ALGORITHM-HDR⟩ is the header that represents the public key cryptography algorithm used (RSA or ECC) and typically looks like:

```
Algorithm:  ECC160
```

The ⟨TIMESTAMP-HDR⟩ header represents timestamps according to the date and time format introduced in the RFC822 standard. Finaly, ⟨WEIGHT-HDR⟩ refers to the “Weight:” header, that is used in RCPT messages.

```
⟨ALGORITHM-HDR⟩ ::== "Algorithm:" SP ⟨ECCID⟩ | ⟨RSAID⟩
⟨ECCID⟩ ::== "ECC" ("160"|"192"|"224"|"256")
⟨RSAID⟩ ::== ("1024"|"1536"|"2048")

⟨TIMESTAMP-HDR⟩ ::== "Timestamp:" SP ⟨date-time⟩

⟨WEIGHT-HDR⟩ ::== "Weight:" SP ⟨DEC-INT⟩

⟨ACTION-HDR⟩ ::== "Action:" SP ("Grant" | "Forbid")
```

3.1.3.4 Protocol Messages

The set of protocol messages that comprise the P2PWNC protocol was presented in 3.1.2 In this section, the syntax of these messages is specified. In addition, examples of each message type are cited. The names of the rules in the specification of the messages are self-explanatory. For example, the rule name ⟨CONN-MSG⟩ implies the “CONN” message. It should also be noted that emphasis is given on the syntax of the messages described, rather that their semantics. However, in situations that it is considered necessary, further explanations on the meaning of fields will be provided.

CONN message

```
⟨CONN-MSG⟩ ::==
        "CONN" CRLF
        ⟨CLEN-HDR⟩ CRLF
        ⟨ALGORITHM-HDR⟩ CRLF
        ⟨B64-DATA⟩
```

The B64-DATA field, in the case of a CONN message, contains the sender’s P2PWNC certificate, represented as a character stream output by a Base64 encoding procedure. The details of the procedure that results to this representation are given in 3.3.3.

Example:

```
CONN
Content-length: 187
```

```
Algorithm:  ECC160
BNibmxStfJlod/LnZubH6pzWHQqKyZFcSMjnZurmTe4KjCRkllhV93MEegPvCsxz
2oe/hqevoPSrwO1JLO/36J8HTIeyeKQqTCfx+EPxweAvYC/ZFb8URLa2faIbvSgD
3lm6Wa1S4cYlSWeSNmFzS/ebDFfzakqNSEs=
```

CACK message

```
⟨CACK-MSG⟩ ::==
          "CACK" CRLF
          ⟨CLEN-HDR⟩ CRLF
          ⟨TIMESTAMP-HDR⟩
```

Example:

```
CACK
Content-length:  42
Timestamp:  Tue, 24 May 2005 17:26:41 +0000
```

RREQ message

```
⟨RREQ-MSG⟩ ::==
          "RREQ" CRLF
          ⟨CLEN-HDR⟩ CRLF
          ⟨ALGORITHM-HDR⟩ CRLF
          ⟨WEIGHT-HDR⟩ CRLF
          ⟨B64-DATA⟩
```

The last field of an RREQ message is the Base64-encoded public key of the provider team. For more information on the representations of public keys one should refer to 3.3.1.

Example:

```
RREQ
Content-length:  89
Algorithm:  ECC160
Weight:  6336
BEXn8BHHViQ/YMyF2ny+KaI4YXz+W60uED7R8wZefDznyncfQKggzAc=
```

RCPT message

```
⟨RCPT-MSG⟩ ::==
          "RCPT" CRLF
```

⟨CLEN-HDR⟩ CRLF
⟨ALGORITHM-HDR⟩ CRLF
⟨TIMESTAMP-HDR⟩ CRLF
⟨WEIGHT-HDR⟩ CRLF
⟨BASE64-DATA⟩

The ⟨BASE64-DATA⟩ field, in this case, represents the fields of a receipt that are not human readable, namely the consumer's certificate, the provider's public key and the signature of the receipt. For a detailed description of the procedure that generates this representation one should refer to 3.3.4.

Example:

```
RCPT
Content-length:  357
Algorithm:  ECC160
Timestamp:  Tue, 24 May 2005 17:26:41 +0000
Weight:  6336
```


```
BNibmxStfJlod/LnZubH6pzWHQqKyZFcSMjnZurmTe4KjCRkllhV93MEegPvCsxz
2oe/hqevoPSrwO1JLO/36J8HTIeyeKQqTCfx+EPxweAvYC/ZFb8URLa2faIbvSgD
3lm6Wa1S4cYlSWeSNmFzS/ebDFfzakqNSEsERefwEcdWJD9gzIXafL4pojhhfP5b
rS4QPtHzBl58POfKdx9AqCDMBxRoGALKJSJYYXlsrwtiyZJKvPlU5B3lWrFuL25P
d+kv2iMVRElXk/4=
```


QUER message

⟨QUER-MSG⟩ ::==
"QUER" CRLF
⟨CLEN-HDR⟩ CRLF
⟨B64-DATA⟩ 2CRLF
⟨B64-DATA⟩

In this case, clarification may be needed as to the ⟨B64-DATA⟩ fields contained in the body of the message. As stated in 3.1.2, the first ⟨B64-DATA⟩ field represents a service provider and the second a potential service consumer. A noticeable difference between QUER and the other messages of the protocol is the precence of two consecutive carriage return - line feed sequences separating the two public key representations. The rationale behind this choice is that it assists in distinguishing between the two keys, which may have different bit length.

Example:

```
QUER
Content-length:  116
```


```
BEXn8BHHViQ/YMyF2ny+KaI4YXz+W60uED7R8wZefDznyncfQKggzAc=

BNibmxStfJlod/LnZubH6pzWHQqKyZFcSMjnZurmTe4KjCRkllhV93M=
```


QRSP message

```
⟨QRSP-MSG⟩ ::==
          "QRSP" CRLF
          ⟨CLEN-HDR⟩ CRLF
          ⟨ACTION-HDR⟩
```

Example:

```
QRSP
Content-length:  14
Action:  Forbid
```

RRSP message

```
⟨RRSP-MSG⟩ ::==
          "RRSP" CRLF
          ⟨CLEN-HDR⟩ CRLF
          ("OK" | ("ERR" [SP *CHAR]))
```

As one can observe, the status code contained in an RREQ message can either be “OK” or “ERR.” In the latter case, ERR can optionally be followed by an explanation of the error that has taken place. The explanation can be an arbitrary application-specific character stream, which is not specified in the protocol.

Example:

```
RRSP
Content-length:  2
OK
```

UPDT message

```
⟨UPDT-MSG⟩ ::==
          "UPDT"
```

⟨CLEN-HDR⟩ CRLF

⟨TIMESTAMP-HDR⟩

As to the UPDT message format, it is considered rather simple and self-explanatory, so no further details need to be provided.

Example:

```
UPDT
Content-length: 42
Timestamp: Sat, 09 Apr 2005 14:53:35 +0000
```

3.2 Cryptography

3.2.1 Supported Cryptosystems

The P2PWNC protocol supports both the RSA and the Elliptic Curve cryptosystem. Protocol messages that carry cryptographic data in their bodies include an "Algorithm" header, which indicates the algorithm that has been used for the generation of those data. The "Algorithm" header is followed by a semicolon, a space character and the algorithm identifier. This identifier starts with "RSA" or "ECC" and is followed by the length in bits of the keys associated with these data. For example, in an RCPT message, the algorithm field specifies the receipt signature algorithm. If the signer of the receipt (mobile user) makes use of a 1024 bit RSA private/public key pair, the value of the "Algorithm" header will be "RSA1024." In a similar fashion, if a visited access point's team public key follows an 160-bit elliptic curve cryptography standard, the "Algorithm" field of an RREQ message (carrying the access point's public key) will be "ECC160."

Although the specification of the P2PWNC protocol includes both the RSA and the ECC cryptosystems, not all of the system's agents need to be both ECC- and RSA-enabled. The Trusted Central Authority (centralized mode), the Team Server (decentralized mode) and the access point modules need to be capable of performing both RSA and ECC cryptographic operations. It is necessary for the TCA to be able to generate both kinds of key pairs and to be able to verify receipts. In a similar fashion, team servers may verify and store receipts generated using any kind of keys and, thus, should support for both cryptosystems. As to the access point modules, they need to be capable of verifying receipts, which can come from both RSA- and ECC-enabled mobile users. Roamers, on the other hand, are not obliged to have support for both cryptosystems, since the only cryptographic

operation that they perform is receipt signing.

However, even if there is no need for a client to be capable of handling both cryptosystems, it is advisable that implementations of mobile users' software agents have support for both RSA and Elliptic Curve cryptography.

3.2.2 Key Generation

Each team, that is each of the "peers" of the system, is identified by a unique public/private key pair. In the centralized case, this key pair might as well have been generated by a Trusted Central Authority. In the decentralized case, teams can generate their identities themselves. In either case, teams generate key pairs and certificates for their members independently, that is without the intervention or control of any other party.

It is mandatory that the keys and certificates of members follow the same algorithm as the issuing team's. Namely, a team that is identified by a 1024-bit RSA public key, MUST generate RSA-1024 key pairs for its members. As a result, the three parts of a member certificate, that is the issuing team's public key, the mobile user's public key and the certificate signature are characterized by the same algorithm.

3.2.3 RSA Cryptosystem Parameters

An RSA key pair is described by three numbers; the public modulus n, the public exponent e and the private exponent d. The public key is composed of the pair (e, n) and the private key is composed of the pair (d, n). In the P2PWNC protocol, the public exponent is fixed to 65537, so it need never be encoded in the public key data. An RSA key pair is characterized by the bit length of its modulus (n). Acceptable values for the length of the modulus in the P2PWNC are 1024, 1536 and 2048. Smaller keys are considered insecure, while longer keys impose a greater overhead on the agents while performing cryptographic operations. Since the protocol has been designed with embedded devices with limited processing power in mind, 1024 bit keys are considered the appropriate solution and are thus recommended for use within the P2PWNC.

3.2.4 Elliptic Curve Cryptosystem Parameters

As far as elliptic curve cryptography is concerned, the choice of the parameters follows the ANSI X9.62 standard. The elliptic curve domain parameters used in the P2PWNC protocol

are recommended in [21]. In particular, the curves that are employed for the generation of key pairs are *verifiably random* named curves.

For the purpose of being easily identified, recommended elliptic curves have been given nicknames [21]. This naming convention was proposed by the "Standards for Efficient Cryptography" group. Curve nicknames used in the P2PWNC protocol begin with "sec", followed by the letter "p" to indicate that the curve parameters are over the F_p finite field. This is followed by the field size p in bits (160, 192, 224 or 256 bits). Finally, there is the letter "r" to indicate that the curve parameters are verifiably random followed, by a sequence number. Table 3.1 summarizes the nicknames of the valid elliptic curve domain parameters in the P2PWNC protocol. The first column shows the size of the field on which the curves are defined, while the second column shows the nicknames of the curves.

Table 3.1: Valid ECC curves in the P2PWNC protocol

Size	Curve Nickname
160	secp160r1
192	secp192r1
224	secp224r1
256	secp256r1

Thus, in the case of an "ECC160" algorithm header field, the curve with the nickname "secp160r1" is implied. In a similar fashion, "ECC192" implies the "secp192r1" curve. Acceptable bit lengths (denoted as n) are 160, 192, 224 and 256. However, the use of the "secp192r1" curve is recommended, for the same reasons that 1024 bit RSA keys are considered more appropriate among other bit lengths. Tests on the time efficiency of cryptographic operations for various key types and sizes that are presented in 5.3 suggest that this is the most appropriate key choice. In an ECC key pair, the secret key is an integer of bit length n and the public key is a point in the curve, defined by its x and y coordinates.

3.3 Data Representation

3.3.1 Public and Private Keys

In this section the representation of keys will be described. In fact, only the representation of public keys is of importance to the P2PWNC protocol, since only public keys may

actually be exchanged.

As to RSA keys, as previously mentioned, the public exponent is fixed to 65537. Therefore, it need not be encoded with the public key data. The part of the RSA public key that is actually encoded is the public modulus. Whenever an RSA public key is mentioned, its public modulus will actually be implied. Each time a public key is to be transmitted through the network, it should be converted to its "wire" format, namely the Base64 encoding of its modulus.

ECC public keys are represented without the use of the point compression technique [20]. The key is encoded as an octet string whose leftmost byte signifies that point compression is off. The remainder of the byte array contains the x coordinate of the public key (a point of an elliptic curve) followed by its y coordinate. The length of this representation is twice the value of n plus one octet to indicate that point compression is not in use.

3.3.2 Digital Signatures

Agents that are identified by RSA key pairs use the RSA digital signature scheme. In this case, the signer first produces the SHA-1 digest of the data that are to be signed. Following that, she digitally signs the 20-byte digest using her private key. The bit length of the signature is equal to the length of the private key modulus. For example, using 1024 bit RSA keys, the signature is 128 bytes.

In the case of elliptic curve cryptography, ECDSA is the signature algorithm used. The output of the ECDSA operation is a couple of integers $S = (r, s)$. These integers are serialized (in big endian form) into an octet string whose leftmost part is r and its rightmost part is s.

3.3.3 Digital Certificates

The certificates used in the P2PWNC scheme represent membership of a mobile user with a team of users. Thus, they contain the public key of the issuing team, the public key of the team member and the digital signature of the above keys produced using the issuer's secret key. Such a certificate is generated carrying out the following operations in the given order:

- Represent the two public keys as described in the previous paragraph.
- Concatenate the public keys so that the public key of the issuing team occupies the leftmost part of the byte array and the public key of the team member occupies the

rightmost part.

- Generate the digital signature of the produced byte array and represent it as described in the previous paragraph.

The wire format of a P2PWNC certificate is generated as follows:

- Concatenate the octet strings representing the keys and the signature included in the certificate so that the leftmost part of the certificate is the issuing team's public key, followed by the member's public key. The rightmost part of the byte array is occupied by the representation of the digital signature.
- Base64 encode the resulting byte array.

The resulting ASCII string can be concatenated in a P2PWNC protocol message, such as a "CONN" message.

3.3.4 Receipts

A receipt is a proof of prior transaction between a roaming user and a service providing team. As such, it must necessarily contain the roamer's certificate, the public key of the service provider team, a timestamp indicating the beginning of the session and the amount of traffic which the roamer has been charged with, namely the traffic that the client has initiated. Finally, a receipt contains the signature of the consumer of the service, thus acknowledging the service consumption on his behalf that the receipt is representing.

It should be noted that the timestamp represents the amount of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time. As far as the receipt weight is concerned, it is measured in bytes. These values are represented using four-byte integers.

A receipt signature is generated by carrying out the following operations in the specified order:

- Serialize the roamer's certificate in a way that the resulting byte array has the public key of the certificate issuing team as its leftmost part, followed by the public key of the member and having the certificate signature as its rightmost part.
- Append the public key of the service providing team, represented as described in Section 3.3.3, to the octet string that was output by the previous step.

- Convert the receipt timestamp in big endian form if necessary and append its octet representation to the byte array that was output by the previous step.

- Convert the receipt weight in big endian form if necessary and append its octet representation to the byte array that was output by the previous step.

- Finally, digitally sign the octet string that was output by the previous step and contains the certificate of the service consumer, the public key of the service provider, the session timestamp and the session weight making use of the consumer's secret key. The resulting signature is represented as described in Section 3.3.2.

3.4 Entities and their Behavior

3.4.1 System Parameters

The system parameters involved in the operation of the P2PWNC protocol entities are resources like TCP ports and timers. Having stated that the design of our scheme was based on the autonomy of actions of system agents, this set of parameters is limited.

The services that the system offers are built on top of TCP. It should be noted, though, that the use of the specified protocol messages can be used to build a similar system based on UDP. However, in this document, the use of TCP will be implied unless otherwise stated.

As to the standard TCP ports used in the P2PWNC protocol, the access point normally listens for incoming client messages on the 9999 port. The TCA and team server modules listen for connections on the 3333 TCP port. However, it is expected that implementations may give the opportunity to configure the software agents to that they can listen to other ports.

As to the various timers that are used to control the operation and communication of agents and the transition between states, they are not specified here. In the line of the agent autonomy principle, these are left to be defined by the protocol implementors, together with other parameters such as the receipt repository size or the time horizon that sets the lifetime of a receipt.

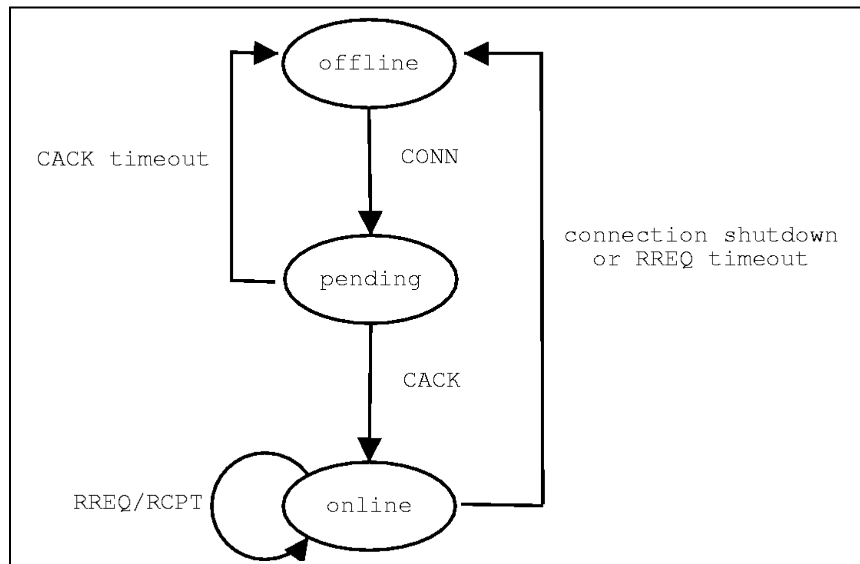

Figure 3.1: Client operation in centralized mode

3.4.2 Centralized Case

3.4.2.1 Client Behavior

In the case of a centralized structure of the P2PWNC scheme, the client need not perform any additional actions in order to request for service, other than send the access point module a CONN message. It is implied that a connection to the standard TCP port that the access point listens for client connections has already been established. After admission control has been performed on the access point side, the client periodically refreshes the session via RCPT messages and finally, when she wishes to discontinue the session, the TCP connection is shut down. There is no explicit disconnection message at the P2PWNC protocol layer.

From the short description that preceded, three states can be distinguished. The client begins in the *offline* state, during which she is not beeing provided service nor does she expect to be provided service. A client is in the *offline* state before having issued a CONN message and after having been disconnected.

The transition from the *offline* state to the *pending* state is performed via a CONN message. When in the *pending* state, a client is waiting for her access request to be evaluated by the access point. A timeout at this point indicates that the CONN request was rejected by the access point or that some other error has occured. Therefore, the client returns to

the *offline* state.

In case the connection request was accepted, access is granted to the client, who thus moves to the *online* state. The mobile user retains her state by accepting RREQ messages by the access point module and replying by RCPT messages. Namely, the access point periodically sends receipt requests (RREQ) and the client may respond with an RCPT, acknowledging the service that has been provided thus far. Failure to provide an RCPT message in time (referring to an access point timeout), a client-side timeout expiration while waiting for an RREQ or a TCP connection shutdown result in a transition back to the *offline* state. The above states and transitions can be seen in Figure 3.1

3.4.2.2 Access Point Behavior

A typical access point software agent is a server that listens for connections by mobile users and initiates a client session for each of them. For each session, state is maintained. State information includes the session timestamp, the certificate of the mobile user, the last receipt sent by the client and the timer specifying when the next RREQ message is scheduled to be sent to the client.

A session between a mobile user and an access point, from the point of view of the access point agent, can be described as the transition between five states. A session is client-initiated and starts as soon as a TCP connection is established. Thus, the access point enters the *session started* state, expecting a CONN message by the mobile user. In case the timer that is associated with the CONN message reception expires, a transition to the *session ended* state takes place. Otherwise, if the CONN message is received in time, the access point enters the *pending* state, during which it is determined whether the mobile user is to be granted access or not.

During the *pending* state, the access point first performs the necessary checks on the client's certificate. In case of a verification error or other error concerning the client's credentials, the TCP connection (e.g. connection shutdown) or the access point's internal functions a transition to the *session ended* state occurs. If the above checks are passed successfully, the access point communicates with the Trusted Central Authority (the receipt repository) via a QUER message to decide about granting access. Depending on the TCA response, there may be a transition to the *connected* state (if the QRSP message indicates that access should be granted) or to the *session ended* state (in case of a QRSP message with an "Action: Forbid" header). A move to the *session ended* state also happens in case, in a similar fashion as in other cases, the TCA delays to respond to the QUER message.

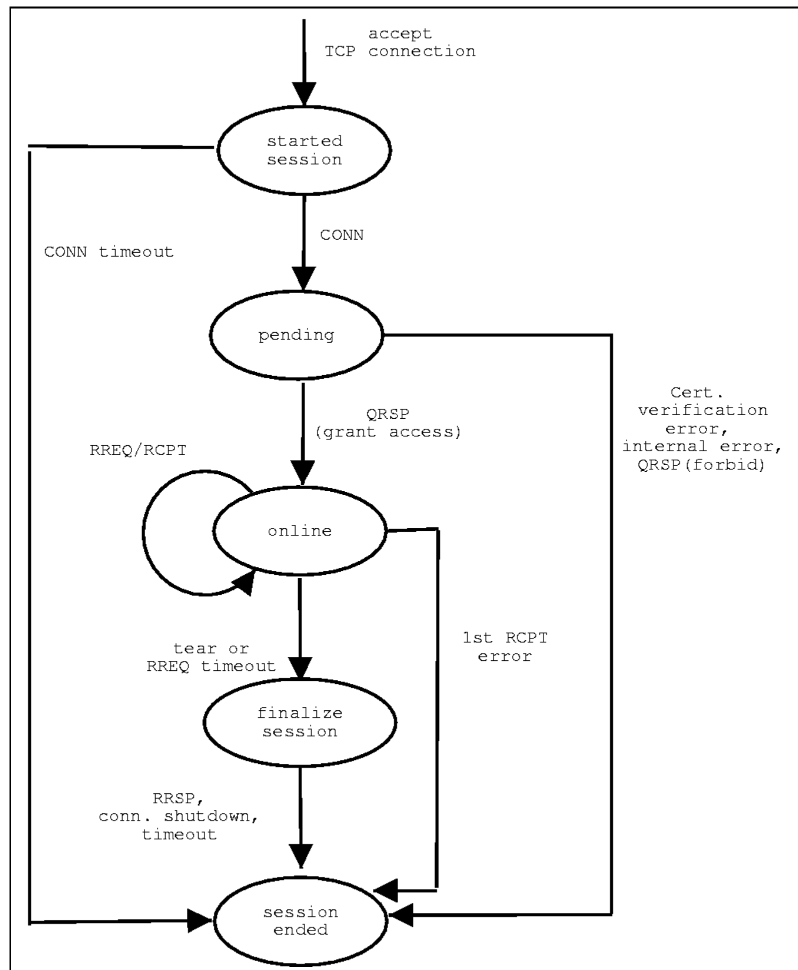

Figure 3.2: Access point client session in centralized mode

If the TCA suggests that access should be granted, after the performance of necessary (application specific) actions associated with providing service to the mobile user (such as setting up firewall parameters or setting up traffic measurement capabilities for the session), there is a transition to the *online* state.

As its name implies, in this state the mobile user can enjoy internet access. This state is retained as long as the mobile user responds in time to the session refreshment messages, that is receipt requests (RREQ). Receipts are requested in regular intervals. Each time an RREQ message is to be constructed, the access point measures the amount of traffic initiated during the session in question and requests that the mobile uses acknowledges this service via an RCPT.

In case there is a failure in the verification of the first receipt or the session is ended before the first receipt has been acquired, the access point automatically moves to the *session ended* state. Otherwise, the next transition will be from the *online* to the *finalize session* state and will occur as soon as the access point is notified that the connection has been terminated by the client or if the timer associated with the receiption of the next receipt expires.

In the *finalize session* state, operations concerning freeing up resources that have to do with this session are released and the last receipt of the session is sent to the central receipt repository (TCA). The access point may wait for an explicit RRSP message from the TCA or may shut down the connection to the TCA. Eventually, the session is ended via a transition to the *session ended* state.

The above procedures are visualized in Figure 3.2.

3.4.2.3 TCA Behavior

The operation of the TCA is simple in terms of the P2PWNC protocol specification. Although internally rather complicated operations may take place, concerning receipt storage and decisions on granting access, they are not specified in this section. Generally, it approximates stateless behavior.

The TCA listens for QUER and RCPT messages, since its role is to store receipts and suggest P2PWNC service providers on granting or forbidding access to visitors. A TCA session is described as the exchange of two messages with the access point.

- In case of a QUER message, the TCA replies with a QRSP.
- In case of an RCPT message, the TCA replies with an RRSP.

3.4.3 Decentralized Case

3.4.3.1 Client Behavior

The client has a slightly modified behavior and some additional capabilities, compared to her operation in centralized mode. Namely, the mobile user has the option of initiating the session by sending the visited access point receipts, so that she can increase the possibility that she will be granted access. After sending an arbitrary amount of receipts, she should send the CONN message. The transaction between the roamer and the access point is then continued in the same manner as in the centralized case.

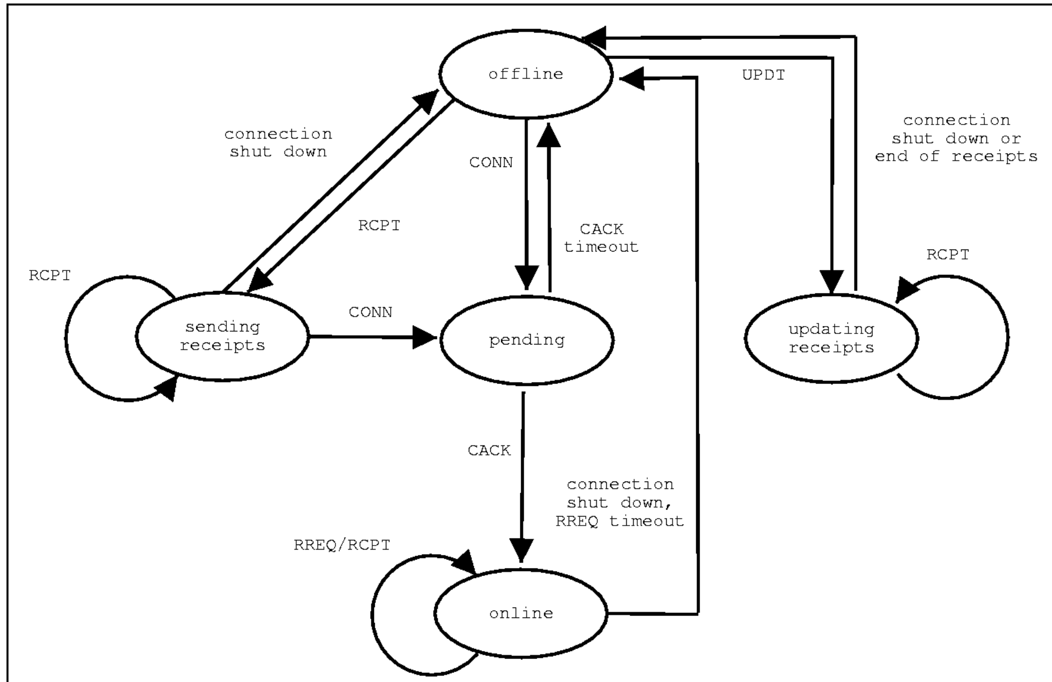

Figure 3.3: Client operation in the decentralized operation mode

Considering the fact that a team's receipt repository is dynamically updated (older receipts are replaced by new ones), the client can arbitrarily communicate with her team server so that she can update her repository via an UPDT message. How communication with the team server is achieved is not specified here. That is, details concerning security and authentication issues or the underlying network infrastructure used are not specified in terms of the P2PWNC protocol. For example, it is supposed that the roamer and the team server may have eshtablished a secure channel to exchange messages and this may be done over a network such as GPRS.

Two additional states are thus introduced. If the client is about to request service from a service providing access point, she enters the *sending receipts* state by sending an RCPT message to the access point. She remains in this state by sending RCPT messages to the access point and makes a transition to the *pending* state via a CONN message. Otherwise, she moves back to the *offline* state in case the connection to the access point is shut down or if she fails to send the CONN message in time.

The other additional state, compared to the centralized state machine, is the *updating receipts* state, to which the client enters by sending an UPDT message to its team server.

This state is preserved while the client keeps receiving RCPT messages from the team server. Finally, she returns to the *offline* state when there are no more receipts to be sent. This is not stated explicitly; rather, it is indicated by a timer expiration or an abrupt connection shutdown. The state machine describing the client functions when operating in the decentralized mode can be seen in Figure 3.3.

3.4.3.2 Access Point Behavior

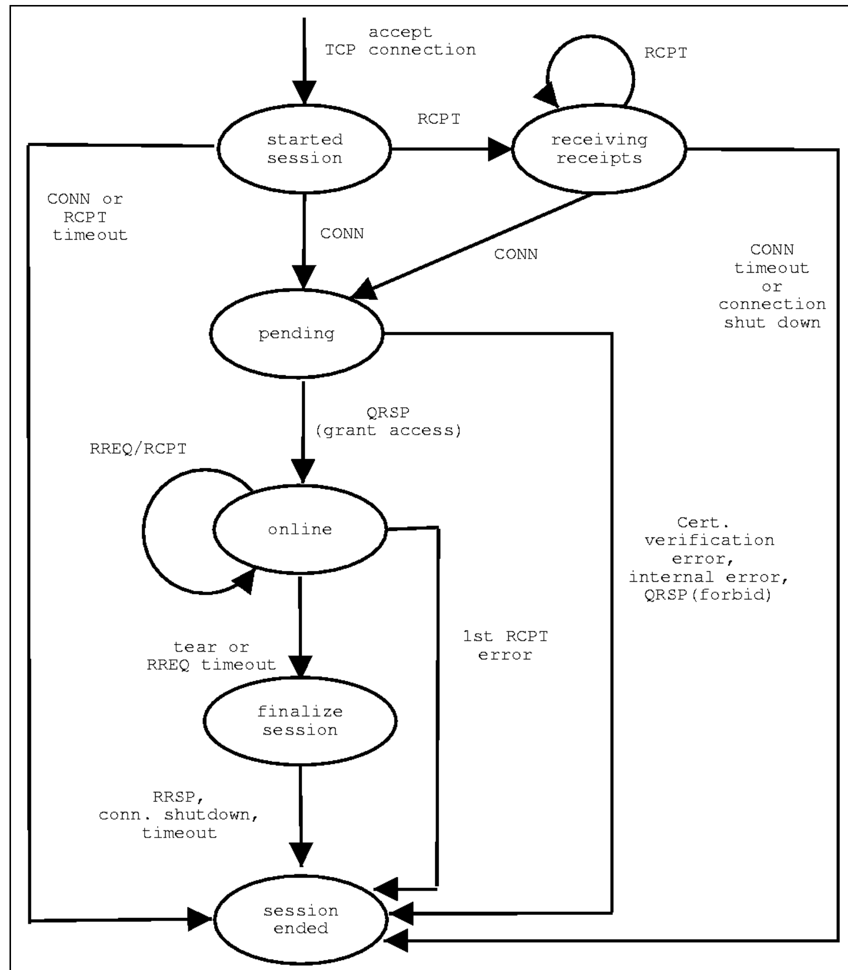

Figure 3.4: Client operation in the decentralized operation mode

The access point module, behaves in a similar manner as in the centralized case. However, as seen in Section 3.4.3.1, it should be capable of receiving receipts in the beginning of the session.

The operation of the access point agent in the decentralized version of the P2PWNC scheme is presented in the state machine of Figure 3.4. The additional state introduced here is the *receiving receipts* state. The transition to this state is performed via an RCPT message received from the client. This must be the first message in the session, if the client wishes to provide receipts to the access point. Otherwise, the message sequence during the session will be identical to that of the centralized case. The way RCPT messages received while the access point is in the *receiving receipts* state will be handled is not specified in the protocol. Normally, they are automatically forwarded to the team server module, which might as well be collocated with the access point. That is, the access point and the team server, although logically independent, may be physically the same entity.

The TCA module is now replaced by the team server. Thus, the QUER, QRSP and RRSP messages, as well as the RCPT message in reply to which the RRSP message is received, are all exchanged between the access point and the team server.

3.4.3.3 Team Server Behavior

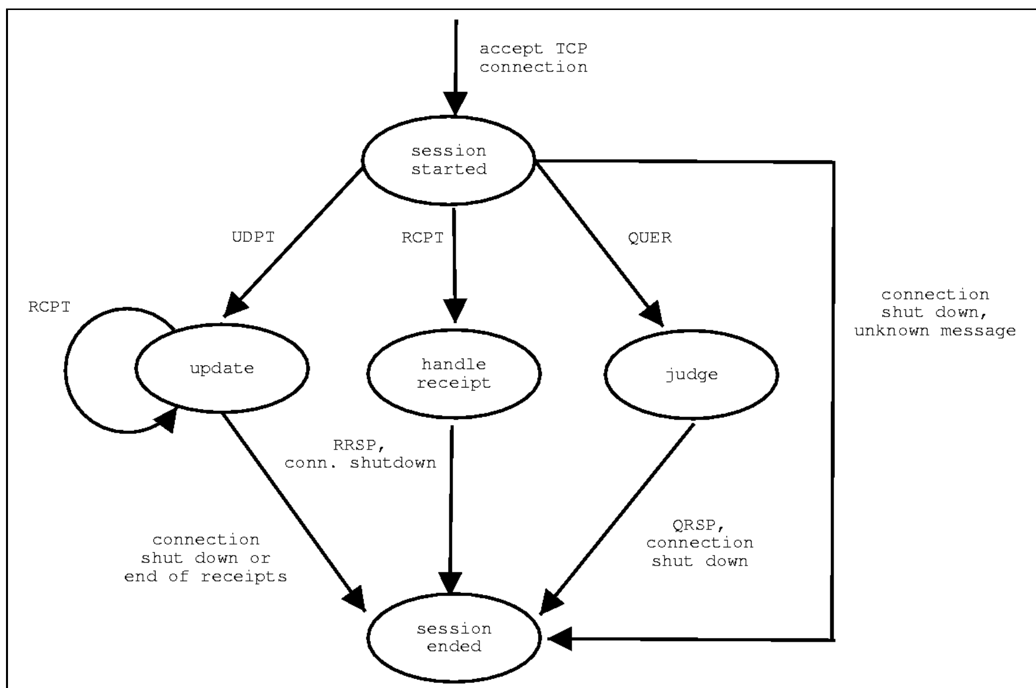

Figure 3.5: Team server operation

The team server acts as a central receipt repository for the access points and the mem-

bers of a team. Its operation resembles that of the TCA in the centralized case, except that it is capable of replying to UPDT queries by a team's members.

In Figure 3.5, its operation is specified. As one can see, a team server session begins as soon as a TCP connection is established between this module and another system entity. At this point, the team server is in the *session started* state. Depending on the type of message that will be received, a transition will take place. Four cases can be distinguished:

- Reception of an UPDT message.
  In this case, the update operation of a client's repository is to take place. Normally, this session is initiated by a client that wishes to update its repository. This state is preserved by the team server while receipts are being sent to the requesting client. As soon as the update operation finishes, the team server moves to the *session ended* state.

- Reception of an RCPT message.
  The team server will now move to the *handle receipt* state. The actions that are performed when such a message is received are not specified by the protocol. Normally, the receipt will be placed in permanent storage, after necessary checks will have been performed on it. A transition to the *offline* state is performed via an RRSP message (or a connection shutdown, as usual) sent to the requesting entity.

- Reception of a QUER message.
  A QUER message leads the team server to the *judge* state. As in the case of an RCPT message, the operations to be performed are not specified. Namely, the methods used by this module to derive an answer to the query are implementation-dependent. A transition to the *offline* state follows a QRSP message sent by the team server to the entity that initiated the session.

- Reception of a message of an unknown type or connection shutdown.
  In this case, the team server simply moves to the *offline* state.

3.5 Message Sequences

In this section diagrams that present the sequence of messages in the communication between P2PWNC system entities are shown. Only typical cases where entities behave normally and without failures are included here. Time intervals between messages are arbitrary, since timeouts are not specified in the protocol (as noted in Section 3.4.1).

In Figure 3.6 the messages exchanged during a mobile user - access point session in the centralized scheme are shown. The same case is presented in 3.7 for the decentralized case. It should be noted that in this Figure, the sequence of RCPT messages at the beginning of the session is optional. Finally, in Figure 3.8 one can see the messages that are exchanged between a team member and its home team server, when the former is updating her receipt repository.

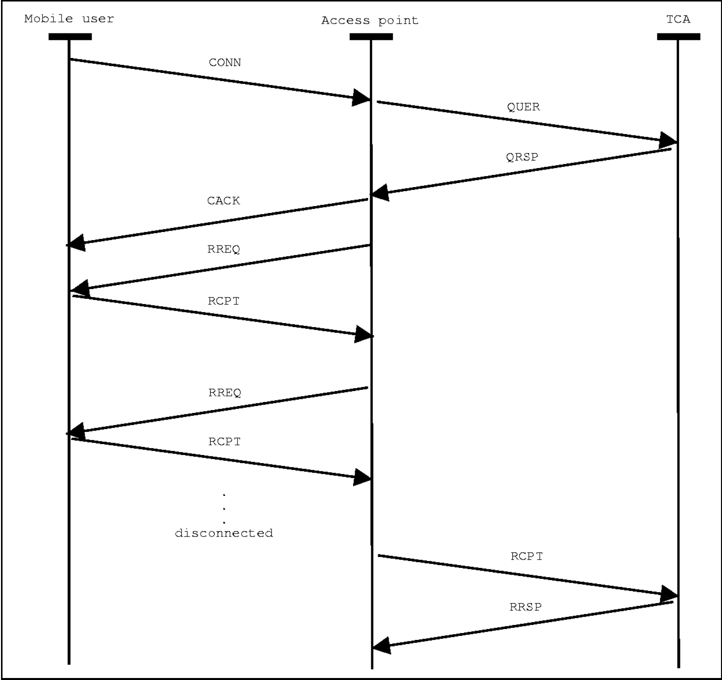

Figure 3.6: Message sequence in the centralized operation mode

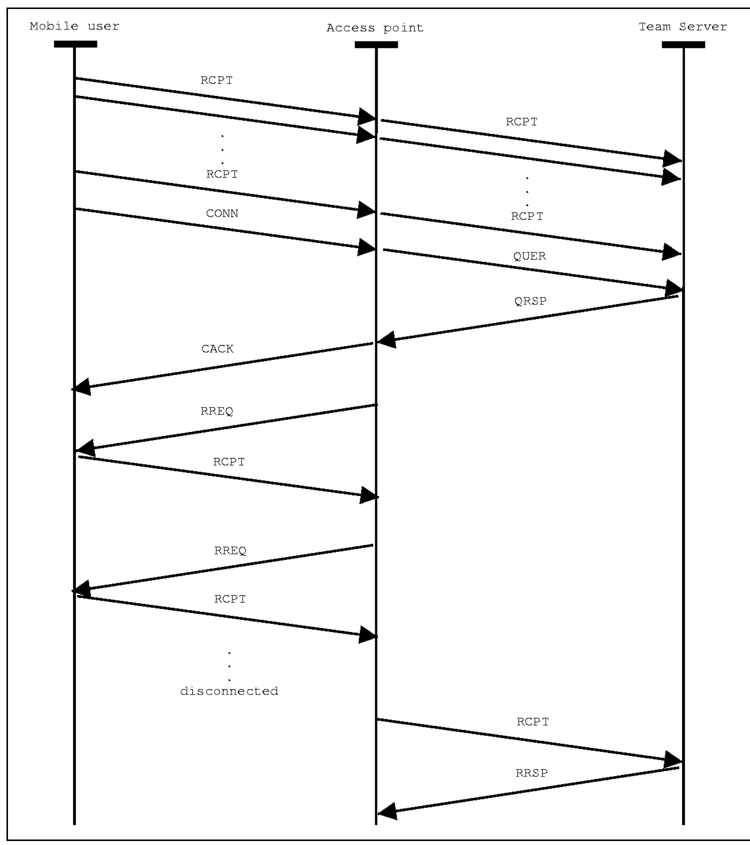

Figure 3.7: Message sequence in the decentralized operation mode

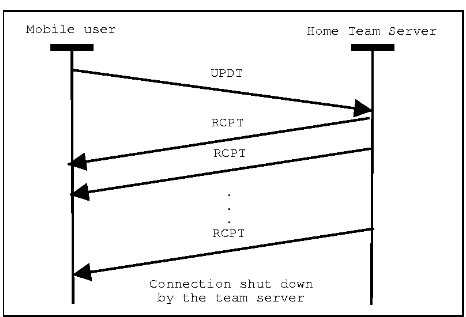

Figure 3.8: Receipt repository update operation

4 Reference Implementation

In this chapter, a reference implementation of the P2PWNC protocol that was specified in Section 3 is presented.

4.1 Generic Modules

4.1.1 Protocol Module

The protocol module is actually the implementation of the core of the P2PWNC protocol. Namely, it contains the basic data structures used by all system entities and the definitions of protocol messages. What is more, it contains routines for generating and parsing protocol messages and routines for converting data from their internal representation to other formats and vice versa. Also, input and output operations such as reading and writing to permanent storage or transmitting and receiving messages over the network are implemented here.

This module is shared by all system agents and is designed in a way that it depends as little as possible on third party implementors. Therefore, it does not include functionality for cryptographic operations nor does it depend on a specific thread library implementation.

4.1.1.1 Data Structures

Since one of the design goals of the core protocol module was generality, portability and independence from third party software vendors, it was convenient to represent data such as public and private keys, certificates and digital signatures in a generic internal format. This internal representation is an abstraction that is not bound on a specific implementation of a cryptography library. It is merely an interface between software modules that may have been developed independently and need to interoperate. One of the advantages of this approach is that one may migrate to another third party library to perform specific tasks

(such as cryptographic operations) only by providing routines for the conversion from the third party data representation to the internal P2PWNC format. What is more, the internal P2PWNC representation is more comprehensive in terms of the protocol, since its data structures do not include unnecessary information and are close to the logical view that one has of the data that they represent. In the following, a detailed description of these structures will be given, making use of C structs.

4.1.1.1.1 Key and Signature Format The internal format of public and private keys, as well as signatures is the same. The information necessary to describe such an entity is summarized in the following structure:

```
typedef struct pwnc_key {
        unsigned short algorithm;
        unsigned short bits;
        unsigned int datalen;
        unsigned char *data;
} PWNC_PUBKEY, PWNC_PRKEY, PWNC_SIG;
```

The *algorithm* field indicates the cryptosystem used. It may take one of two possible values for RSA and Elliptic Curve cryptography respectively. The *bits* field shows the bit length of the keys used. That is, if using RSA cryptography with 1024 bit keys, this field has a value of *24*. The actual key or signature data are stored in the *data* byte array whose length is *datalen* bytes. The content of the *data* byte array is specified in detail in Sections 3.3.1 and 3.3.2.

4.1.1.1.2 Client Certificate Format A client certificate is represented internally by the following C struct:

```
typedef struct pwnc_certificate {
        PWNC_PUBKEY *TeamPubKey;
        PWNC_PUBKEY *UserPubKey;
        PWNC_SIG *TeamSig;
} PWNC_CERT;
```

As one can see, a certificate is defined making use of the structure defined in the previous paragraph. Note that, although the members of the *pwnc_certificate* struct seem to be of different types, they actually have the same structure, since they all *pwnc_key*s.

4.1.1.1.3 Receipt Format The C struct that represents a receipt in the P2PWNC protocol reference implementation makes use of the structs defined for keys, signatures and certificates. It is shown below:

```
typedef struct pwnc_receipt {
        PWNC_CERT *ConsumerCert;
        PWNC_PUBKEY *ProviderPubKey;
        unsigned long timestamp;
        unsigned long weight;
        PWNC_SIG *RcptSig;
} PWNC_RCPT;
```

As their names imply, the *ConsumerCert* field represents the certificate of the service consumer, the *ProviderPubKey* is the public key of the service provider, the *timestamp* indicates the exact time that the session started and the *weight* field is the amount of traffic forwarded by the access point during a session. Finally, the *RcptSig* represents the receipt signature, produced by the service consumer. For details on the semantics of the above fields and a high-level description of the routine used for generating and signing a receipt see Section 3.3.4.

4.1.1.2 Conversion Routines

Since the internal representation of data is different than the format used when data are transmitted across a network, there is a need for routines that convert data from one format to another. Input to these routines are objects represented as described previously (e.g. *pwnc_key* or *pwnc_rcpt* objects) and their output are text messages. Conversely, there are routines that take text messages as input and produce internal structs. Therefore, there are two layers involved in the above description.

Figure 4.1 shows the high level view of the procedures taking place when sending and receiving a P2PWNC protocol message.

4.1.1.3 Input/Output Operations

This module is responsible for I/O operations performed by the P2PWNC software modules. Such operations include permanently storing and retrieving data and sending and receiving data from network. As of this section, two data formats have been defined; the

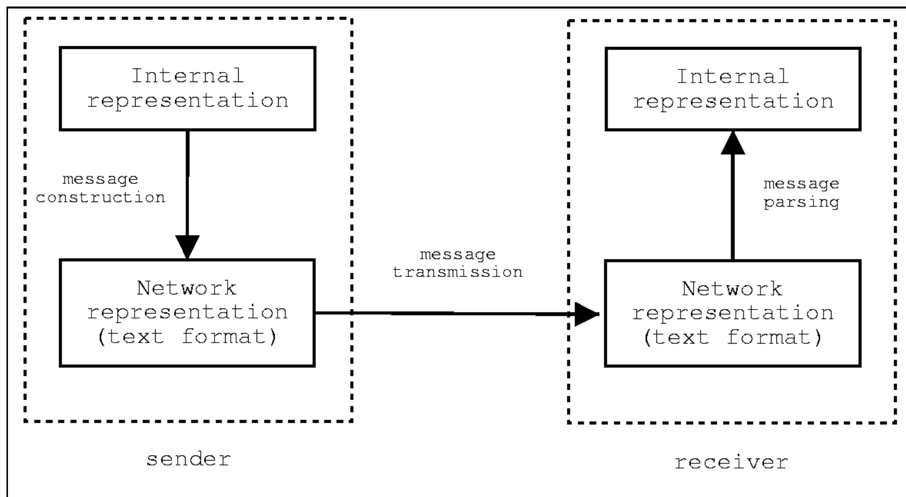

Figure 4.1: Message conversions

network format, which is specified by the P2PWNC protocol and the internal data representation. Now a compact format designed for storing data permanently is introduced.

4.1.1.3.1 Compact Data Representation This format is implementation dependent. That is, implementors may choose a storage scheme that best suits their needs. Since our software is aimed at memory and storage constraint devices, the main purpose of this design is compactness. To achieve compactness, storing data in text format is out of the question. Rather, data are stored in raw format, as plain byte arrays with the addition of some extra fields indicating properties of the data such as the length of the above arrays.

The main information unit in the P2PWNC scheme is a receipt. The receipt repository thus stores such objects. In this paragraph, the way these data are stored on permanent storage devices (hard disks, flash cards) is specified. Figure 4.2 shows the structure of a receipt as stored using its compact representation.

The first byte of this structure, denoted by “A” in the figure, indicates the encryption scheme used. If its value is 0 then elliptic curve cryptography is used. If it is 1, then the RSA cryptosystem is employed. This field is followed by the bit length that characterizes the cryptosystem. For example, in the case of 1024 bit RSA keys, the value stored in this two-byte field is 1024. In a similar fashion, if an 160 bit elliptic curve is used, this value is 160. Following that, there is a field that shows the total length of this structure. Since the largest key length permitted by the P2PWNC protocol is 2048 bits, two bytes

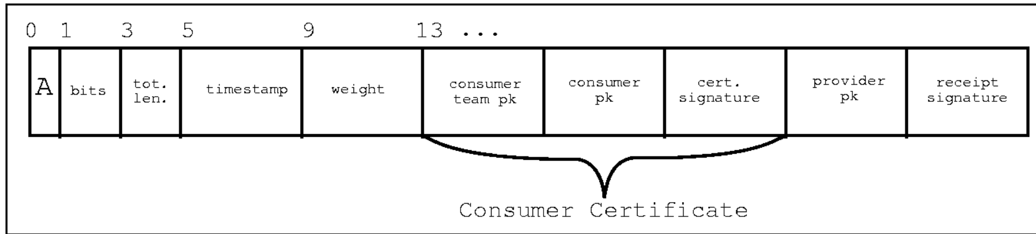

Figure 4.2: Compact representation of a receipt

are adequate to represent the length of the compact representation of a receipt, without the risk of overflowing. Next, there is the receipt timestamp for which four bytes are reserved. The next four bytes are the receipt weight, that is the amount of bytes forwarded by the access point on behalf of a client during a session. The remaining part of the structure is occupied by the client certificate, the public key of the service provider and, finally, the receipt signature. As stated in Section 3.2.2, the key algorithm used by a team member is the same as that of the key issuer. Thus, the fields of a receipt where the key algorithm and its bit length are specified provide enough information to calculate the length of the certificate fields as well as the length of the receipt signature. For example, if a receipt was generated by a client who uses 1024 bit RSA keys, one can safely conclude that the length of the certificate's team and user public keys, the certificate's signature as well as the receipt signature are all 128 bytes long. However, there is no information on the length and the algorithm associated with the service provider's public key. This is the reason why the *total length* field on the compact receipt representation is necessary; subtracting the length of the keys and signatures whose encryption algorithm is known, the receipt timestamp and weight, and the *bits* and *total length* fields yields the length of the provider public key in bytes.

Representing other structures such as public keys in compact format when storing them permanently is of little importance, since their volume is negligible for the system, compared to that of the receipts. Public and private keys and certificates may be represented either in a raw (compact) format or in plain ASCII text, with a header indicating the algorithm used (e.g ECC160), followed by a CRLF sequence, and the actual data, which are Base64-encoded.

4.1.1.3.2 Network I/O Receiving and transmitting messages over the network is the other task of this module. Receiving messages is performed in a unified manner for all

protocol messages. Having stated that all system messages have the same generic format (see Section 3.1.3.2), parsing is easy. The first two message headers indicate the type of the message (e.g. RCPT, UPDT) and the length of the data that follow (Content-length header). The routine that receives messages takes a socket identifier as input and returns the text message that was read, as well as its type so that it will be dispatched to message-specific parsing routines. As an example, the prototype of the message receiving routine used in our C implementation of the P2PWNC scheme is presented.

```
unsigned char* read_message(int sockfd, int *mtype);
```

The first argument of this function is the socket descriptor from which data will be received. After the function has returned the text message read, *mtype* will be set to the type of the received message. The text message that this function returns will then be passed to a message handling routine. For example, in case of an RCPT message, a routine that converts text RCPT messages to the internal receipt format (see Section 4.1.1.1) will be called.

On the other hand, when a message is to be sent, conversion routines (see Section 4.1.1.2) will be called so that internally-represented data will be transformed to or embedded into text messages. These text messages will then be written to a socket so as to be transmitted to a receiving entity.

4.1.2 Cryptographic Operations

Cryptographic operations are based on third party libraries. In our reference implementation we make use of the OpenSSL [7] library for the modules written in C and the Java Cryptography Extension (JCE) [3] packages for our Java-based clients. Regardless of the implementation of the cryptography related operations, associated structures such as public and private keys must be converted from the vendor-specific to the internal P2PWNC-specific representation that was described in Section 4.1.1.1.

The cryptographic operations module provides functionality for generating key pairs and P2PWNC certificates, signing and verifying certificates and receipts and converting from the internal P2PWNC format to vendor-specific formats. In this module, the procedures for generating certificates and receipts, as specified in Sections 3.3.3 and 3.3.4, are implemented.

4.1.3 Multithreading

In the P2PWNC protocol reference implementation presented here, most software modules make use of multithreading. For example, the access point module starts a new thread for each client session and the team server module has a separate thread for each request. A major advantage of using multiple threads over spawning new process images for each request is that there is less memory overhead, since different threads share the same resources. This is crucial for modules such as that of the team server, where all threads must have a view of the receipt repository; creating a child process (via the fork system call, for example) for each new request would require the memory context of the parent process to be copied, thus resulting in huge memory squandering. Also, a new process created via the fork system call is more time-expensive due to costly context switching.

In the C implementation of the P2PWNC system, that runs under Linux, the POSIX threads (libpthread) package is used. A variant of the *boss/worker* design pattern has been developed. This model is described next.

The main thread(boss) of the multithreaded server is responsible for dispatching client requests to handling threads(workers). Each thread is bound to a TCP connection. Thus, each time a new TCP connection is accepted by the server, a new thread must be started to handle the client request. Such threads act independently, apart from the fact that they may share some resources. For example, in the TCA module, the client threads share the receipt repository. Shared resources are protected by mutices, which are defined by the application that uses the thread module. When the client session is to end, the worker thread must exit.

Since resources are limited, especially in the case of software modules that run on embedded platforms (e.g. access points), and as a means of avoiding denial of service attacks, the main thread must control the number of client threads that are currently operating. Thus, there is a thread pool of fixed length from which a new client thread is picked in the event of a new session. If all the threads of the pool are occupied, then the session is rejected. When a session is terminated, then the thread responsible for that session is returned to the pool so that it can be used to serve future client requests.

Each element of the thread pool is described by the following C struct:

```
typedef struct _client_thread_t {
        pthread_t client_thread;
        int is_free;
} CTHREAD;
```

The *client_thread* field is the actual thread identifier and the *is_free* field indicates whether this thread is in use for a client session (value 0 if it is occupied). Whenever a new TCP connection is accepted, the thread pool is searched linearly so that the first free thread is located. Linear search is expensive. In practice, though, the size of the thread pool is expected to be small (that is, a few hundred elements at most), so the overhead imposed is negligible. As soon as the free thread is located, the new thread is started. In the *pthreads* package, a new thread is created via the following call:

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
        void *(*start_routine)(void *), void *arg);
```

The *attr* pointer in the above definition is ignored. The *thread* field represents the thread identifier. The *start_routine* argument is a pointer to the function that the thread will execute. This function is defined by the application. Finally, the *arg* field represents the arguments that are passed to *start_routine* as soon as the new thread has started.

In our case, when it comes to one of the threads of the thread pool, this function is never called directly. Instead, the *start_new_thread* routine has been implemented. Its definition is as follows:

```
void start_new_thread(void *thread_func, int sockfd);
```

The *thread_func* argument is the client handling thread function that will be passed as a parameter to *pthread_create*. *sockfd* is the TCP socket over which the multithreaded server communicates with a client.

Since the thread pool is shared among many threads, there is a mutex that controls access to it. Its definition is shown below:

```
static pthread_mutex_t thread_pool_mutex;
```

When the "boss" thread is about to start a new "worker," it attempts to lock this mutex. After picking up a thread from the pool and starting it (or after failure to do so due to lack of free threads), the mutex is unlocked. In a similar fashion, when a thread is about to exit, the mutex is acquired by the exiting thread, the element of the pool is marked as *free* (setting *is_free* to 1) and the mutex is released.

One subtlety that has not yet been discussed is how a running thread knows its position in the thread pool array and the socket descriptor of the connection that it serves. This problem is solved via the argument that is passed to the thread handling function (*thread_func*) discussed above. The generic prototype of such a function is as follows:

```
void thread_func(THREAD_INFO *ti);
```

The *THREAD_INFO* data type is defined as follows:

```
typedef struct _client_thread_info_t {
        int client_socket;
        int thread_index;
} THREAD_INFO;
```

The *client_socket* field is the socket that corresponds to the client session handled by the thread and *thread_index* is the index of the specific thread in the pool array. The *ti* argument of *thread_func* is generated inside the *start_new_thread* function, using the *sockfd* argument that is passed and searching through the thread pool array to find a free position.

4.2 Receipt Repository

4.2.1 The Role of the Module

The role of the receipt repository module is to store receipts, answer queries as to whether a client requesting service should eventually be admitted in a visited network and, in some cases, to answer range queries with the receipt timestamp as the search criterion.

Receipts may be stored by various system entities. Starting with the centralized case, receipts need only be stored at a central repository by the TCA. Clients and access points need not carry any receipts; newly generated receipts are automatically send to the TCA. In the decentralized case, on the other hand, there is no global central receipt repository. For each team there is a team server, the (logical) entity that is responsible for receipt storage. Also, there must be a storage module in the client software, since mobile users tend to carry receipts with them to present them to the visited access points at the beginning of a session. It should also be noted that a team server may be collocated with one of the access points of a team.

The core functionality of the receipt repository is the same for both the centralized and the decentralized case. The exception to this rule is that in the decentralized case it is necessary that the repository server should be capable of answering UPDT queries. Also, the storage module on the client side may be more lightweight, since it need not be equipped with decision function capabilities.

4.2.2 Supported Operations

From the above discussion it occurs that there is a need for a versatile data structure that should be able to perform the following operations efficiently:

- Insertion
  Receipt insertions will be taking place quite often, especially in case the receipt repository is global. Thus, the data structures employed should provide for fast insertion.

- Deletion
  Receipt deletion will be taking place mainly during receipt replacement. That is, when the receipt repository is full, a new receipt will replace the one with the oldest timestamp, since the receipt timestamp is a criterion for deciding on receipt validity; the older a receipt, the less valid it is for the P2PWNC system.

- Search
  Searching for a receipt using its timestamp as the key of the search is necessary when attempting to locate the oldest receipt of the repository or when trying to answer an UPDT query.

- Receipt lookup
  Receipt lookups are performed when a receipt is about to be inserted in the repository to check if a receipt already exists in the database.

- Team lookup
  Looking up a team is mainly performed when there is a query on whether access should be granted to a client. It is necessary that the identifiers of the two teams involved in a transaction (provider and consumer public keys) are located so that they will be passed as input to the decision function.

- Receipt range queries
  Such queries may be issued to the receipt storage module by a team server in case an UPDT client message has been received. As has already been specified (see Section 3.1.3.4), an UPDT message contains a timestamp field. This will be used in the range query. Namely, the receipts with timestamps greater or equal to it will be the query output.

- Graph operations
  Finally, apart from the above operations, the data structures representing the receipt repository should be capable of performing graph operations. This repository can be represented by a weighted directed graph, whose edges are receipts and vertices are teams. Graph operations are performed during the execution of the decision function, which is based on discovering the maximum flow between two graph nodes.

4.2.3 Module Architecture

The operations that should be supported by the receipt repository module, as mentioned in the previous paragraph, suggest that a composite data structure should be used to satisfy their diverse needs as far as efficiency is concerned. Graph operations need a representation of the graph by a structure such as an adjacency list or an adjacency matrix. Such a data structure on its own, though, would not be appropriate to search for a receipt using its timestamp as a key or performing other dictionary operations.

The approach that was adopted is to build a composite data structure that includes hash tables that store pointers to the teams and receipts, an adjacency list representation of the receipt graph and a red-black tree [23] where each node has a pointer to a receipt and node keys are receipt timestamps. These data structures combined provide an efficient means of carrying out the necessary receipt repository operations. Figure 4.3 shows the architecture of the receipt database module.

4.2.3.1 Team and Receipt Hash Tables

As one can see, there are two hash tables. One that stores pointers to structures that represent system teams and one that contains pointers to receipt structures. These hash tables are used to speed up receipt and team lookup. Specifically, insertion and lookup take constant time. The C struct representing a receipt (*PWNC_RCPT*) is shown in Section 4.1.1.1.3. Team information are kept in a struct shown below:

```
typedef struct __pwnc_team {
        PWNC_PUBKEY *team_pk;
        int served;
        int consumed;
        double credits;
        AL_HEAD *graph_node;
```

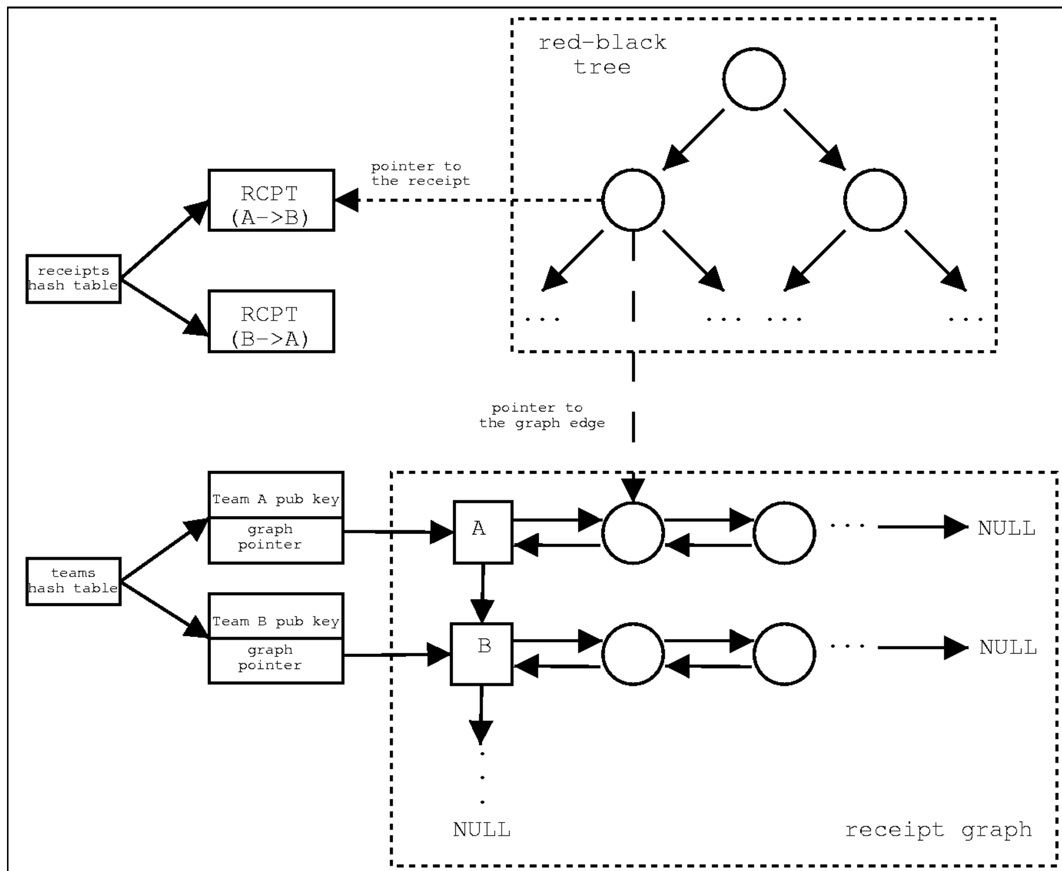

Figure 4.3: Receipt repository architecture

```
}PWNC_TEAM;
```

The `team_pk` field, as its name implies, is a pointer to the team's public key, represented in its internal format. It should be noted again that a team's unique identifier for the P2PWNC system is its public key. The fields `served` and `consumed` show the number of times access points of this team have provided service and the number of times that its members have consumed service respectively. Also, `credits` contains the total amount of service a team has provided, measured in kilobytes. These fields may useful for some decision functions, but may also be ignored by the application. Additional fields may also be added to this struct, in case a new decision function that is to be built makes use of them. Finally, `graph_node` is a pointer to a head list node of the receipt graph. The head list is a singly connected list of nodes, each of whom represents a (consumer) team in the graph. In Figure 4.3, this list is denoted by the singly-connected labeled squares in the receipt graph. The

details of the graph representation will be explained in Section 4.2.3.4.

Hash tables provide for $O(1)$-time insertion, deletion and search operations. *lhash*, the dynamic hash table implementation included in the OpenSSL package is used. The application that makes use of this library needs to define the hash function that wishes to be used for a hash table. Therefore, in the receipt repository module, different hash functions have been defined for the receipts and teams hash tables, since the two tables store pointers to different structures.

In the implementation used, hash functions take a pointer to a structure as input and map it onto a 32 bit integer number, based on the structure's key. This key is the identifier of the structure. For example, the identifier of a *PWNC_TEAM* structure is the team's public key, that is the `team_pk->data` field. Note that the key field is not defined anywhere explicitly. Rather, the user defined hash function is responsible for discerning which fields of the struct will be treated as its unique identifier. How the mapping of the input struct on the 32 bit integer space will be performed is application dependent. Care should be taken, though, since the mapping over the 32-bit integers should be as uniform as possible. For example, for the case of *PWNC_TEAM* structs, one may choose the first four bytes of the `team_pk->data` field, as public keys tend to be quite random arrays of bytes.

In the case of `PWNC_RCPT` structures, the unique identifier of a receipt is the following tuple:

$$\langle \text{ Consumer certificate, Provider public key, Receipt timestamp} \rangle$$

The hash function here operates in a similar manner. Since during a client session more that one receipts with the same key may be generated, only the last one of them is valid, that is the one with the highest "weight" value (represents the last receipt sent by the client, which is the one that characterizes the session).

More information on the *lhash* library can be found in the *lhash* Linux manual pages or in the OpenSSL online documentation [7].

4.2.3.2 Red-black Trees

4.2.3.2.1 Properties Red-black trees [23] are a special case of balanced binary search trees. Each node contains a field that indicates its color, which can either be red or black. A binary search tree is a red-black tree if it satisties the following properties:

1. Every node is either red or black

2. The root is black
3. Every leaf is black
4. If a node is red, then both its children are black
5. All paths from a node to descendant leaves contain the same number of black nodes

Each red-black tree node has pointers to its parent, its left child and its right child. In case a node does not have a parent or a child, the respective pointers point to a special *NULL* node whose color is black, so that property 3 is not violated. The parent, left and right pointers of the *NULL* node are ignored and can be set to arbitrary values.

It can be shown that the height of a red-black tree is at most $2\log(n+1)$. That is, such a tree is approximately balanced. Insertion, deletion and search for a node are thus performed in $O(\log n)$ time.

4.2.3.2.2 Applicability In this section the applicability of the use of red-black trees for receipt storage will be argued. First, red-black trees offer worst case logarithmic search, insertion and deletion time, as opposed to splay trees [26], where the logarithmic time bounds stem from an amortized analysis. Also, splay trees are not efficient when it comes to accessing tree elements in sorted order. This operation is performed, for example, in the case of an UPDT message where all nodes (receipts) with key (timestamp) greater or equal to a specified value are returned. Since each time an item is accessed it is moved to the root of the tree, re-accessing the first retrieved element of the sorted sequence would require $O(n)$ time.

Second, receipts are all stored in main memory, since they need to form a receipt graph, upon which a maximum flow algorithm is run. Thus, a red-black tree is more suitable than a B-tree structure, since the latter is optimized for minimizing disk accesses and is designed for large data sets residing in secondary storage.

A normal binary search tree is also out of the question. The nature of the data stored in the tree, that is receipts, implies that, as soon as the receipt database is full, older receipts will be replaced by new ones. Since the key of the tree nodes is the receipt timestamp, receipts from the left end of the tree will be removed and new receipts will be added to the right end of the tree. Therefore, the tree will gradually degenerate to a linked list, resulting in $O(n)$ runnning time for search, insert and delete operations.

A data structure whose applicability is comparable to that of red-black trees are AVL trees [12]. AVL trees are balanced binary search trees. For each node x, the heights of the

left and right subtrees of x differ by at most 1. Height balancing is stricter that in red-black trees, but rebalancing after an insertion and deletion is more costly (albeit still logarithmic).

Red-black trees are also more suitable for the needs of the receipt repository module due to the fact that they can easily be converted to a *persistent* data structure. Such data structures allow for versioning of the instances of data; that is, past versions of the data set can be maintained as the set is being updated. This may be important for the P2PWNC receipt repository in case there is a need to store receipts temporarily, perform operations on the updated data set (e.g. execute the decision function) and, finally, restore the repository to its previous state (i.e. before inserting the temporary receipts).

4.2.3.2.3 Implementing Red-Black Trees For the needs of the P2PWNC software, a C implementation of red-black trees is supplied. This implementation is as generic as possible and thus independent of the actual type of the data that are stored in it. A tree node is represented by the following C struct:

```
typedef struct rbtree_node_t {
        void *data;
        struct rbtree_node_t *left, *right, *parent;
        unsigned short color;
        void* (*key_func)(struct rbtree_node_t *);
        int (*comp_func)(
                const struct rbtree_node_t *,
                const struct rbtree_node_t *);
        void (*set_key_func)(struct rbtree_node_t *, void *);
} RBTNODE;
```

As one can see, the data stored in a tree node are represented by the `data` pointer, whose type is not specified. The `left`, `right` and `parent` pointers are self explanatory, and so is the `color` member. The latter, can take one of the values `NODE_COLOR_RED` or `NODE_COLOR_BLACK` (constants).

The three function pointers are callbacks that have to be defined by the application. Their roles are explained below:

`key_func` This function takes a tree node as an argument and returns the node's key. The calling function must cast the returned value of `key_func` to the appropriate type. For example, in a tree that contains receipts which are keyed by their timestamp, this function returns the timestamp of the receipt. `key_func` may be called as follows:

```
RBTNODE *node;
...
unsigned long ts = (unsigned long)node->key_func(node);
```

It is supposed that `node` has already been initialized and that the `key_func` callback has been implemented by the application so that it returns the timestamp of the receipt included in the node's data.

`comp_func` This callback compares two nodes according to their keys. It is used when performing tree operations, such as search, insertion or deletion. It returns -1, 0 or 1 if the first argument's key is less than, equal or greater than that of the second argument respectively. `comp_func` normally makes use of `key_func`.

`set_key_func` As its name implies, `set_key_func` sets the key of a node (first argument) to the specified value(second argument). The type of the key value is not specified in the above definition. The application where this callback is implemented needs to perform the necessary type castings.

The above callbacks need to be specified for every tree node and should be the same for all these nodes. It is convenient for application developers that wish to use this red-black tree implementation to create a routine that allocates memory and initializes a tree node, passing it pointers to the implemented callbacks, among other arguments. An example of the prototype of such a function is shown in the following example:

```
RBTNODE *RBTNODE_new(
        PWNC_RCPT *rcpt,
        const void *key_func,
        const void *set_key_func,
        const void *comp_func);
```

The application that makes use of the red-black tree library does not need to know the details of the implementation. It is adequate to implement the above callbacks and use the following API calls:

- **`void rbtree_insert(RBTNODE **tree, RBTNODE *z)`**
  This function inserts node z to the tree. It also handles tree rebalancing.

- **RBTNODE *rbtree_delete(RBTNODE **tree, RBTNODE *z)**
  This function removes node `z` from tree and returns a pointer to it. `z` may have been returned by a prior call to `rbtree_find_min` or `rbtree_find_node`.

- **RBTNODE *rbtree_find_min(RBTNODE *tree)**
  Returns a pointer to the node with the minimum key in the tree. In fact, this function starts from the root of the tree and follows the left pointers. The leftmost node of a red-black tree is also the minimum.

- **RBTNODE *rbtree_find_node(RBTNODE *tree, RBTNODE *node)**
  Finds and returns a pointer to a tree node whose key value is equal to `node`'s key. `node` is an artificial node, constructed by the calling function and need only have its key, key_func and comp_func set.

- **void rbtree_range_query(**
  ```
          RBTNODE* tree,
          void (*do_stuff)(RBTNODE*, void *),
          void *do_stuff_args,
          RBTNODE *node)
  ```
  This routine performs a variant of a range query on the tree nodes. In particular, nodes with keys greater than or equal to the key value of `node` are processed. `rbtree_range_query` visits the appropriate nodes in an in-order fashion. However, it does not visit all of them; rather, it is called recursively for the left subtree of the current node only in case its key value is greater of equal to `node`'s. For each visited node, `do_stuff` is called and it is passed `do_stuff_args` as an argument. The `do_stuff` callback and its argument are application dependent. Its time complexity is $O(n)$, where n is the number of tree nodes, which is asymptotically equal to that of a normal in-order tree traversal. However, as far as the P2PWNC software is concerned, a red-black tree is used for receipt storage and such queries will normally request receipts with recent timestamps, which reside near the rigthmost end of the tree. Thus, visiting only the necessary nodes will result in important performance gains.

4.2.3.3 Permanent Receipt Storage

Apart from residing in the system's main memory, receipts may also be put in secondary storage (e.g. hard disks or flash cards). Thus, each time a new receipt arrives, it is inserted

in the hash table, in the red-black tree, in the receipt graph and in permanent storage. As has been stated in Section 4.2.3.2, insertion takes time logarithmic on the number of the tree nodes. To take full advantage of that property, we need to be able to write the receipt on disk in constant time. That is, the position where the data will be written permanently must be located as fast as possible. In the following, the way data are stored is explained.

Receipts are written in a file sequentially. Each record in this file has fixed length, equal to the maximum size of a receipt in its compact format (see Section 4.1.1.3.1) that is permitted by the protocol (such a receipt uses 2048 bit RSA keys). When a receipt is removed from the tree, the position it occupied in the receipts file is marked as deleted so that it can be overwritten by a new receipt. The positions (slots) that are free (as a result of receipt deletions) are stored in another file. This file stores a stack of free slots. In the event of a new receipt insertion, a free slot is popped from that stack and the new receipt is written there. Conversely, when removing a receipt from the repository, the slot that it occupied is released and is pushed back in the stack.

Locating the slot where a receipt is written is performed as follows. In the `data` field of a tree node a structure that is shown below is stored:

```
typedef struct rbtree_rcpt_data_t {
        PWNC_RCPT *rcpt;
        AL_NODE *graph_entry;
        int disk_pos;
} RBTREE_RCPT;
```

The `rcpt` field is a pointer to the actual receipt. `graph_entry` points to the receipt graph edge that corresponds to this receipt, as explained in Section 4.2.3.4. Finally, `disk_pos` is the position of the receipt record in the file. Note that `disk_pos` is measured in records rather that in bytes. Therefore, to access the receipt as stored in the file, one has to calculate its exact position multiplying `disk_pos` by the maximum receipt size. Then, a transition to that position is performed via a call to the `fseek` function.

4.2.3.4 Receipt Graph

Having stated that there needs to be a graph representation of the receipt repository, the graph data structure used for this purpose is to be described in this section.

To represent the graph an adjacency list data structure has been developed. It is composed of a singly-linked list of "head" nodes, each of them representing a P2PWNC consumer team. Nodes of this list have the following format:

```
struct adjacency_list_head_t {
        int label;
        int color;
        int height;
        int discharged;
        long excess;

        struct adjacency_list_rcpt_node_t *neighbors;
        struct adjacency_list_rcpt_node_t *last_neighbor;
        struct adjacency_list_head_t *next_head;

        unsigned short node_state;
        struct incoming_edge_list_node_t *incoming;
        struct incoming_edge_list_node_t *last_incoming;

        struct __pwnc_team *team;
};
typedef struct adjacency_list_head_t AL_HEAD;
```

The fields `label`, `color`, `height`, `discharged` and `excess` have to do with the maximum flow algorithm used by the decision function. The same apply for the `node_state`, `incoming` and `last_incoming` fields, which are used but the global relabeling maximum flow algorithm heuristic (see Section 4.2.3.5).

Each `AL_HEAD` represents a graph node. Graph edges correspond to receipts and are directed from the consumer to the service provider node. For each node of the head list, there is a doubly connected list of the service providers to whom the head node owes service. This doubly connected list represents the edges that are directed from this graph node to others and is accessed via the `neighbors` pointer, which is its first node. Each of its nodes represents a receipt which has this `AL_HEAD` node as the consumer and some other `AL_HEAD` as the provider. The provider is represented by a field of the neighbor list node, whose stucture will be shown next. The `next_head` member points to the next element of the head list. Finally, `team` is a pointer to a `PWNC_TEAM` structure, as described in Section 4.2.3.1.

It should be noted that a graph edge may correspond to more that one receipts, since all receipts from a team A to a team B are merged to one edge, whose weight is the sum of all

$A \rightarrow B$ receipts. This is done to avoid representing the receipt graph as a multigraph, and thus achieve the desired time complexity of the graph algorithms used.

The structure of the graph edges is shown below:

```
struct adjacency_list_rcpt_node_t {
        struct adjacency_list_rcpt_node_t *next;
        struct adjacency_list_rcpt_node_t *prev;
        struct adjacency_list_head_t *myhead;
        struct adjacency_list_head_t *consumer;
        int saturated;
        long weight;
        long residual_weight;
        int is_foo;
        struct incoming_edge_list_node_t *incoming_ptr;
};
typedef struct adjacency_list_rcpt_node_t AL_NODE;
```

Since it is a doubly-connected list node, `next` and `prev` pointers can be used to traverse it in both directions. The source and the destination of an edge are represented by `AL_HEAD` objects. Thus, the `myhead` member points to the service provider (destination of the edge) and `consumer` points to the edge source. In fact, `consumer` points to the head node to which this edge list belongs. The `weight` field, as its name implies, shows the edge weight. The `saturated`, `residual_weight` and `is_foo` fields are used by the maximum flow algorithm. The meaning of `incoming_ptr` will be explained next. Figure 4.4 depicts the adjacency list receipt graph representation.

For the needs of the maximum flow algorithm, a list of the incoming edges of each node should be kept, since it is required for the backwards BFS execution in the global relabeling heuristic (see Section 4.2.3.5). For each graph node, there is a doubly-linked list of its incoming edges. The C struct that follows represents a node of this list:

```
typedef struct incoming_edge_list_node_t {
        AL_NODE *node;
        struct incoming_edge_list_node_t *next;
        struct incoming_edge_list_node_t *prev;
} INCOMING;
```

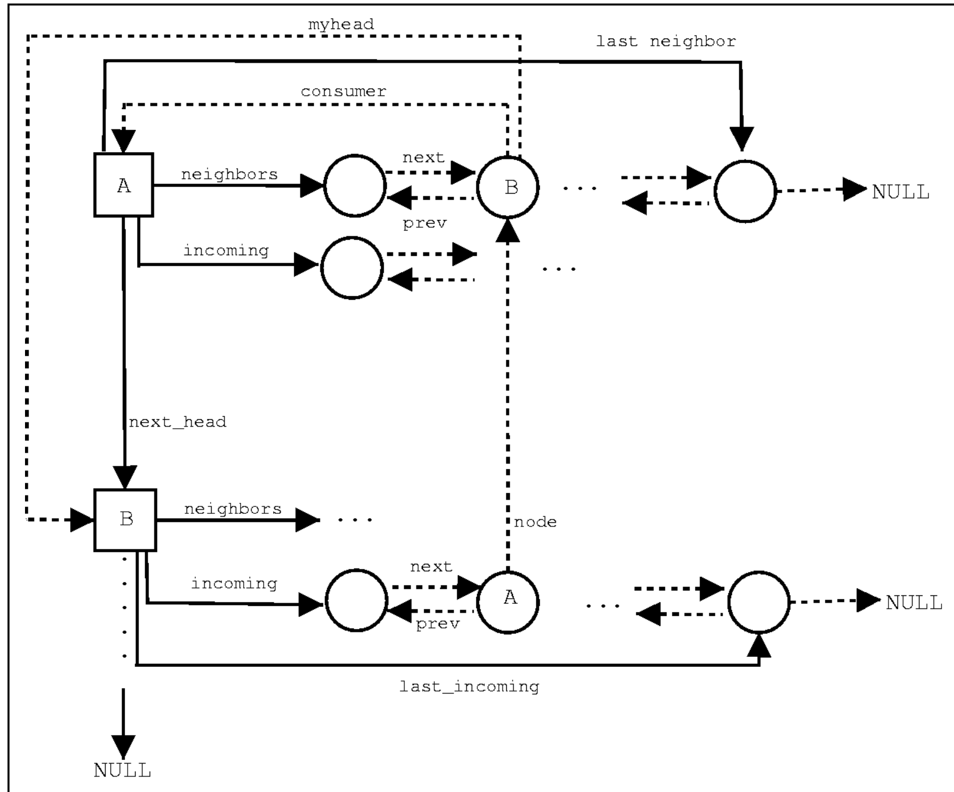

Figure 4.4: Adjacency list representation of receipt graph

The `node` field is a pointer to the edge that this list node represents. In a similar manner, `node->incoming_ptr` points to this list node. Figure 4.4 may clarify this description.

4.2.3.5 Maximum Flow Algorithm Implementation

4.2.3.5.1 Terminology The graph data structure described in Section 4.2.3.4 is used for running a maximum flow algorithm, which is the core of the basic family of decision functions proposed in the P2PWNC scheme. For the needs of these functions, a version of the push-relabel maximum flow algorithm has been implemented. In particular, a first-in-first-out variant of the generic push-relabel algorithm [22] is being used. To improve the efficiency of this algorithm the *global relabeling* heuristic has been applied [14]. In the remainder of this section, the operation of this maximum flow algorithm, as implemented in terms of the P2PWNC software, is described. A performance evaluation of the implementation presented here follows in Section 5.4.

The maximum flow algorithm takes as input two vertices of the graph and computes

the maximum amount of flow that can be pushed along the edges of the graph from the source to the sink vertex, without violating the capacity constraints of the edges. Namely, the maximum amount of flow that can be pushed along an edge is the edge's weight.

In push-relabel algorithms, for each vertex there is a *height* function, which is in fact an estimate of the distance of this vertex from the destination. Each node also maintains an amount of flow that is to be pushed to neighboring nodes via its outgoing edges. This amount is called the *excess flow* of a vertex and the respective node is called *overflowing*. The `height` and `excess` fields of the `AL_HEAD` structure represent the above. An edge through which no more flow can be pushed is called *saturated* (the respective field of the `AL_NODE` struct). When an edge is not filled to capacity, the amount of flow that it still admits is called the edge's *residual capacity* and is denoted by the `residual_weight` field of `AL_NODE`. When pushing x units of flow over the $u \to v$ edge, its residual capacity is decremented by x, while the residual capacity of $v \to u$ is incremented by x (in our implementation, if $v \to u$ does not exist, it is added to the graph and is marked as "foo"; this is the meaning of the `is_foo` field of `AL_NODE`). Such an operation results in a modified graph, called *residual graph*. For a strict and complete definition of these terms, see [22] and [14].

4.2.3.5.2 Basic Operations The basic actions of the maximum flow algorithm are pushing flow along an edge and relabeling a vertex. By relabeling, increasing the height of a vertex to the height of its lower neighbor plus one is meant. Pushes can be further categorized in saturating and non-saturating. As implied by their name, saturating pushes fill an edge to capacity. In our implementation, a saturated edge disappears from the residual graph (In fact, it does not disappear; rather, it is moved to the end of the list of outgoing edges of the respective head node and marked as saturated. After the maximum flow has been terminated, the graph is restored by marking the edge as non-saturated and setting its residual weight to the original edge weight). On the other hand, non-saturating pushes may cause a new edge to be added to the residual graph (this edge is marked as "foo" and is removed from the graph during the restoring procedure that was just mentioned).

4.2.3.5.3 The Push-Relabel Algorithm For the needs of the algorithm presented here, a FIFO queue is used for keeping the vertices that are overflowing, and will be referred to as *active* nodes. A short description of the operation of the algorithm will now be presented. In this description, the maximum flow from a node s to a node t of a graph $G = (V, E)$ is calculated, where V is the set of vertices and E is the set of edges of the graph.

Initialization involves setting the height of the source to $|V|$ and the height of all other nodes to 0. The algorithm starts by pushing as much flow as possible from the source to its neighboring nodes, thus filling its outgoing edges to capacity and saturating them, and generating an initial *preflow*. The newly-overflowing vertices are inserted in the queue of the active nodes.

The algorithm then continues by repeatedly removing an active node from the queue and pushing as much flow as possible to neighboring nodes via its outgoing edges. In this phase, new nodes may be pushed in the queue. When the vertex in consideration is no longer overflowing or no more flow can be pushed via non-saturated edges, the node is relabelled. This procedure is called vertex *discharging*.

The algorithm terminates when there are no more active nodes in the queue. The value of the *excess* of vertex t is the computed maximum flow from s to t.

4.2.3.5.4 The Global Relabeling Heuristic The above algorithm runs in $O(V^3)$ time. This upper bound, though polynomial, has proven poor for the needs of the P2PWNC system. Therefore, the global relabeling heuristic has been implemented. As to this heuristic, in the beginning of the execution of the maximum flow algorithm and periodically, the *height* values of the graph vertices need to be updated. Since, the *height* of a node is an estimate of its distance from the terminal node, a more accurate value is obtained via this method.

To update the height values, a backwards *Breadth First Search* is performed, starting from the terminal node. After BFS has been executed, the *height* fields of the vertices contain the length of the path from each node to t (measured in edges). In order to run a BFS backwards, the lists of incoming edges are used (see 4.2.3.4).

In the P2PWNC reference implementation, the above procedure takes place every V relabel operations. In comparison to the push and relabel operation, BFS is an expensive task, since it requires $\Theta(V + E)$ time. However, as it seems, it is performed rather rarely. According to our experiments (see 5.4), as well as related work [14], the performance of the maximum flow algorithm is dramatically improved by this heuristic.

4.2.4 Decision Algorithms

4.2.4.1 Decision Algorithm as a Pluggable Module

In the event of a roamer visiting a foreign access point, the potential provider team needs to decide whether or not the visitor will be granted internet access. In order to come to this decision, the access point will normally consult the receipt repository module, since the information on which the decision will be based reside in it.

In the centralized case, the module responsible for making suggestions to the access point on the decision to take is the Trusted Central Authority. In the decentralized one, each team may follow the advice of its team server.

Since the P2PWNC is a framework that provides agents with independency concerning such decisions, the access point can choose its own access provision algorithm. An important family of decision algorithms is based on the maximum flow algorithm presented in Section 4.2.3.5. Work that is in progress [19] [18] suggests that this family of functions supplies agents with the right incentives that promote cooperation between them.

As to the P2PWNC reference implementation, care was taken so that it will be easy for third party developers to implement their own decision algorithms and embed them to the existing scheme. The input to every decision function should be the identifiers of the provider and the consumer team. In the case of this scheme, teams are identified by their public keys. The prototype of the decision function is shown below:

```
int rdb_judge(
        PWNC_PUBKEY *consumer, PWNC_PUBKEY *provider)
```

This routine implements the decision algorithm and returns 1 to recommend that access should be granted and 0 otherwise. As it seems, the calling applications need only be aware of the function's interface. It is supposed that the generic data types such as `PWNC_PUBKEY` are implemented by all protocol entities.

It is also supposed that the necessary functionality needed by the decision algorithm is implemented by the decision function developer. This includes any underlying data structures used by the decision function. For example, when applying the maximum flow algorithm to decide if access should be granted, data structures to represent the receipt graph must be provided.

In the following, the implementation of two decision functions will be presented. First, a simple score-based algorithm will be shown and, second, a more sophisticated method based on the maximum flow algorithm presented in Section 4.2.3.5 will be described.

4.2.4.2 Score-based Decisions

In this section, an access algorithm based on service provision measurements is presented. Since access points measure traffic forwarded on behalf of roamers, the total amount of service a tream has provided is the sum of kilobytes that the team's access points have forwarded. This sum is used as the team's *score* (which from now on can be referred to as a team's *credits*).

In the event of a client visiting a foreign access point, the decision whether access should be granted to the client is based on the ratio of the score of the consumer team to the score of the team providing service. Access is granted with probability equal to that ratio (or 1, if the ratio is greater that 1). Note that in case the provider's score is equal to zero, access should be granted. This implies that a client whose team score is greater to that of the provider team will certainly be granted access. The above are summarized by the following formula.

$$P(access) = \begin{cases} 1 & \text{if } credits(Pr) = 0 \\ \min\{1, \dfrac{credits(C)}{credits(Pr)}\} & \text{otherwise} \end{cases} \tag{4.1}$$

In the above, the provider and consumer teams are denoted by Pr and C respectively, $P(access)$ is the probability that the client will be granted access. $credits(Pr)$ and $credits(C)$ are the scores of the provider and the consumer teams respectively, measured in kilobytes.

The implementation of the above decision function is rather straightforward. In our C implementation, the `credits` field of the `PWNC_TEAM` struct (see 4.2.3.1) is used for the representation of a team's score. Each time a new receipt is added, the score of a team is adjusted accordingly. When the `rdb_judge` function is called, a fast lookup is performed on the hash table storing the system's teams so that pointers to the provider and the consumer teams taking part in a (potential) session are retrieved. The function returns success (1) or failure (0) with probability $P(access)$, computed according to the above formula.

The above decision function behaves well in peer populations composed of peers that adhere to this function, malicious peers that refuse to grant access when they ought to and others who provide service in an altruistic manner. However, this decision algorithm is not collusion-proof; peers may start reporting fake transactions so that they can increase their scores. A workaround to this may be to use the ratio of a peer's service provision to service consumption instead of just the amount of service provisions.

4.2.4.3 Maximum Flow-based Decisions

Having presented the implementation of the maximum flow algorithm, its use in terms of the decision function will now be shown. In this case, the total amount of service a team has offered is not used explicitly. Rather, when judging if a client deserves to be served by an access point, the maximum flow of provided service between the two teams involved in the transaction is considered. Namely, since there is a receipt graph whose edges denote service provision, running the maximum flow algorithm between two graph vertices (teams) returns the amount of service that the source team *indirectly* owes to the destination team.

From the above discussion it appears that peers should reciprocate service directly to each other or indirectly via other peers. The example of Figure 4.5 will make things clear. In this figure, the maximum flow from team A to team B is 3. That is, although the two

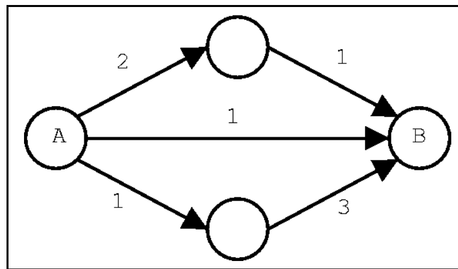

Figure 4.5: Receipt graph segment

nodes are connected directly via only one edge with weight equal to 1, the total amount of service that A indirectly owes to B is 3. As it seems, service is treated as a transitive value between teams.

When the decision function is applied to a pair of teams, the maximum flow between them is computed in both directions. The criterion for deciding whether access should be granted to a client is similar to that of the score-based function described in Section 4.2.4.2. That is, access should be granted with probability equal to the ratio of the maximum flow from the provider to the consumer team to the maximum flow in the opposite direction. When there is no flow from the consumer to the provider, it is implied that the consumer *owes* no service to the provider. In this case, if there is flow in the opposite direction (implying that the provider *owes* some service to the consumer), service is granted. The

probability with which access is granted is calculated via the following formula:

$$P(access) = \begin{cases} 1 & \text{if } maxflow(C \to Pr) = 0 \\ \min\{1, \dfrac{maxflow(Pr \to C)}{maxflow(C \to Pr)}\} & \text{otherwise} \end{cases} \quad (4.2)$$

The notation in the above formula is the same as in formula 4.1. $maxflow(C \to Pr)$ represents the maximum flow from the consumer to the provider team.

In case no flow exists in either direction, instead of unconditionally providing access, as Formula 4.2 implies, there are various criteria that can be used by the function to output a decision. The decision then may be random. A more accurate result, though, may be produced if access is granted with probability equal to the ratio of the sum of weights of the consumer's incoming edges to that of the provider's incoming edges. Recall that the more the incoming receipts of a graph vertex, the more the amount of service other teams *owe* to the team represented by that node. Intuitively, when no flow exists between two nodes, this ratio is an estimate of the ratio of the utility of the two teams for the community.

As to the implementation of this function, first the two teams involved in the decision are looked up and retrieved from the hash table storing teams. The `PWNC_TEAM` pointers retrieved have a `graph_node` field, which points to the respective nodes in the receipt graph (for more information on the receipt graph data structure, see 4.2.3.4). The graph nodes are in fact `AL_HEAD` pointers, which are passed as arguments to the maximum flow function. Its prototype is as follows:

```
long maxflow_run(AL_HEAD *source, AL_HEAD *sink)
```

`maxflow_run` returns the maximum flow from `source` to `sink`. It is run for both directions, that is from the provider to the consumer and vice versa. The `rdb_judge` function will return 1 (access should be granted) according to the probability calculated using formula 4.2.

4.3 Access Point Agent

4.3.1 Architecture

The behavior of a generic access point software agent was specified in Section 3.4.2.2 and 3.4.3.2, for the centralized and the decentralized cases respectively. The access point uses

is a multithreaded TCP server to listen for client messages and to communicate with the receipt repository residing in a TCA or in a team server. The access point is responsible for maintaining per-session state, where information such as session timestamps, timers and traffic measurements are kept. Furthermore, it is responsible for blocking access to unauthorized clients and accurately measuring client session traffic. Figure 4.6 shows the modules that comprise the P2PWNC access point software agent.

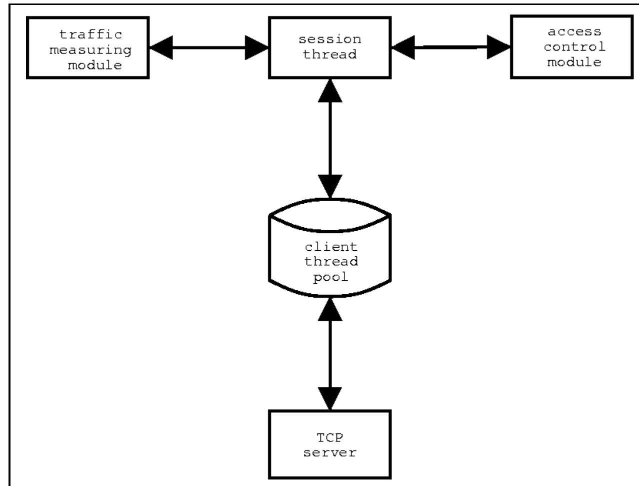

Figure 4.6: Access point agent modules

The access point software is desinged to run in various platforms, but is intented for use on top of embedded linux-based wireless access points. Performance measurements presented in Section 5 show that it is feasible to provide the functionality of the P2PWNC protocol efficiently over such constrained devices.

4.3.2 Client Session Handling

The number of client sessions that can run simultaneously is limited. For each session there is a client thread which handles session state and communication with the client. As described in Section 4.1.3, there is a thread pool from which an available thread is picked for a new session (via a `start_new_thread` call), as soon as the server has accepted a new TCP connection from a client.

The state maintained for each session includes the following:

Client IP address The IP address of a client is permanent during a session. Also, since IP addresses are assigned to clients by a DHCP server, they are also unique. To

avoid duplicate sessions, there is a hash table where client information are stored. In this structure, clients are keyed by their IP address. A new record is added to the hash table as soon as a session has been eshtablished, that is after the client has been admitted to the system.

Client certificate The certificate of the client is extracted by the CONN message at the beginning of a session and is verified. If verification fails, the session is cancelled and resources are released.

Last receipt The most recent receipt that a client has sent during a session is kept by the access point, since there is no need to store older receipts. After all, only one receipt per session is considered valid, and that is the last one. The access point periodically receives fresh receipts in reply to RREQ messages.

Session timestamp The session timestamp indicates the moment when the client is granted access to the system. It is unique for a session and it is contained in all subsequent receipts. Its format has been specified in Chapter 3.

Forwarded traffic The volume of data a client has uploaded and downloaded during a session is being recorded by the traffic measuring module. Traffic measuring begins as soon as the client is admitted to the network.

Index of client thread As described in Section 4.1.3, the thread pool is represented by an array of threads of fixed length. This state information refers to the index of the client thread in question in the pool array.

Timers There are various timers that control the access point behavior during a session. These timers are implementation dependent and are not specified by the protocol. Examining session timers in the order that they appear in the lifecycle of a session, the first one of them is activated as soon as the new thread has been started, while waiting for the CONN message from a client (almost the same applies in the decentralized case, where a number of RCPT messages precede the CONN message). A timeout may also occur while waiting for a connection to the receipt repository server and when expecting the outcome of the decision function. Another timer needs to be employed each time a receipt is expected.

As specified by the protocol, sessions are client initiated. In the decentralized case, optionally, the client may begin the session by sending RCPT messages. The receipts are

thus received, parsed, verified and send to the receipt repository. Afterwards, the session handling thread receivs a CONN message, which contains the client certificate. After the certificate is verified, the access point initiates a TCP connection to the receipt repository. This may either be a trusted central authority or a team server. The receipt repository runs the decision function and sends the access point a QRSP message.

It should be noted that in the decentralized case, the team server may be collocated with the access point. For example, they may both reside in the access point's firmware. As of today, they are two separate modules, which communicate over TCP even in the case of running on the same host.

If the outcome of the decision function is negative for the client, the session must be stopped. Thus, the client information are removed from the hash table, state resources must be freed and the thread must exit and be returned to the thread pool.

Otherwise, the client should be admitted to the network. For this purpose, the session handling module needs to interoperate with the access control and the traffic measurement modules (their functionality will be described in Section 4.3.4 and 4.3.5 respectively). Since the access control module is a separate process which communicates with the session handling threads via a message queue, it is informed by the client thread that the IP address of the client should no more be blocked. Also, the traffic measurements module should from now on keep track of the traffic initiated by the specific client. Traffic logging will be explained in detail in Section 4.3.4.

After the above steps, a session is refreshed by sending the client RREQ messages and receiving receipts. A timer is used to control the intervals when receipts are requested. Choosing these intervals is left to the application. Frequent receipt requests result in communication and processing overhead in both the client and the access point side, since a receipt needs to be signed by the client and verified by the access point. On the other hand, an access point that sends RREQ rarely is susceptible to attacks. Namely, since the RREQ is a request that the client acknowledges the service that she has enjoyed, she may refuse to sign a receipt after consuming an important amount of server resources. The session will be ended, but the client will have overconsumed service for free.

The value of the *weight* field of an RREQ is acquired by a query to the traffic logging measurement module. As it seems, all the RREQ messages sent over a session are identical, except for the value of the *weight* header, which is increasing.

The session is ended when either a receipt verification fails, a timeout occurs when waiting for a receipt or the TCP connection is closed. The last receipt of the session (that

is the one with the largest weight value), if any, is send to the receipt repository. Finally, the state resources are freed and the thread is returned to the thread pool.

4.3.3 Communication with the Receipt Repository

Communication with the receipt repository happens in two of the phases of a client session. First, it occurs in the beginning of a session. In decentralized case the session may start with the client supplying receipts to the access point. After sanity checks, these receipts are sent to the receipt repository via RCPT messages. It should be noted that the receipt repository in consideration is the access point's team server. Then, a QUER message is send to the repository so that a decision on whether to grant access to a client is taken. This step is the same in both the decentralized and the centralized case. Then, the QRSP message is received and parsed. If a timeout occurs while connecting to the repository or while waiting for the QRSP message, the client session ends.

Second, in the end of the session, the session handling thread sends an RCPT message to the repository, so that the session's receipt is stored. As it seems, communication with the receipt repository is stateless.

4.3.4 Traffic Measurements

For proper implementation of an access point software agent, a module that performs accurate measurements of client traffic is vital. However, since agents are autonomous they may skip this part, and thus this module is not protocol-specified.

In the P2PWNC protocol reference implementation, traffic measurements are performed by a Linux kernel module. This module is based on the Netfilter framework [6], which is standard in Linux 2.4 and 2.6 kernels. The choice of measuring traffic inside the kernel rather than from userspace was mandated by the need to obtain accurate results. Thus, no loss of accuracy is incurred during measuring traffic, at the expense of the time overhead caused by the extra processing of each packet to extract traffic information, which might slow down network performance. However, since this extra processing is minimal, network speed degradation is negligible. What is more, using a userspace utility for packet capturing, such as *libpcap*, would take up precious space in the firmware of the space-limited embedded wireless access points that our system is targeted at.

The traffic logging module keeps traffic statistics in a character device file (`/dev/tstats`). Statistics are kept for the IP addresses that are currently being used by client sessions. Each

session can uniquely be identified by the IP address of the client. Traffic information for each session is kept in a struct of the following format:

```
struct stats {
        unsigned int nClientIP;
        unsigned int nTotalPacketsIn;
        unsigned int nTotalPacketsOut;
        unsigned int nTotalBytesIn;
        unsigned int nTotalBytesOut;
};
```

The traffic information of all concurrent sessions are kept in an array which resides in `/dev/tstats`. Its format is as follows:

```
struct client_list {
        unsigned int nRecNum;
        struct stats clist[MAX_CLIENTS];
};
```

As obvious, `client_list` contains an array of `stats` structs called `clist`. `nRecNum` is the number of elements in `clist`. The maximum number of sessions for which traffic information can be kept is `MAX_CLIENTS`, which can be set arbitrarily. In our reference implementation, the value of `MAX_CLIENTS` is 1024, which is more than enough.

When a new client session has been established, that is after sending the client a CACK message, state concerning the client's traffic volume is set. That is, a new element is added in `clist`. Conversely, when a session has ended, this state must be removed. Periodically during the session the character device is polled by the session handling thread to acquire the current traffic measurement for the particular session, so that an RREQ message is constructed. The above three types of operations take place via `ioctl` calls on `/dev/tstats` from the userspace application (client thread). Each time such an `ioctl` takes place, the kernel module handles the request and, if necessary, copies data to userspace. For example, when there is a request for traffic statistic for a specified client (whose IP address is passed as an argument to `ioctl`), `clist` is traversed to locate the address in question and the respective array element is copied to a userspace buffer (via the `copy_to_user` call).

During the session, the trafic logging module intercepts every packet and checks its source and destination IP addresses against the addresses of the client whose traffic is being

measured (that is the elements of `clist`). If the source or destination IP addresses match with those of clients, the fields of the respective `clist` element are updated accordingly.

From the perspective of the userspace application, that is the access point's multithreaded TCP server, in order for the system to work properly, the character device described needs to be created as soon as the server starts up (if it does not already exist). This is performed via the `mknod` system call. The following command sequence carries out this task:

```
dev_t tstats_dev;
tstats_dev = makedev(241, 0);
mknod ("/dev/tstats", S_IFCHR | 0666, tstats_dev);
```

The above create a character device whose major number is 241 and its minor number is 0. If the path specified already exists, the call will fail. Moreover, the proper `ioctls` need to be performed regularly, as specified above.

4.3.5 Network Access Control

Since our implementation of the access point software is based on Linux 2.4/2.6 kernels, access control is carried out using the *iptables* package. *Iptables* is a firewall system built on top of the *netfilter* [6] framework. Apart from its advantages such as flexibility and efficiency, *iptables* has a drawback; it lacks an official API. Therefore, in this implementation, firewall operations such as blocking or granting access to clients are performed programmatically via the `system` function.

A separate process (it will be referred to as `cmd_handler`) is responsible for carrying out these operations. This process communicates with the multithreaded server process via a message queue. `cmd_handler` should be started prior to the access point server. It is responsible for creating the queue using the `msgget` System V IPC call. From then on, it waits for messages from the access point server.

The access point server, in turn, when started up, has to get a handle of the message queue (using `msgget`). All firewall commands are then sent to `cmd_handler` via the message queue. Messages have the following format:

```
typedef struct msgbuf {
        long mtype;
        char mtext[1];
} MSG_BUF;
```

The `mtext` field is in fact a string which needs to be allocated before making use of the struct. For example, before receiving a message from the queue, space needs to be allocated for the string and then the `MSG_BUF` variable must be passed as an argument to the `msgrcv` function.

Session threads of the access point server send the `cmd_handler` process iptables commands. Such a command occupies the `mtext` field of an `MSG_BUF` struct. The command handling process receives messages from the queue and executes the specified commands via the `system` function.

Firewall commands need to be executed when the server starts up, since access to the network should be. Only the port where the server listens for client messages should be left open. First, the access point starts by creating two new iptables chains which will be responsible for the traffic of the P2PWNC clients. This is done as follows:

```
iptables -N PWNC_INPUT
iptables -N PWNC_FORWARD
```

After creating the new chains, it inserts the following rules which “redirect” the INPUT and FORWARD chains to the new chains created. In the examples that follow we suppose that the local subnet in which clients are assigned addresses is 192.168.1.1, with a 255.255.255.0 subnet mask.

```
iptables -I INPUT 1 -s 192.168.1.0/24 -j PWNC_INPUT
iptables -I FORWARD 1 -s 192.168.1.0/24 -j PWNC_FORWARD
```

Then, all incoming traffic is blocked but for messages to the server standard TCP port (9999). This is achieved by appending the following rules to the firewall rule set:

```
iptables -A PWNC_INPUT -s 192.168.1.0/24 --protocol tcp
        --destination-port 9999 -j ACCEPT
iptables -A PWNC_INPUT -s 192.168.1.0/24 -j REJECT
iptables -A PWNC_FORWARD -s 192.168.1.0/24 -j REJECT
```

From this point on, when a new client is admitted to the network, rules of the following form will be placed on top of the rule set:

```
iptables -I PWNC_INPUT 1 -s 192.168.1.100 -j ACCEPT
iptables -I PWNC_FORWARD 1 -s 192.168.1.100 -j ACCEPT
```

The above rules provide full access to the client with the IP address 192.168.1.100. Note that these rules are appended on top of the existing rules, since when a new packet arrives, rules are applied to it in the order that they appear in the set. Thus, if the first rule is applicable, all others are ignored. The rules that block access still exist, but they are never applied since the above rules are used first.

When a session is ended, the session handling thread sends `cmd_handler` the appropriate commands so that the above rules disappear and the IP address of the specified client is again blocked. Such commands are shown below (the -D option is used for rule removal):

```
iptables -D PWNC_INPUT -s 192.168.1.100 -j ACCEPT
iptables -D PWNC_FORWARD -s 192.168.1.100 -j ACCEPT
```

At this point, is should be noted that if one needs other applications such as an SSH daemon to operate properly, the appropriate iptables commands that permit traffic concerning these applications should be issued.

4.4 Mobile User Agent

The mobile user software agent is somewhat simpler that that of the access point. As of this document, there are working implementations in C and Java (albeit not with the same capabilities).

4.4.1 Operation in the Centralized Case

In the centralized case, client software need only implement the basic functions of initiating a session with an access point, receiving and parsing periodical RREQ messages and signing receipts in reply to these messages. No receipt storage functionality is required.

A subtlety in implementing mobile user agent software is now discussed. It has been explained that a client is periodically requested to sign a receipt and send it to the access point. The RREQ message issuing this request includes the measurement of the traffic that the access point has forwarded on behalf of the client. Since the receipt is the access point's "payment" of the service that has been provided, the higher the receipt weight the more the cost of the service for the client.

If the client does not measure the traffic that she has initiated, then she is susceptible to service overcharging by the access points. That is, an access point may place a weight

value much greater that the real traffic volume of the mobile user. If the mobile user signs a receipt with that weight value, she admits that she has used much more service that the actual amount.

Therefore, is is good for the client to have this functionality implemented. In our C Linux implementation, traffic measuring capability is implemented in a similar way as in the traffic measurements module of the access point software (see 4.3.4), that is inside the Linux kernel. Our Java version of the client does not have such capabilities as of this writing.

4.4.2 Operation in the Decentralized Case

In the decentralized mode, a client may carry receipts so that they will be presented to the access point at the beginning of the session, to have more chances of being admitted to the visited network.

For this, the client needs to have a receipt repository module built in its software. This repository need not be as sophisticated as that of the team server's or the TCA's. This repository is filled via UPDT messages sent to the repository server of the team the client belongs to.

Since the client need not run the decision algorithm, its repository does not support for receipt graph operations. Thus, its structure is simpler, as there is no graph representation of data. Apart from that, receipts are stored in the same way as in the repository described in 4.2. That is, receipts are pointed to by the leafs of a red-black tree (see 4.2.3.2). Permanent storage take place as described in 4.2.3.3.

In the case of updating the receipt repository, the client first locates the most recent receipt of its database and sends its team server an UPDT message containing the most recent receipt's timestamp. Then, receives a set of recent receipts and inserts them in the red-black tree and stores them permanently.

In the case of sending the visited access points receipts in the beginning of the session, the client performs a range query on the tree. During the query, each accessed node (receipt) is sent to the access point.

Apart from the above, the other operations of the client in the decentralized case are the same as in the centralized.

4.5 Trusted Central Authority

The TCA module, as has been explained, is needed in the centralized case of the P2PWNC scheme. It acts as a central receipt repository, answers queries about granting access using a decision function and generates key pairs for the P2PWNC teams.

The TCA is equipped with a receipt repository module as described in Section 4.2. Also, it keeps track of the teams that it has generated. That is, the public keys of the teams that the TCA has created are kept and used for checking a receipt's validity. When a new receipt is encountered, the two peers that are involved in it are looked up in a teams database. In case a team is not found, it means that its public key is invalid (not generated by the TCA).

The implementation of the TCA presented here is written in C and it is Linux based. However, other implementations (such as a Java-based one) can easily be developed.

The basic module is a multithreaded TCP server. This server is stateless. It accepts TCP connections and answers to QUER and RCPT messages. When receiving a QUER message, it executes the decision function and returns its outcome to the issuing access point via a QRSP message. When receiving an RCPT message, it passes it to the receipt repository module, which performs the necessary checks and eventually stores it.

The TCA module need not know the details of the operation of the receipt repository module. They communicate via a simple API, which gives receipt insertion and decision function facilities. These operations are carried out transparently to the TCA server.

As far as team generation utilities are concerned, a separate process is responsible for notifying the TCA server that a new team has been generated. New key pairs are being issued by another application (a command line utility). When such a pair has been constructed, a process that communicates with the TCA via a message queue sends the server module a message containing the public key of the new team. The TCA server reads messages from this message queue and inserts (or removes) the specified public keys from its team database. This way, the TCA server need not be restarted every time new teams are generated.

The team database is in fact a repository of team public keys. On server startup, these keys are loaded in a hash table for fast lookup. When the TCA receives a message from the queue it updates this hash table accordingly. The architecture of the TCA software modules is shown in Figure 4.7.

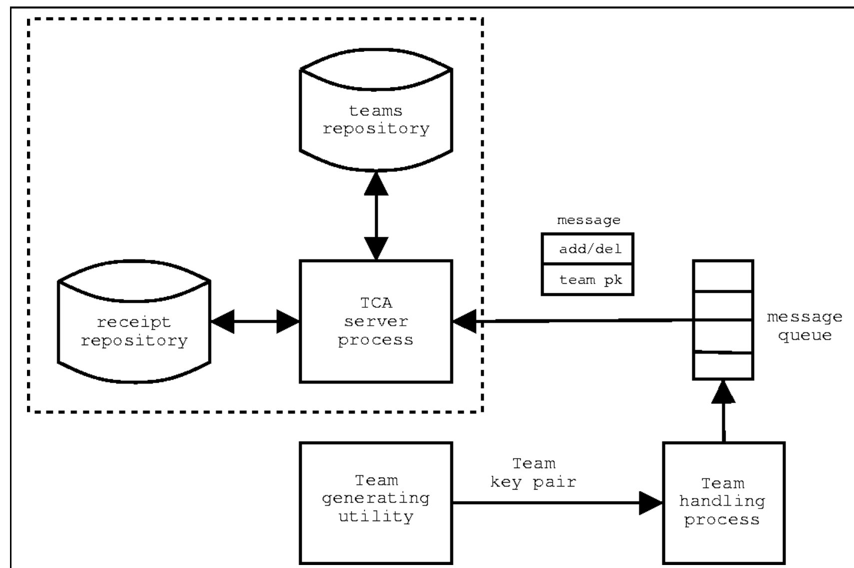

Figure 4.7: TCA software architecture

4.6 Team Server

The team server operates in a similar fashion as the TCA. However, unlike the TCA, it is not globally trusted. It acts like a trusted central receipt repository only for the access points and members of one team. It uses the same underlying receipt repository module as the TCA, with some minor differences. The main difference is that it does not maintain a team database, since in the decentralized case, team key pairs are self-generated. Receipt storage and decision function operation is identical to that of the TCA and is equally transparent to the team server.

The team server module is a multithreaded stateless TCP server. It accepts TCP connections by team members and access points and starts a new thread for each connection. It receives QUER and RCPT messages in the same way as the TCA does. Also, it can reply to UPDT messages from a team's mobile users.

Each time an UPDT message is received, the team server performs a range query on the receipt repository, using the `rdb_range_query` receipt repository API call. Each receipt in the specified range (that is, with timestamp greater or equal to the one specified) is sent to the requesting client via an RCPT message.

The team server software is also written in C and runs in Linux. It can also be ported easily to the Java platform. However, since the team server may be collocated with an

access point, the Linux version can be included in an access pointss firmware, unlike a Java version (as it would require that the Java VM is included in the firmware, too). In the case when the team server runs on top of an access point, communication with the access point module does not change. They still communicate via TCP, although they are located at the same host.

5 Performance Evaluation

5.1 System Testbed

The systems on which the tests were carried out were an AMD AthlonXP 2800 laptop and a Linksys WRT54GS 802.11g wireless router. The AthlonXP machine has 512Mb of RAM and it has a 2.08GHz CPU. Its operating system is RedHat Linux 8.0 with a 2.4.18 kernel.

The Linksys router has a 200MHz MIPS CPU, 32Mb RAM and 8Mb of flash memory. In this work, the firmware of the device has been modifiad so that the P2PWNC software is included, as well as some other auxiliary applications such as the Dropbear [2] SSH daemon. It runs embedded Linux with a 2.4.20 kernel and it comes with an out-of-the-box read-only compressed file system (cramfs) residing on the router's flash memory. However, there are 32Kb of non-volatile RAM (nvram) on the flash card, which are writable and are organized in variable-value pairs. This memory area is mainly used for permanent storage of configuration settings, since, obviously, data that reside in the system's main memory are erased when the device is powered off. Nvram can be accessed via the `nvram` command line utility or, programmatically, via the nvram API which is included with the firmware's Linux source distribution.

No attempt has been made to change the router's file system to a writable one, although there are such approaches. OpenWRT [8], for instance, is a Linux distribution targeted at the Linksys WRT54G/GS routers and one of its characteristics is the JFFS2 [28] file system, which is both compressed and writable. The P2PWNC software modules have been included in the firmware image, while test programs, scripts and data are put in the router's RAM (which is mapped to the /tmp directory). Measurement data are generated and sent to another host via the router's network interfaces.

The cryptography library used was OpenSSL 0.9.8 (beta5). OpenSSL [7] is an open source toolkit and it is one of the most widely used cryptography libraries. It is written in C and assembly, which is however machine depended. Assembly code for speeding up

some operations is provided mainly for the x86 platforms (such as Intel or AMD Athlon). The 0.9.8 version was used, though in its beta version, due to its support for elliptic curve cryptography. The stable release of the 0.9.8 is expected by July 2005.

The programs developed for the tests, as well as the OpenSSL library, were built using the gcc version 3.2 compiler. For the MIPS platform, cross-compilation on an x86 machine was carried out. Level 3 code optimization was used (`-O3` gcc flag) on both platforms. For the MIPS CPU, the `-mcpu=r4600` and `-mips2` flags were also added.

The exact specifications of the two platforms are summarized in Table 5.1.

Table 5.1: Platform specifications

	Athlon XP	**Linksys**
System type	AMD AthlonXP	Broadcom MIPSel
CPU speed	2.08GHz	200MHz
RAM	512Mb	32Mb
Permanent storage	60Gb hard disk	8Mb flash (read only) 32Kb NVRAM
Operating System	RedHat Linux 8.0, 2.4.18 kernel	Linux 2.4.20 (Broadcom)
OpenSSL version	0.9.8 beta 5	0.9.8 beta 5
Compiler	gcc v3.2	gcc v3.2 (cross compilation)
GCC optimizations	-O3	-O3 -mcpu=r4600 -mips2

5.2 Performance Metrics

5.2.1 Time Measurements

In the tests that are presented in this section, time measurements have been performed via the `times` function, which is defined in the `<sys/times.h>` header file and its prototype is as follows:

```
clock_t times(struct tms *buf);
```

On Linux systems, it returns the number of clock ticks that have elapsed since the system was booted. Its return value is not being used in the measurements presented here. Rather, the information contained in the `buf` argument after the execution of the function are used. This argument a `struct tms` pointer. This struct is presented below:

```
struct tms {
      clock_t tms_utime;  /* user time */
      clock_t tms_stime;  /* system time */
      clock_t tms_cutime; /* user time of children */
      clock_t tms_cstime; /* system time of children */
};
```

`tms_utime` represents the time spent by the CPU executing instructions of the calling process (while the process was running in user mode). System time is the CPU time spent by the system while executing tasks on behalf of the process (that is, while the process was executed in kernel mode, for example, performing I/O operations). The other two fields refer to user and system time (as explained above) spent by children of the process.

When referring to time, `tms_utime` will be implied. Using `tms_utime` on its own and omitting system time is more accurate for the sake of comparisons, since it involves only instructions of the application code. On the other hand, calculating wall time (via, for instance, the `gettimeofday` call) would give a more realistic view of an applications performance, but it depends heavily on the system's load and thus is not appropriate for our measurements. Using the `times` function is also compatible with the *speed* utility, which is provided with the OpenSSL package. The *speed* program, as its name implies, measures the speed of cryptographic operations, as implemented in the OpenSSL package. When supported by the system, it uses the `times` function for speed measurements.

It should be noticed that the granularity of the `times` function is not fine. Its resolution is defined by the frequency of clock ticks. In the platforms that these tests were run, the frequency of clock ticks was 100ticks/sec. Thus, the lowest time interval that can be accurately measured by the `times` function is 10msec. The clock ticks frequency can be acquired via the following call:

```
long ticks = sysconf(_SC_CLK_TCK);
```

`gettimeofday` provides for microsecond time resolution, but, as mentioned above, it does not measure time per process and it may report false measurements depending on the system's current load.

To overcome the low resolution of the `times` function, consecutive runs of the same experiments are performed. The total time needed for these consecutive executions is measured and averaged over the number of these runs.

5.2.2 Space Measurements

Measurements concerning memory usage by P2PWNC software modules are expressed by the *resident set size* (rss) of processes. The rss of a process is the number of pages it has in real memory. It includes the process' *text* (program code), *data* and *stack* space.

Memory usage measurements report the maximum rss value during the lifetime of the process in kilobytes. For example, when running the maximum flow algorithm, the rss will vary during its execution, since memory will be dynamically allocated and freed. In the measurements presented in this document, the maximum of these rss values is of interest. Information about the rss of a process can be accessed via the `/proc` file system.

To have an accurate view of the memory usage of a process which makes a frequent and unpredictable use of dynamic memory, its rss needs to be constantly monitored. Since only the greatest rss value in the lifetime of a process is of interest, it is adequate to measure memory usage in the event of the allocation of a new data segment via the `malloc` call (freeing memory obviously cannot result in an rss increase).

For this purpose, the `malloc` has been overridden so that it can monitor memory usage when appropriate and update the maximum rss value for the process. Whenever needed, the `/proc` file system entry of the process is polled so that the resident set size is determined. In Linux systems, for each process, there is the respective directory under `/proc`. For instance, for a process whose pid is 2345 there is a `/proc/2345` directory where various information is stored. In our case, the required information can be found in the `/proc/2345/stat` or `/proc/2345/statm` files. An example of the contents of the `statm` file is shown below:

```
1898 1898 81 87 0 1811 1817
```

The second value is the rss of the process measured in pages. To obtain its value in kilobytes, it needs to be multiplied by the page size in the system where the process runs. In both platforms where the P2PWNC software is being tested, the page size is 4096 bytes. This value can be obtained via the `getpagesize` system call. The rss value in kilobytes may also be accessed via the `/proc/2345/stat` file.

It should be noted that this approach is recommended only for testing purposes. Polling the proc file system on every malloc call is very time consuming. Alternatively, in some systems, the maximum rss of a process can be obtained via the `getrusage` call. However, this information is not available under Linux.

5.3 Cryptographic Operations

5.3.1 Parameters

Since in the P2PWNC software not all of the OpenSSL library's functionality is employed, only the features the performance of whom is of interest to the system are tested. In particular, the efficiency in key generation, digital signing and signature verification is measured.

In the tests that will be described in this section, the performance of elliptic curve cryptography compared to the RSA cryptosystem is studied. Experiments were carried out on the two platforms specified in Section 5.1 using the time metrics described in Section 5.2. In particular, to overcome the granularity problems of the `times` function, each test was carried out for at least 10 seconds or at least 10 consecutive times. For example, the same signature was being generated for a period of ten seconds. The user time spent for this procedure, as well as the number of signature operations were measured. After the 10 seconds had elapsed, the total user time spent was divided by the number of operations performed, to acquire the average signature time. In the case of an operation that takes more that 10 seconds to be executed (for instance, generation of a 3072-bit RSA key), 10 consecutive iterations were carried out and the time spent per operation was being computed as above.

As to the elliptic curve parameters, randomly verifiable curves over the finite field F_p were used. For their exact domain parameters and their semantics one can refer to [21]. The RSA cryptosystem parameters are the length of the key and the size of the public exponent, which is fixed to 65537.

The operations that are benchmarked are key generation, digital signing and signature verification. In all the experiments that were carried out, the OpenSSL pseudo-random generator was seeded with the same 2048-byte seed. It should be pointed out that the digital signature and verification times include the time needed to generate the SHA-1 hash of the signed data, as producing the message digest of the data that are to be signed is part of the specification of the signing and verification operations. Since, in practice, a P2PWNC agent may normally be requested to sign or verify a receipt, the length of the data that were input to the SHA-1 hash function was the sum of the lengths of a member certificate, a public key and two 4-byte integers (receipt timestamp an weight). For example, in the case of the tests of the performance of 192-bit ECDSA signature operations, the length of the input data is

$$3 * 49 + 48 + 2 * 4 = 203 \text{ bytes}$$

(since, according to the P2PWNC representation of cryptographic data, such a public key needs 49 bytes to be represented, while an ECDSA signature requires 48 bytes).

5.3.2 Measurements

The results of the tests are shown in Table 5.2. The first column indicates the bit length of the keys involved in each operation. Each row refers to experiments using keys of equivalent security levels for RSA end ECC. The list of comparable key sizes between RSA end ECC cryptosystems can be found in [20]. The first numbers in the "Bit length" column refer to RSA key sizes while the second numbers refer to ECC key sizes. For example, 160-bit ECC and 1024-bit RSA keys provide similar security.

Table 5.2 is divided in three sections, one for each of the operations that is benchmarked. For each operation, the results for the two platforms are under the "Athlon XP" and "Linksys" columns. For each platform, there are two sub-columns showing the average time spent on an operation using RSA and ECC cryptography respectively (for the same security level). Execution times are expressed in seconds.

The first noticeable advantage of ECC over RSA cryptography is the fact that it offers the same security levels with much smaller keys. The space efficiency of ECC becomes more apparent as the security level increases. What is more, since the bit length of RSA keys grows faster as more security is required, the time needed to perform cryptographic operations increases in a similar pace. Table 5.3 shows the key size ratio of ECC and RSA cryptography for the same security levels.

In 5.4, the performance ratio of RSA over ECC operations is shown. The ratios shown are derived from Table 5.2. Higher values imply a slower operation. Obviously, apart from the case of verifications, the speed advantage of ECC over RSA increases even faster than its space advantage.

The main drawback of using ECC for the needs of the P2PWNC protocol is its higher verification times. Given that such operations will often be taking place in the access point agent, which will most probably be running in top of an embedded device, the access point's operation may be slown down in the presence of a large number of clients. However, even under such circumstances, the verification of a P2PWNC receipt takes time in the order of 0.1 seconds for 160, 192 and 224-bit curves on the Linksys platform (200MHz processor). This value is acceptable for most cases. Problems may occur in the distributed design of the P2PWNC scheme, where a client may start a session offering receipts to the visited access point. In this case, workarounds to overcome the verification load may

Table 5.2: Cryptographic operation performance

Key Generation				
	Athlon XP		Linksys	
Bit length	RSA	ECC	RSA	ECC
1024/160	0.3738	0.0051	8.8064	0.0924
1536/192	1.4827	0.0046	22.0127	0.0827
2048/224	3.4945	0.0056	49.9800	0.1105
3072/256	11.2082	0.0067	277.7727	0.3646
Signature				
	Athlon XP		Linksys	
Bit length	RSA	ECC	RSA	ECC
1024/160	0.0090	0.0013	0.3006	0.0203
1536/192	0.0259	0.0012	0.6556	0.0185
2048/224	0.0473	0.0014	1.5290	0.0234
3072/256	0.1491	0.0017	3.9390	0.0731
Verification				
	Athlon XP		Linksys	
Bit length	RSA	ECC	RSA	ECC
1024/160	0.0004	0.0065	0.0123	0.1147
1536/192	0.0008	0.0060	0.0214	0.0999
2048/224	0.0013	0.0071	0.0379	0.1357
3072/256	0.0028	0.0086	0.0753	0.4530

Table 5.3: Key size ratio

Security level	Ratio (RSA/ECC)
1024/160	6.4 : 1
1536/192	8 : 1
2048/224	9.14 : 1
3072/256	12 : 1

include performing offline receipt verification or by limiting the number of the accepted receipts according to the current system load.

Table 5.4: Cryptographic operations running time ratios

Key Generation		
Bit length	Athlon XP	Linksys
1024/160	74.12 : 1	95.31 : 1
1536/192	322.33 : 1	266.23 : 1
2048/224	624.02 : 1	452.31 : 1
3072/256	1672.87 : 1	761.86 : 1
Signature		
Bit length	Athlon XP	Linksys
1024/160	6.92 : 1	14.81 : 1
1536/192	21.58 : 1	35.44 : 1
2048/224	33.79 : 1	65.34 : 1
3072/256	87.71 : 1	53.89 : 1
Verification		
Bit length	Athlon XP	Linksys
1024/160	1 : 16.25	1 : 9.33
1536/192	1 : 7.50	1 : 4.67
2048/224	1 : 5.46	1 : 3.58
3072/256	1 : 3.07	1 : 6.02

As one may observe, although the CPU of the Linksys router is 10 times slower than that of the AthlonXP machine, operations were more than 10 times slower on the wireless access point, contrary to what one might expect. Possible reasons for that mismatch include the different processor architectures and, above all, the fact that some routines for operations on big integers have been implemented in assembler only for the x86 platform. The respective assembler routines are not available for the MIPS architecture in the OpenSSL package, and this may explain the slower than expected operations. Hence, it occurs that further optimizations are possible for the MIPS platform.

Evaluating the results of the experiments on the performance of cryptography operations in terms of the P2PWNC protocol, it seems that very long keys are inappropriate due to their space overhead and the slow and resource consuming operations that they incur. One should always bear in mind that the P2PWNC scheme and protocol were designed with resource-constrained embedded devices in mind. Such devices may be PDAs or embedded wireless access points, with capabilities and performance similar to those of the

Linksys WRT54GS router. On the one hand, the roamer who may carry a PDA is interested in digital signature speed, while the access point cares more about fast verification times. From Tables 5.2 and 5.4, it is confirmed that public key operations (e.g. signature verifications) are faster when RSA is employed, while for private key operations (such as digital signing), ECC seems more appropriate.

Considering that portable devices such as PDAs are battery-powered, while access points typically operate on AC power, the applicability of ECC is more obvious. Power consumption is of more concern on the client side (PDA). Therefore, since the client bears the weight of producing digital signatures, it is more crucial that signing is performed with as little processing overhead as possible, so that more battery power is conserved. As obvious from Table 5.4, RSA signing requires much more processing than ECC and is thus less appropriate for battery-powered devices. Using ECC trades verification for signing efficiency. Thus, choosing ECC burdens the access points with more verification overhead. However, since access points typically do not have battery limitations, the computational overhead incurred by ECC as to signature verifications is of minor importance.

Key generation, where ECC also has a clear advantage, is not discussed, since such operations are performed rarely in the P2PWNC scheme.

Summing up the results of the cryptographic operations experiments, it seems that the most appropriate solution of cryptosystem parameters is using the 192-bit randomly verifiable elliptic curve. As can be seen in [21], this curve is recommended by the X9.62, X9.63 and NIST standards, while being compliant to other standards, too. This choice offers better security than 160-bit curves, while requiring the same (or even less) verification and signature times. Also, compared to the equivalent RSA 1536-bit keys, it offers the same security with 8 times smaller keys.

5.4 Maximum Flow Algorithm Performance

As stated in Section 4.2.4.3, the decision on whether a consumer (mobile user) will be granted access or not may be based on the execution of a maximum flow algorithm on the graph formed by the receipts and the peers of the P2PWNC. In this section, the performance of our implementation of the maximum flow algorithm, as presented in Section 4.2.3.5 is evaluated.

Two types of experiments have been performed. First, the speed of our maximum flow algorithm is measured, when run on repositories of various sizes. Second, the memory

usage of the above experiments is measured. The metrics used have been described in Section 5.2.

In these experiments, the time spent and the maximum amount memory consumed during the execution of the algorithm on receipt sets of increasing size and for various peer populations have been measured. Each peer corresponds to a graph vertex, while a graph edge represents a consumer - provider pair (an edge is directed from a consumer to a provider). It should be mentioned that one edge may correspond to more that one receipts.

In each of the graphs presented in this section, three curves have been plotted, referring to populations of 100, 500 and 1000 peers respectively. The x-coordinate of a data point indicates the size of the receipt repository on which the algorithm was run. Values on the x-axis start from 100, followed by 1000 receipts and going up to 10000 receipts with a step of 1000 receipts (that is, 100, 1000, 2000, etc. receipts).

It should be noted that the code of the test programs was identical to that of the P2PWNC reference implementation, with the addition of time and memory usage measuring functionality. The program through which the maximum flow algorithm was run included the P2PWNC receipt repository module, as well as the core protocol, cryptography and threads modules. However, in the presented experiments there was no use of multithreading. The inclusion of all these modules, though unnecessary for measuring the algorithm's running time, is of importance when it comes to measuring memory utilization.

5.4.1 Running Time Evaluation

5.4.1.1 Parameters

For the evaluation of the running time of our maximum flow implementation, the `times` function was used, as described in Section 5.2. Since the input to the maximum flow algorithm is a receipt graph and two graph nodes, for each point in the curve, 20 random couples of nodes were chosen. The algorithm was executed for each of these couples on the same graph for at least 3 seconds or 10 iterations and the total time needed for these runs was averaged over the number of iterations. This approach was followed in order to obtain more accurate time measurements, since the resolution of the `times` function on the platforms where the experiments were run was 10msec. As explained in Section 5.2, the resolution of the `times` function was 10ms, and the running time of the algorithm was usually less than that.

The 20 time values obtained by the above procedure were averaged, and, thus, a point

in the curve was output. The rationale behind this method was to test the average running time of the maximum flow algorithm implementation for various random instances on the same graph. The algorithm was run with identical input on both the AMD Athlon XP and the Linksys WRT54GS platforms.

5.4.1.2 Measurements

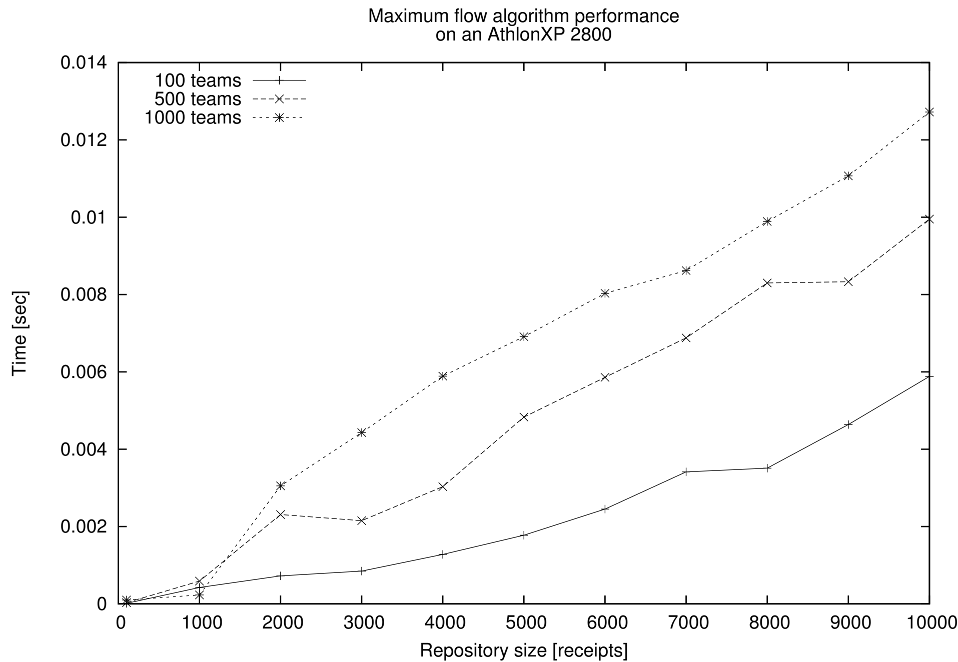

Figure 5.1: Maximum flow running time on an AMD AthlonXP 2800

The results of the tests on the maximum flow algorithm running time are show in Figures 5.1 and 5.2. Figure 5.1 shows the running time of the maximum flow algorithm when run on the AMD AthonXP platform, while the outcome of the same experiment run on the Linksys platform is plotted in Figure 5.2. In both cases the plotted curves follow the same trend. The running time of the algorithm increases proportionally to the size of the receipt repository, when the number of peers (graph nodes) is fixed. As stated in Section 4.2.3.5, the theoretical worst case running time of the FIFO variant of the generic push-relabel maximum flow algorithm is $O(V^3)$, where V is the number of graph nodes. However, for practical use, this has proved inefficient. In the experiments that are shown, the global

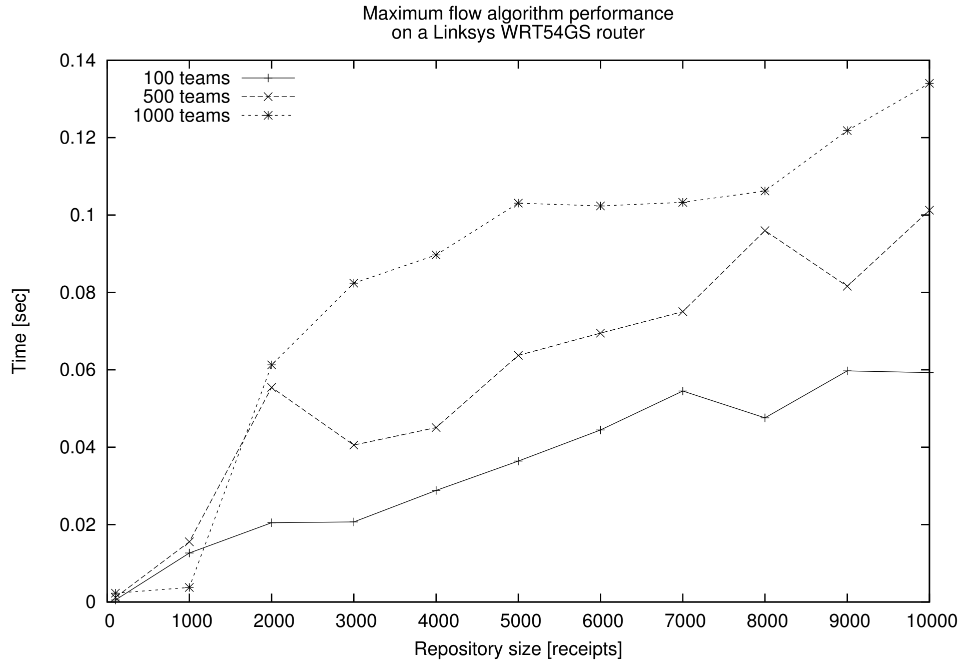

Figure 5.2: Maximum flow running time on Linksys WRT54GS

relabeling heuristic has been implemented (see Section 4.2.3.5) and, due to it, the running time of the algorithm has been drastically improved.

The average time spent on the execution of the algorithm on sets of 10000 receipts and 1000 peers is in the order of a few milliseconds on the AMD AthlonXP platform. On the other hand, the Linksys WRT54GS wireless router is roughly 10 times slower in the execution of the algorithm. For the same data set, it takes around 130 milliseconds on average to run the maximum flow algorithm on the MIPS platform. This is what one might expect, considering that the router's processor is an order of magnitude slower that that of the Athlon XP.

We believe that these time values are acceptable in terms of the P2PWNC scheme. In the case of centralized repositories, more powerful computers are expected to be used (than the Athlon XP). These time bounds are expected to be further improved using additional maximum flow algorithm heuristics, such as the *gap heuristic* [14].

5.4.2 Memory Utilization Evaluation

5.4.2.1 Parameters

In the case of memory usage experiments, the x-axis, in a similar manner to the the case of time measurements, includes 100 values of receipt repository sizes, from 100 to 10000 receipts with a step of 100 receipts. The y-axis represents maximum memory usage by the process in which the maximum flow algorithm is run. For each curve point, the maximum flow algorithm was executed for 20 random pairs. The pairs used to yield the respective data points for the time measurement curves described in the previous section are included. Since time accuracy is of no importance in memory usage experiments, for each pair, only one execution of the algorithm was carried out. During each execution, the memory used by the process was constantly being monitored (see Section 5.2). The curve point's y value represents the maximum of the memory usage measurements in the above 20 runs.

Values of the y-axis are measured in kilobytes and represent a process' maximum resident set size (rss). The meaning of the rss metric was explained in Section 5.2. Memory usage was measured only on the Linksys platform, since it is more resource-constrained and thus memory utilization is of greater concern. Also, since the source code of the experiments was identical, regardless of the platform for which it was compiled, the results of these measurements would be the same for both the Linksys box and the AMD Athlon computer.

The repositories used in these experiments were composed of receipts including 160-bit ECC keys and signatures. Thus, the actual data size of each of these receipts totalled to 211 bytes. The internal representation of such a receipt (see Section 4.1.1.1.3) incurs some extra overhead due to the additional fields of the data structures used. Thus, each 160-bit ECC public key or signature takes up the following memory space:

- 2 bytes for the `algorithm` field
- 2 bytes for the `bits` field
- 4 bytes for the `datalen` filed, which indicates the length of the actual key data
- 41 bytes of the actual key data (`data` pointer)

The above fields total to 49 bytes. The same apply to the representation of an ECDSA digital signature, with the difference that the actual signature data are 40 bytes. Thus, the

total length of the representation of such a signature is 48 bytes. A `PWNC_RCPT` structure containing only 160-bit ECC keys and signatures, takes up the following memory space:

- 49 bytes for the member certificate's team public key
- 49 bytes for the member certificate's user public key
- 48 bytes for the member certificate's signature
- 49 bytes for the service provider's public key
- 48 bytes for the receipt signature
- 4 bytes for the receipt timestamp
- 4 bytes for the receipt weight

Therefore, the total size of such a receipt is 251 bytes in its internal P2PWNC software representation. Thus, a repository of 10000 receipts (of 160-bit ECC keys end signatures) needs at least 2.51 megabytes for the representation of the receipts in main memory. This amount does not include the memory spent on representing the receipt graph, the red-black tree and the hash tables storing pointers to the receipts and teams of the repository.

5.4.2.2 Measurements

As to the memory usage measurements, it appears that memory consumption by the maximum flow algorithm grows only moderately as the number of peers increases. The amount of used memory is dominated by the volume of the receipt repository and the overhead for its graph and tree representation. Thus, memory usage increases proportionally to the volume of the data loaded (receipts). The memory space occupied by receipts was described in Section 5.4.2.1.

From the graph displayed in Figure 5.3 it appears that the maximum amount of memory used in the experiments never excceeded 6.012 megabytes. One should bear in mind that this value represents the maximum *resident set size* of the process running the maximum flow algorithm, which includes program instructions, data and stack usage. As described in Section 5.1, the Linksys WRT54GS wireless router, on which memory tests were run, has 32 megabytes of RAM. The stock Linksys firware with the addition of an SSH daemon (and without running any of the P2PWNC software on it) has roughly 22 megabytes of

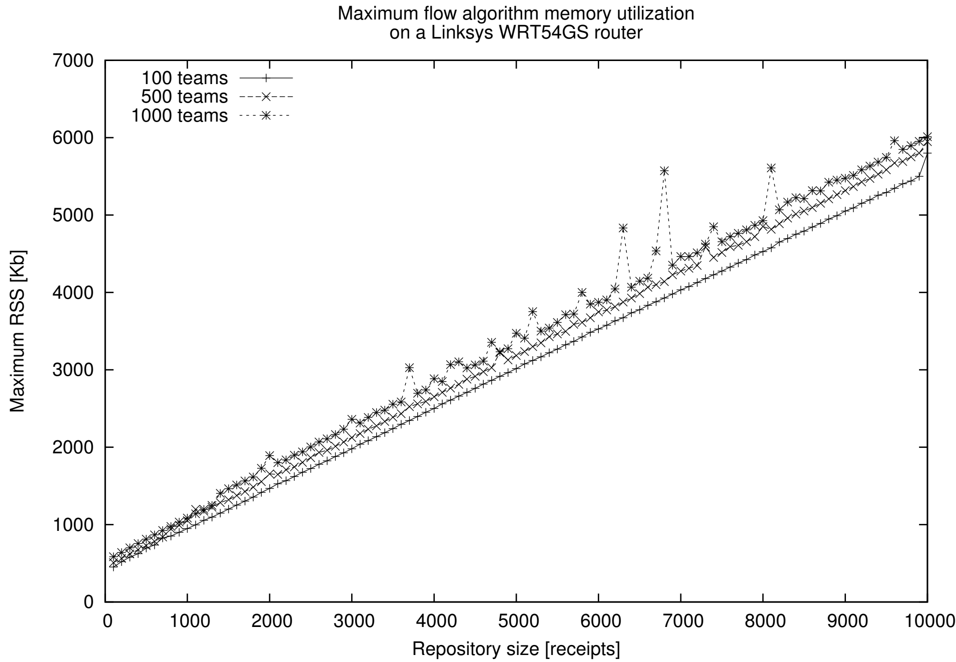

Figure 5.3: Maxflow memory usage on Linksys WRT54GS

free RAM as soon as it starts up. Thus, there is enough free memory to run the P2PWNC software, even in the case of a large receipt repository.

It should be noted that the tests presented here do not make any use of multithreading. The programs that have been run test only the performance of the receipt repository module. However, even in the presence of a multithreaded application, memory overhead while running the maximum flow algorithm will be similar to that presented in this section, since the maximum flow algorithm is run by a single thread at a time and the receipt repository is *locked* by the calling thread.

6 Future Work and Extensions

Although the design and implementation of the P2PWNC scheme, as of this writing, has advanced and there is a working implementation of the scheme in client and access point systems as well as in hosts acting as receipt repositories, there are still open issues concerning various system aspects.

6.1 Security Considerations

There are certain types of attacks that the system may be susceptible to. One of them has its root to the probabilistic nature of the decision algorithm used. As described in Section 4.2.4, access is granted with a probabity equal to a ratio (see, for example, Formula 4.2). In this case, if the enumerator of the fraction is smaller than the denominator, implying that the provider team "owes" less service to the consumer than the latter "owes" to it, this probability is less that 1. The potential consumer may send consecutive CONN messages to the access point, thus invoking the decision algorithm until its outcome is positive. For example, if the value of the ratio is 0.1, it will take roughly 10 attempts of the client until she is finally granted access.

To combat such malicious behavior, an access point may apply some heuristics. For example, the MAC address of the client may be cached so that consecutive attempts of the client to gain access may be discovered.

Apart from attacks that can be avoided by low-level implementation tricks, there may occur problems that are inherent to the families of decision algorithms used. For example, the score-based decision algorithm described in Section 4.2.4.2 is not collusion-proof. That is, teams may form groups that send false receipts to the TCA, reporting transactions that have never taken place, in order to raise their credits in the system and enjoy internet access without offering.

What is more, there are other secirity issues that need to be resolved, as they were

by design considered out of the scope of this work. For example, in the case of UPDT and RCPT messages exchanged between a member and its team server, securing communication and the client's membership with the team are not specified, nor have they been implemented.

6.2 Implementation Issues

As far as the implementation of the P2PWNC scheme is concerned, improvements can be achieved in issues concerning system funtionality and performance. Most of them are related to the receipt repository module. As to the module's performance, additional heuristics applied to the maximum flow algorithm may be developed so that its operation is sped up. Such is the *gap heuristic* [14], which may require some changes to the graph data structure used.

As to the module's functionality, taking a closer look to its implementation, one may see that it may prove to be a single point of failure for the P2PWNC scheme, especially in the case of the TCA. That is, in case of an error that will put the repository out of order (e.g. network or hardware failure), the entities affected my scale from a few access points and clients (decentralized case) to the whole P2PWNC system (centralized case). To eliminiate this threat, building a distributed receipt database on top of other entities may be necessary. For example, in the decentralized case, this database may be built on top of a team's access points. In the centralized case, there may be a grid of hosts that share the load of storing the receipts and running the decision algorithm.

Entities making use of a distributed receipt repository should have a transparent view of it. For example, consider the case of the repository built on top of a team's access points. Team members that wish to update their repository should sent their UPDT queries to a well-known IP address which will be the entry point to the team server. The query will then be processed by the distributed database module and the results will be sent back to the client in a transparent fashion. As it seems, this receipt repository should support the same operations as its non-distributed counterpart. That is, insertion, deletion and search operations should be provided. These may be achieved by a form of a distributed hash table, such as a *Chord* ring [27] access point arrangement (however, this structure is not appropriate for range queries).

The trickiest part of all, though is running the decision function in a decentralized manner. For this, a distributed version of the maximum flow algorithm needs to be developed.

A parallel and a distributed version of a push-relabel maximum flow algorithm is presented in [22].

The red-black tree data structure that was presented in Section 4.2.3.2 may also be extended. In particular, consider the case when a client visits an access point and sends a bulk of RCPT messages to improve the chances of being served. This can be a source for possible attacks. Namely, if the receipts sent by the client are fake and the access point's repository is full, these receipts may replace older but valid receipts. Thus, the transactions history maintained by the team server of the visited access point is polluted. A heuristic that can be applied to protect teams from such an attack is to finally accept this receipt set only in case they actually yield a higher maximum flow value from the consumer to the provider team.

In order to achieve this, a data structure that supports for versioning is required. Therefore, in the case of an attack, the repository will be reverted to the state prior to the insertion of the fake receipts. Data structures that can offer such versioning facilities are called *persistent*. A study of persistence in data structures and a method of adding such functionality to red-black trees is presented in [17].

6.3 Deployment Issues

The deployment issues discussed here mainly concern porting the P2PWNC to more platforms. As of this writing, there are Linux-based implementations of all software components. Also, there is a Java implementation of the P2PWNC client. However, the Java-based client has not yet been tested on devices such as PDAs.

Our Java implementation also lacks traffic measuring facilities. In general, this functionality is platform-dependent. The implementation of a traffic logging module for Linux kernels was presented in Section 4.3.4. In MS Windows environments, the *iphlpapi* library provides system calls which return management information per network interface. The amnount of traffic over an interface is part of this information. To take advantage of this functionality, either a Windows client should be written (possibly in *C#*) or this library should be employed by the existing Java client via the Java Native Interface. A similar approach should be followed in order to incorporate traffic logging capabilities in the Linux platform.

What is more, ECC functionality should be added to the Java-based client. Sun's Java Cryptography Extensions do not yet offer an ECDSA implementation, thus one should

resort to a third-party implementation, such as that provided by the Legion of the Bouncy Castle [11].

As to the access point software, this runs mainly on top on Linux-based wireless access points. In particular, it has been included in the firmware of the Linksys [4] WRT54GS 802.11g wireless router, which runs Linux 2.4. One of the drawbacks of the currently used firmware is that it runs on a read-only file system (Compressed ROM File System or *cramfs*). Migrating to a modern writable file system like JFFS2 (Journaling Flash File System) [28] would solve this problem.

Finally, porting the receipt repository module (and consequently the TCA and team server modules) in other platforms (such as Windows), is also possible, since it does not depend striclty on any Linux-specific features.

6.4 Evaluation Issues

The P2PWNC scheme's evaluation, either via simulations or via performance metrics has been based on the assumption that user mobility is random. For example, the maximum flow algorithm has been tested on randomly generated receipt graphs and random consumer-provider pairs. In practice, though, this may not usually be the case. Therefore, more user mobility models need to be studied and applied to the system's evaluation methods.

7 Conclusion

In this work, the design, reference implementation and performance evaluation of of the Peer-to-Peer Wireless Network Confederation protocol has been presented. Motivated by the proliferation of WLAN and broadband access technologies and aiming at providing ubiquitous internet access, a prototype WLAN roaming framework attempting to exploit under-utilized Wi-Fi resources has been developed.

The P2PWNC is a WLAN roaming scheme based on the peer-to-peer paradigm and built upon service reciprocity and peer autonomy. The system's users team up in small groups and pool their wireless access points. Each team needs to operate a number of public access points, while, at the same time, their members may be roaming around access points of other teams. Such teams are the peer entities of our scheme.

The design goals that were set for the P2PWNC scheme were those of simplicity, ease of deployment, reduction of management complexity and peer autonomy and self-organization. These goals were achieved by means of a simple, open protocol specification which provides for autonomous agent behaviour and relaxed service accounting, and a proof-of-concept implementation, running on top of resource constrained embedded devices as well as desktop PCs.

Two deployment scenarios have been studied and developed. First, there is a centralized operation mode, where history of transactions between peers is preserved by a trusted central repository. Second, there is a decentralized mode of operation, in which no trusted central authority exists and each peer keeps her own history of transactions.

In the P2PWNC scheme, users are identified by public/secret key pairs and membership with a team is asserted via certificates. Proofs of transactions (receipts) are also secured by cryptographic primitives. The P2PWNC public key infrastructure supports for both the RSA and the Elliptic Curve cryptosystems. Since heavy use of cryptographic functions takes place during the operation of the system, measurements have been performed so that the two cryptosystems can be compared as to their computational needs. Since the protocol

is aimed at resource constrained devices, test have been performed on two platforms; an embedded Linux-based wireless router and a more powerful AMD AthlonXP platform. These tests have proved the applicability of the emerging ECC cryptography standards for the needs of the P2PWNC scheme.

Given that service provisioning decisions are autonomous, the P2PWNC reference implementation software architecture provides for a pluggable decision algorithm module. Different users of the system may choose to implement their own means of deciding on whether access should be granted to a visiting roamer. In this work, a maximum-flow based decision algorithm has been implemented and evaluated in terms of processing time and memory usage. Performance measurements have shown that it can be time- and space-efficiently executed even on top of processing and memory constrained wireless access points.

In conclusion, it is our strong belief that the P2PWNC scheme can help in fueling the deployment of ubiquitous internet access and harness the unexploited Wi-Fi resources of underutilized residential wireless hotspots. Also, it is expected that its deployment will be assisted by the low cost of wireless equipment and the low risk and minor configuration effort that its use incurs for hotspot operators.

References

[1] *Athens Wireless Metropolitan Network*,
http://www.awmn.gr.

[2] *Dropbear SSH server and client*,
http://matt.ucc.asn.au/dropbear/dropbear.html.

[3] *Java Cryptography Extension (jce)*,
http://java.sun.com/products/jce/.

[4] *Linksys company web site*,
http://www.linksys.com.

[5] *Linspot*,
http://www.linspot.com/businessmodel.html.

[6] *Netfilter/iptables project homepage*,
http://www.netfilter.org.

[7] *Openssl: The open source toolkit for SSL/TLS*,
http://www.openssl.org.

[8] *OpenWRT project*,
http://openwrt.org.

[9] *Seattle Wireless Broadband Community Network*,
http://www.seattlewireless.net.

[10] *Speakeasy WiFi NetShare Service*,
http://www.speakeasy.net/netshare/.

[11] *The Legion of the Bouncy Castle*,
http://www.bouncycastle.org.

[12] G. M. Adel'son-Vel'skii and E. M. Landis, *An algorithm for the organization of information.*, Soviet Mathematics Doklady **3** (1962), 1259–1263.

[13] Kostas G. Anagnostakis and Michael B. Greenwald, *Exchange-based incentive mechanisms for peer-to-peer file sharing*, ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04) (Washington, DC, USA), IEEE Computer Society, 2004, pp. 524–533.

[14] Boris V. Cherkassky and Andrew V. Goldberg, *On implementing the push-relabel method for the maximum flow problem*, Algorithmica **19** (1997), no. 4, 390–410.

[15] D. Crocker, *Standard for the format of ARPA Internet text messages*, RFC 822, Internet Engineering Task Force, August 1982.

[16] P. Overell D. Crocker, Ed., *Augmented BNF for syntax specifications: ABNF*, RFC 2234, Internet Engineering Task Force, November 1997.

[17] J R Driscoll, N Sarnak, D D Sleator, and R E Tarjan, *Making data structures persistent*, STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing (New York, NY, USA), ACM Press, 1986, pp. 109–121.

[18] E. C. Efstathiou and G. C. Polyzos, *A self-managed scheme for free citywide wi-fi*, IEEE WoWMoM Autonomic Communications and Computing Workshop, June 2005.

[19] E. C. Efstathiou and G. C. Polyzos, *Self-organized peering of wireless lan hotspots*, European Transactions on Telecommunications, (Special Issue on Sefl-Organization in Mobile Networking) **16** (2005), no. 5.

[20] Standards for Efficient Cryptography Group, *SEC 1: Elliptic Curve Cryptography*, Available from http://www.secg.org, September 2000.

[21] Standards for Efficient Cryptography Group, *SEC 2: Recommended Elliptic Curve Domain Parameters*, Available from http://www.secg.org, September 2000.

[22] Andrew V. Goldberg and Robert E. Tarjan, *A new approach to the maximum-flow problem*, Journal of the ACM **35** (1988), no. 4, 921–940, Preliminary version in Proc. 18th Annual ACM Symposium on the Theory of Computing, pages 136–146, 1986.

[23] L. Guibas and R. Sedgewick, *A dichromatic framework for balanced trees*, IEEE-FOCS, Proc. FOCS Conf (1978).

[24] Ed. S. Josefsson, *The Base16, Base32, and Base64 Data Encodings*, RFC 3548, Internet Engineering Task Force, July 2003.

[25] Naouel Ben Salem, Jean-Pierre Hubaux, and Markus Jakobsson, *Reputation-based wi-fi deployment protocols and security analysis*, WMASH '04: Proceedings of the 2nd ACM international workshop on Wireless mobile applications and services on WLAN hotspots (New York, NY, USA), ACM Press, 2004, pp. 29–40.

[26] Daniel Dominic Sleator and Robert Endre Tarjan, *Self-adjusting binary search trees*, Journal of the ACM **32** (1985), no. 3, 652–686.

[27] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan, *Chord: A scalable Peer-To-Peer lookup service for internet applications*, Proceedings of the 2001 ACM SIGCOMM Conference, 2001, pp. 149–160.

[28] David Woodhouse, *JFFS: The Journalling Flash File System*, Ottawa Linux Symposium, RedHat Inc., 2001.