

PART I	NETWORK GAME PROGRAMMING	4
	BASIC OVERVIEW OF NETWORKS	5
1.1	<i>Network definition</i>	5
1.2	<i>Data transfer.....</i>	5
1.2.1	Introduction	5
1.2.2	Network protocols	6
1.2.2.1	Typical design of a protocol	6
1.2.2.2	Differences between protocols.....	7
1.2.2.3	Tasks of modern protocols.....	10
1.2.3	OSI reference model.....	11
1.2.4	TCP/IP protocol stack	13
1.2.4.1	Introduction.....	13
1.2.4.2	TCP/IP reference model	14
1.2.4.3	Ports, network addresses and classes	14
1.2.4.4	Transmission Control Protocol	18
1.2.4.5	User Datagram Protocol	19
1.2.4.6	Comparison between TCP and UDP	20
1.2.4.7	Conclusion for Game Programming	22
1.3	<i>What you should know</i>	23
	USING NETWORKS WITH RAKNET.....	24
2.1	<i>Why RakNet?</i>	24
2.2	<i>Major components</i>	26
2.2.1	Major features.....	26
2.3	<i>Setting up RakNet</i>	27
2.3.1	Windows.....	28
2.3.2	Linux/Unix	28
2.4	<i>First Example: Sending data over a network</i>	29
2.4.1	Creating the project for Windows	29
2.4.2	Creating the project for Linux/Unix	30
2.4.3	Developing the program	30
2.4.3.1	Peer-To-Peer	30
2.4.3.1.1	Listen-mode	31
2.4.3.1.2	Send-Mode.....	35
2.4.3.2	Client/Server	39
2.5	<i>Second Example: Peer-to-Peer chat.....</i>	40
2.5.1	Server.....	40
2.5.2	Clients.....	44
2.5.3	Extending both programs	48
2.5.3.1	Remote Procedure Calls	48
2.5.3.2	Ban list and kicking	50
2.5.3.3	Passwords	52

2.5.3.4	Encrypted connections	53
2.5.3.5	Timestamps	56
2.5.3.6	Statistics	58
2.6	<i>Third Example: Simple Game</i>	59
2.6.1	The gamefield	59
2.6.2	The client	65
2.6.2.1	The client's code	66
2.6.3	The server	67
2.6.4	Final words	69

Introduction

Today almost all games offer a multiplayer mode. Some are even developed solely for an intense multiplayer experience, such as Battlefield 2™ or Quake Wars: Enemy Territory™. “Multiplayer” in this case means game play via networking. Only a negligible number of games still support having several different people playing on a single computer.

The magic behind the multiplayer mode of a game consists of a mixture of (optical) illusions and approximation. The current network versions do not yet allow all of the game data to be sent to all the players involved. This fact also stacks with a problem of synchronisation, as the data never reaches the players immediately, but needs more or less time for the journey between systems.

A local simulation is commonly used to avoid or minimize both of these problems until the latest updates are available. The big challenge for a multiplayer game is to get the simulation running as smoothly as possible.

However, even before one can think about starting this simulation, it is necessary to select a functional style of architecture and then proceed to set up communication between all participating players. This is what the first two chapters are about. The remaining chapters will explain successful game simulation.

As you might already have noticed there is a whole lot of work waiting for you, so let's get it going...

Part I

NETWORK GAME

PROGRAMMING

- ★ **Chapter 1: Basic Overview of Networks**
- ★ **Chapter 2: Introducing RakNet**
- ★ **Chapter 3: Game Programming with RakNet**

Chapter 1

Basic Overview of Networks

- **How to send data over a network**
- **Client/Server and Peer-To-Peer**
- **TCP and UDP**

"You made three mistakes."

Riddick

1.1 Network definition

A network is a combination of at least two computers, which have the ability to communicate with each other. The basis for a network is a physical (e.g. cable or wireless) connection between the involved systems, which enables data transfer and conning and chiefly establishes rules systems (protocols).

1.2 Data transfer

1.2.1 Introduction

Now we have at least two computers, cables and interfaces (network cards) connected. This is the minimal physical requirement for a network.

I want to compare this state with linguistic communication: In biological sense this state is equivalent to existence of lung, phrenic, vocal chords and mouth. For an effective communication we need to fulfill some more requirements: The sounds, you create with these organs, need a syntax

and semantics, which your partner can understand. The sounds must be processed and encoded, so that the nerves can transport them to the right position of your brain. In the brain the signals are encoded and processed. Finally your dialogue partner awaits some kind of confirmation, that the message was understandable. Understandable means the message was complete and was received in a language, which your partner understands.

You find an example of such a “layered arrangement” in the communication between computers. The rules, which allow linguistic understanding, are called „protocols“. The responding partner in the communication process is determined from a network address. If more than two computers are involved in a dialogue, an access method controls sending and listening of messages.

1.2.2 Network protocols

The whole Internet is based on those network protocols. They are used for the loading of websites into your browser (http), sending of emails (smtp) or downloading files (ftp or http). There are many, many more out there, but these are the most important.

Computers and other digital machines use network protocols for communication on how data is transferred between them (a package of data is called “datagram”). Protocols are exact agreements between two (ore more) computers, on how data is transferred between them using a network infrastructure. Exchange of data between systems requires cooperation from different protocols. Every protocol is for a certain task. Many protocols mean a high complexity. To manage this complexity the protocols are organized in functional layers (often you will find only “layer”). In the context of this layered architecture, every protocol belongs to a certain layer and is responsible for a certain task. Protocols from higher layers rely on the services from lower layers. Together the protocols build so-called “protocol stacks” (or a protocol family).

1.2.2.1 Typical design of a protocol

Besides the user data, you want to send, every protocol needs additional data. In most protocols you will find a header in front of the user and sometimes a trailer after it. These headers contain several informations about the data package, e.g. sender and receiver, packet length or a checksum (to make sure, your data does not get corrupted). Often the protocols need determined package sequence to establish a connection between the computers. That is the so-called “overhead” of a protocol. Maybe you have seen this, when you have downloaded a file from a fast server and your download rate is around 80-90% of the theoretical maximum. The remaining traffic is the overhead of the used protocol (this would be FTP or HTTP here). In most cases you have to accept this overhead, as it provides additional features like security.

1.2.2.2 Differences between protocols

- If communication only takes place in one direction, this is called “simplex”. You have “half-duplex”, if your data flows alternatively in both directions. And “full-duplex”, if the data is send in both directions (sending & receiving) at the same time. If you have a LAN at home, you can check this setting in the driver details of your network.
- If all computers in a communication process are equal, this is called a “Peer-To-Peer” or symmetric communication. File transfer tools like BitTorrent use a Peer-To-Peer connection to exchange data. The second case is an asymmetric or “Client/Server” communication. Instead of communication you will often find “Client/Server-Architecture”, this is the same. A service provider (the “server”) handles request from different clients. Only clients can initiate a connection, a server will never contact a client. Every big network game uses client/server-architecture, because it has many advantages over the peer-to-peer-architecture. But this is part of another chapter later on. In general, the fastest computer with the best connection should act as the server, the rest of them as clients.
- When you need an acknowledgement for your request, you have a synchronal communication. Otherwise an asynchronal.

- In a package-oriented communication, the data is sent via messages or datagrams (short form of “data telegram”). On the other side you have “streaming”: Data is sent with a continuous data stream character wise. This is used in videoconferences over the Internet or if websites want to present a new game trailer so the users do not need to download it for watching.

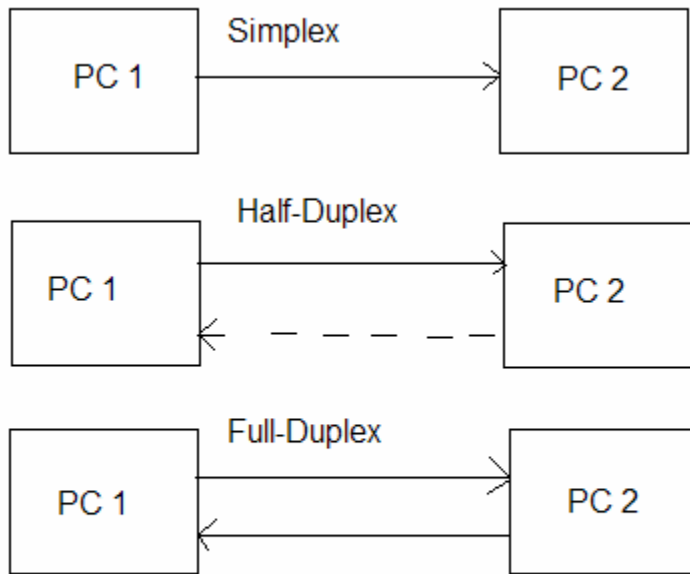


Figure 1: Communication directions

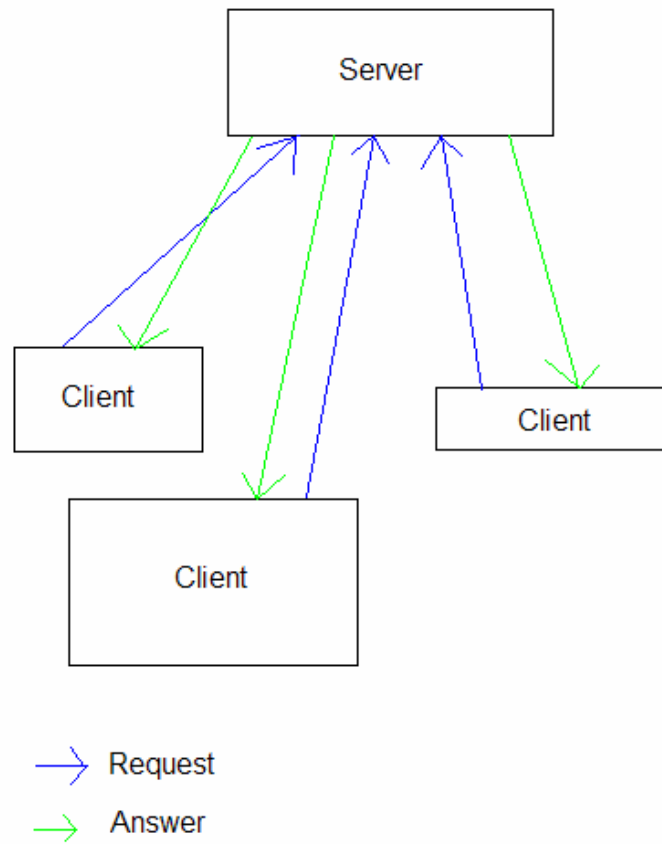


Figure 2: Client/Server Architecture

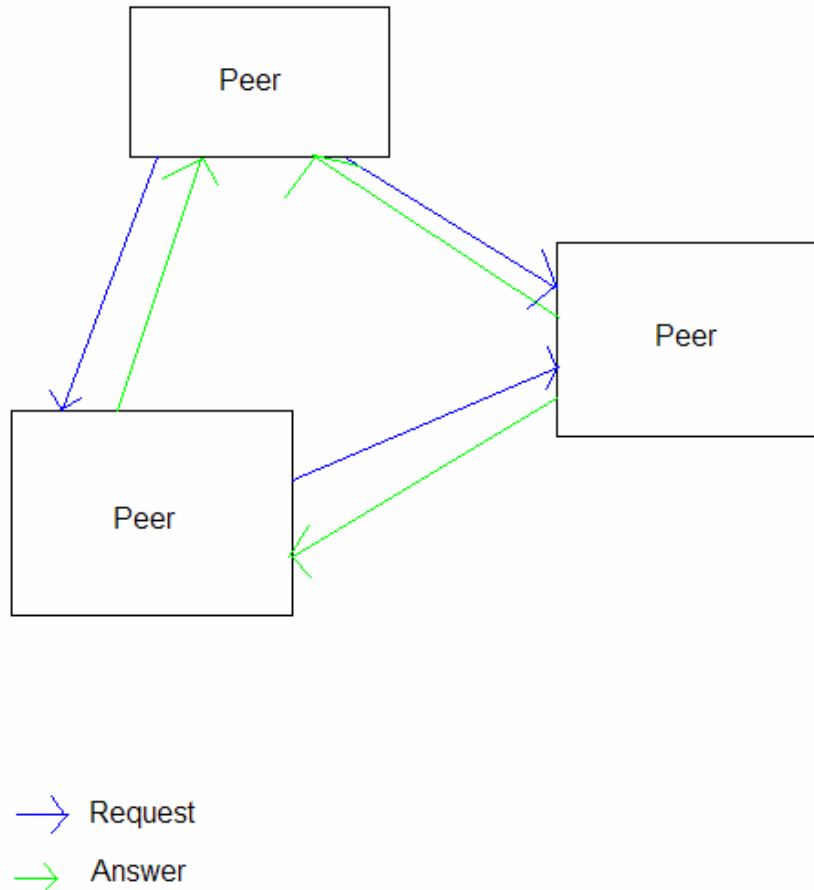


Figure 3: Possible Peer-to-Peer Architecture

1.2.2.3 Tasks of modern protocols

The most important thing here is the reliable sending of your data. Data loss on the way to the receiver is really problematic and should be avoided. If a package does not reach the destination, it should be resent until it does (or cancelled. But then the sender has to get a message that

the data can't be delivered). Reliable delivering includes correct delivering of packages. Your data becomes useless, if the transport way changes it. Imagine your players send highscores to your server and they get modified on their way. Your players will get really angry, if they get the wrong rank on the highscore list ;-)

Encryption adds an additional layer security. Many protocols support this. E.g. all orders or username/password combinations should be encrypted, so other people in the network can't read them.

1.2.3 OSI reference model

When you look at technical documents, network protocols are often described with a "reference model". Those models give a very abstract survey of the topic. I'll give you an example to explain what I mean.

Reference models are used very often in computer science. Those models give a very abstract view on a topic. Their main usage is to create some understanding of the underlying working logic. Protocol stacks are just one possibility; you will find them for example in Operating System descriptions.

For general networks the ISO (International Standardisation Organisation) has presented a model, the so-called "OSI reference model". When this model was developed, the aim was to connect the systems of different manufacturers - hence the name "Open System Interconnection", or just "OSI". The OSI reference model is often used to show a network protocol design and helps to understand the protocol's functions. A lot of protocols have used this model as a basis, but those protocols are mostly used in public communication (for example those protocols are used by Internet Service Providers). In private networks you will only find the TCP/IP-protocol stack (more about them later). These protocols are mainly used for connecting different networks - hence the name "Internet": inter network connection. Both the protocols from OSI and TCP/IP have a similar hierarchical model. But there are some important differences: OSI defines

which functionality each layer has to provide. TCP/IP is not that strict, which is the reason, why TCP/IP is more efficient than a pure OSI implementation. This freedom has its price: Every service has its own TCP/IP protocol; while with an OSI model every protocol offers a bigger performance. But let's have a look at the model now:

Layer number	Name
7	Application layer
6	Presentation layer
5	Session layer
4	Transport layer
3	Network layer
2	Data link layer
1	Physical layer

Every layer relies on the correct function of the layer beneath. If there is an error in layer 2, this error will be handed over until it reaches the application layer, where the user sees it. Now the layers in detail:

- Application layer: This layer makes a lot of functionality available, like sending mails, transferring files or surfing the web.
- Presentation layer: System-depending parts of data are converted into a standardized format; so all computers will understand the

message. Maybe you have ever heard anything of little-endian and big-endian? Transformation between the two happens in here. Also data compression and encryption is part of this layer.

- Session layer: This layer presents checkpoints. When a transmission is interrupted, the data does not have to be resent. It can start, where the last checkpoint was. Download managers work similar (but on another layer), you can pause a file transfer and continue it later.
- Transport layer: The transport layer offers an interface to the layers 5-7 for data transmission. So the layers above do not need to know something about the network structure, they just send their data.
- Network layer: For a package-oriented communication, this layer is responsible for the correct routing of the packages from sender to receiver. In most cases a direct connection between sender and receiver is not possible, so the packages need to travel over several so-called “nodes” (routers and backbones) to reach their aim.
- Data link layer: This layer makes sure, that your data reaches the destination unaltered.
- Physical layer: The last layer manages physical connections and transmission of your data between computers.

Every layer adds additional data to your data: headers and trailers. Both of them contain information for the destination layer of the same height level. How much overhead you have depends on the used protocol. In general: the more features, the more overhead.

1.2.4 TCP/IP protocol stack

1.2.4.1 Introduction

At the beginning of the 1980's the original ARPANET was grown large and the old protocols weren't good enough. So the *Advanced Research Project*

Agency decided to develop and use new protocols. So the “Internet Protocol” stack was born, or just “IP” in short. Nowadays the whole Internet is based on those protocols. The main protocols are HTTP, FTP, TCP, UDP, IPv4 and IPv6. There are many more, but they are rarely used.

1.2.4.2 TCP/IP reference model

The TCP/IP reference model is simpler than the OSI model. Its main task is data exchange over the borders of local networks (“Internetworking”). The protocols are reliable for sending data over several nodes and making connections over those nodes. The physical connections are not defined, you can choose between wires, fiber optic or radio.

IP layer	~ OSI-layer	Example protocol
Application layer	5-7	HTTP, FTP, SMTP
Transport layer	4	TCP, UDP
Inter-network layer	3	IPv4, IPv6
Network layer	1-2	

The Application layer contains all protocols, which work together with network applications. The transport layer creates point-to-point connections. The TCP protocol is the most important here; it creates connections between two communication partners. With this connection, both can transmit data securely. The last layer determines the next node for a packet.

1.2.4.3 Ports, network addresses and classes

In order to understand the TCP/IP addressing, you must know, how the Internet is organized: It consists of sub networks, which result all together in the Internet. Routers and gateways are machines dedicated to connect those sub networks to each other. If you connect to the Internet with an individual computer, the computer belongs to a sub network, which is owned by the provider. If you attach your LAN (local area network, e.g. your connected computers at home form a small LAN) to the Internet in such a way, this LAN becomes a subnetwork of the provider's network. You need only one IP address, in order to bind up to 254 computers to the Internet. Sub networks are identified over the IP address. IP addresses have always the form x.x.x.x, whereby x is a number between 0 and 255. Each IP address is split into several parts: The first part describes the net, the remainder the computer in the net. If n is the net and r of the computers, the IP can look for example in such a way: n.r.r.r or n.n.r.r or n.n.n.r. On the basis of these three examples the class of the net also can be determined: n.r.r.r is a class A net. There is only one byte for the number of nets in this address; therefore there can be only maximally 256 class A nets. The values 0 and 255 are special cases and can't be used - thus 254 possibilities remain. Hence there are worldwide only 254 class A networks. Each of these networks can contain however many computers, which are marked by the remaining three bytes: The number of computers in a class A network computes itself in such a way: $256 \cdot 256 \cdot 256$ (minus the special cases). Such large (and also rather expensive) networks are only assigned to very large companies. The next smaller net is the class B network: Here the first two bytes are used for addressing the sub network. Finally the smallest is the class C net, with which the first three bytes mark the sub network and one byte remains for the computer identifications. Networks are further partitioned in order to distribute the network addresses better. An important technology are subnet masks. The subnet mask consists - like an IP address - of four bytes and specifies the size of the sub network. Within a subnet the computers can exchange data without a connection to a router or as a gateway.

Ports are address components to assign data packets to the right service/protocol. This concept is implemented e.g. in TCP, UDP and SCTP. In these three protocols the port number is 16 bits large, i.e. it can take values from 0 to 65535. Certain applications use port numbers, which

are publicly known and assigned by the IANA (“Internet Assigned Numbers Authority”, www.iana.org). They are usually located between 0 and 1023, and are named „well-known ports“. Between port 1024 and 49151 are the „registered ports“. If necessary these can be registered by application manufacturers for own protocols, similarly as domain names, with the purpose that communication runs in the net ends in chaos, if new applications send packets through the Internet. Remaining ports between 49151 and 65535 are „dynamic and/or private“ ports. These can be used variable, since they are not registered and so that are associated to no application.

It is not possible to differentiate multiple connections with the same port number on a computer. So every program, that needs to send data over the network needs its own unique port number. The only exceptions are servers, where data is sent and received on a certain port. Here only clients need to have their own port number, so the server can send them their individual data.

Example ports and applications:

Port number	Service	Description
20	FTP Data	File transfer (data transfer from server to the client)
21	FTP	File transfer (initiation of a session through the client)
22	SSH	Secure Shell

25	SMTP	E-Mail transmission
53	DNS	A DNS server translates a domain name into a IP adress
80	HTTP	Web server
110	POP3	E-Mail pick up
119	NNTP	Usenet
143	IMAP	E-Mail access and administration
443	HTTPS	Web server with SSL encryption
666	DOOM	Doom game server
1836	MSN Messenger	MSN Instant Messaging
3306	MySQL	MySQL
5050	YIM	Yahoo! Instant Messaging
5190	ICQ	ICQ Instant Messaging

5222	Jabber	Jabber Instant Messaging
6667-6669	IRC	Internet Relay Chat
8767	TeamSpeak	TeamSpeak VoiceChat

1.2.4.4 Transmission Control Protocol

The „Transmission Control Protocol“ (TCP) was developed by Robert E. Kahn and Vinton G. Cerf. TCP was part of the research work they began in the year 1973 and which took them several years. The first standardisation of TCP therefore didn't take place until the year 1981 as RFC 793.

TCP creates a virtual channel between two end points (sockets) of a network connection. Data can be transmitted into both directions on this channel. In most cases TCP is based on the IP protocol. It is settled in layer 4 of the OSI reference model. TCP was developed, in order to deal with the uncertainty of the OSI layers, which are under it. It therefore examines the integrity of the data with a checksum in the package head and guarantees the correct order of the received data. The transmitter repeats a sending of packages, if no confirmation/acknowledgement arrives within a certain time interval. The data of the packages are joined by the receiver in a buffer in the correct order to a data stream. The transmitter can only get to know, whether the receipt of data of the receiver was correctly acknowledged or not. The data transfer can be perturbed, retarded naturally at any time after the "connection establishment" or interrupted completely. The transmission system then runs into a timeout. The "connection establishment", which is transacted first, thus represents no guarantee for a following safe transmission.

In an Ethernet environment (which is the preferred connection method for small LANs) the user data can have a maximum size of 1500 bytes. But you must take care of the protocol headers. Those are 20 bytes for TCP and IP in each case. 1460 bytes remain. To send, for example 100Kb of data, the data is segmented to fit into that 1460 bytes and send. Each segment is identified with a certain number. The receiver uses this number to recreate the 100Kb data in the right order. You don't need to take care of segmentation, the network card's driver does it for you. For every received package the sender has to get an acknowledgement from the receiver. This makes sure, that your data reaches the desired aim. If the sender gets no acknowledgement, it repeats the data package until it got the acknowledgement.

1.2.4.5 User Datagram Protocol

The „User Datagram Protocol“ (UDP) is part of the TCP/IP protocol stack, too. Unlike TCP it is not made for reliability, but for speed. With UDP you can send data packages. But here you need to take care of the segmentation and the correct composition of the packets. Packets can get lost on their way to the receiver and you need to detect it and react on it. In comparison to TCP you lose some security and comfort, but you get a higher transfer speed and lower response times. And that is what we need in games. So many games use a self made protocol, based on UDP. Additional speed brings the fact that UDP does not use connections. So connecting to a game server works much faster than with TCP. That means that a connection is not established, but that the data packages are sent immediately to the receiving station. It is not guaranteed that a once sent package arrives or that packages arrive in the same order, in which they were sent; an acknowledgement is not intended. The communication partners cannot determine thus whether packages are lost or delayed. Also a duplication of packages can occur. An application, which uses UDP, must be insensitive and re-sorted packages to be insensitive or even contain appropriate corrective measures.

Since before beginning of transmission a connection does not only have to be developed, a host can begin with data exchange faster. This particularly carries weight with applications, in which only small data is exchanged. Besides the unsecured transmission also offers the advantage of small transmission delay fluctuations: if a package is lost within a TCP connection, then it must be requested again. This needs time, the transmission duration can therefore vary, which is bad for games: The players “jump” around or disappear and suddenly appear somewhere else. With VoIP e.g. it would come to sudden misfires and/or the rendition buffers would have to be put on more largely. Therefore such applications are set to UDP. Lost packages in here do not bring the entire transmission in coming to a hold.

1.2.4.6 Comparison between TCP and UDP

Description	UDP	TCP
General Description	Simple, high-speed, low-functionality “wrapper” that interfaces applications to the network layer and does little else.	Full-featured protocol that allows applications to send data reliably without worrying about network layer issues.
Protocol Connection Setup	Connectionless; data is sent without setup.	Connection-oriented; connection must be established prior to transmission.
Data Interface To Application	Message-based; data is sent in discrete packages by	Stream-based; data is sent by the application with no

	the application.	particular structure.
Reliability and Acknowledgments	Unreliable, best-effort delivery without acknowledgments.	Reliable delivery of messages; all data is acknowledged.
Retransmissions	Not performed. Application must detect lost data and retransmit if needed.	Delivery of all data is managed, and lost data is retransmitted automatically.
Features Provided to Manage Flow of Data	None	Flow control using sliding windows; window size adjustment heuristics; congestion avoidance algorithms.
Overhead	Low	Low, but higher than UDP
Transmission Speed	Very High	High, but not as high as UDP
Data Quantity Suitability	Small to moderate amounts of data (up to a few hundred bytes)	Small to very large amounts of data (up to gigabytes)

Types of Applications That Use The Protocol	Applications in which data delivery speed matters more than completeness, in which small amounts of data are sent; or in which multicast/broadcast are used.	Most protocols and applications sending data that must be received reliably, including most file and message transfer protocols.
Well-Known Applications and Protocols	Multimedia applications, games, DNS, BOOTP, DHCP, TFTP, SNMP, RIP, NFS (early versions)	FTP, Telnet, SMTP, DNS, HTTP, POP, NNTP, IMAP, BGP, IRC, NFS (later versions)

1.2.4.7 Conclusion for Game Programming

Almost all games rely on fast data transmission and/or low transmission times (often referred to as “ping”). So if you have a real-time game, your only way to get a good network experience is with UDP. TCP is used too, but to a lesser extent. You will find it in round-based games like FreeCiv or in Internet card games. All other games use a specialised protocol, based on UDP. These protocols improve the data reliability at the cost of minimal speed loss and are therefore way faster than TCP.

We will focus on UDP in this course, but not on the low level details of protocol design and implementation. As this is a game-programming course, our aims are games. RakNet will handle the dirty details of UDP.

1.3 What you should know

- TCP and UDP are part of the TCP/IP protocol stack.
- TCP is connection based, UDP is connectionless,
- TCP connections are established between sockets.
- TCP give a small guarantee, that your data reaches the aim. UDP packets are like “fire and forget”.
- Most games use UDP for the network part with a protocol built on top of UDP.
- In client/server-architecture the server handles all requests from the clients.
- In a peer-to-peer-architecture all clients are connected to each other.
- Data is sent with packets. Each packet has a maximum size, the so-called MTU (“maximum transfer unit”). Common values for a LAN are around 1400-1500. When ADSL was introduced, many users had to lower this value, because many websites (GMX for example) blocked packages with a MTU > 1452.
- To specify a certain computer, an IP address is used. This address consists of four bytes. To identify the application a data package is sent to, ports are used.

Chapter 2

Using Networks with RakNet

- **Why RakNet**
- **Overview of RakNet**
- **Using RakNet**

*"First, you took the job."
Riddick*

2.1 Why RakNet?

Every multiplayer game needs its own network protocol to send and receive data. A lot of games use UDP as underlying layer, because it is fast. Maybe you have suspected that developing your own network engine is a hard task. You need to take care of connection establishment, data sending, keep ping and bandwidth usage low, etc. This course is about network game programming and not network protocol design, thus Raknet will handle all the details of sending and receiving network data. RakNet is an advanced networking API that provides services for and over network sockets. To do this, it provides a layer over UDP data packets. It allows any application to communicate with other applications that also use it, whether that is on the same computer, over a LAN, or over the Internet. Although RakNet can be used for any networked application, it was developed with a focus on online gaming and provides extra functionality to facilitate the programming of online games as well as being programmed to address the most common needs of online games. This is, was RakNet does for you:

- RakNet can automatically resend packets that did not arrive.

- RakNet can automatically order packets that arrived out of order.
- RakNet protects data that is transmitted and will inform the programmer if that data was externally changed.
- RakNet provides a fast, simple, connection layer that blocks unauthorized transmission.
- RakNet transparently handles network issues such as flow control and aggregation.

RakNet does all these things very efficiently. If it didn't, it wouldn't be of much use ;-)

But RakNet can do some more things than just sending data over a network:

- You can synchronize class instances with a few lines of code. You do not need to send update packages for this.
- RakNet supports real time voice communication.
- With RakNet you can easily run a master server for the clients to find games.
- Use Remote Procedure Calls (RPC), which allow you to call functions on a remote computer.
- RakNet provides different statistics like ping, packetloss, bytes sent, bytes received, packets sent, packets received and many more.
- Synchronize the time and random number generator for all connected computers.

All these features are easy to use and implement in your game. This makes RakNet a perfect choice for any multiplayer game.

2.2 Major components

The main components of RakNet are peer, server and client. Server and client are both specialised peer objects with extra features.

When a peer is created, it creates an update thread, which runs in either a poll or update state. The poll state checks to see if the `Receive()`-function is called on a regular basis. If so, the thread sleeps until the next poll time. Otherwise, it enters the update-state in which case the thread handles normal traffic communications.

When you or the system sends a message it is parsed, compressed if compression is used, and sent to the reliability layer. The reliability layer converts the message into one or more `InternalPacket`-objects. These objects are saved in one of several lists, depending on their priority level. Every message can get a certain priority, so you can send important game code updates like character moving or death first and chat messages last.

When a message arrives it is decrypted, validated, and parsed (based on ids) into individual messages. If a message was split and all pieces have arrived, the original message is reassembled. All messages, and messages that originated from the user, will be put into a `Packet` struct, which is allocated from a pool, and will be saved in a mutexed queue. Messages that are intended for the user at this time will be sent to `RakPeer`, which will decompress it if compression is activated. Certain system messages will be handled immediately such as lost connection notifications. When `Receive()` is called, the queue is popped and the popped packet is parsed. Certain kinds of packets are not returned to the user, such as RPC packets, which will instead call a function. If a packet is returned, sometimes the packet is intended for another user-level system such as distributed network objects, `RakVoice`, or the master server.

2.2.1 Major features

All these features are supported by the three core classes (peer, server and client):

- Remote Procedure Calls: RPCs allow you to call functions on other systems. This is much easier as sending a specific message.
- Timestamping: Timestamps are saved in every packet. Hence you know, when an event happened on another system.
- Data compression: RakNet can analyze network traffic and apply compression to it. The analysis helps to decide, if compression is useful. As with every compression, this adds some CPU work, but saves bandwidth.
- Distributed Network Objects: The *DistributedNetworkObject* is a class you can derive from that will give your classes the ability to automatically propagate them over the network and to synchronize member variables. It derives from *NetworkObject*, which provides a unique identifier for all your instanced classes so you can refer to them over the network. It assumes you are working in a client / server environment and requires registration of your client and / or server with the *DistributedNetworkObjectManager* Singleton.
- Autopatcher: The autopatcher is a class that manages the copying of missing or changed files between two or more systems. It handles transferring files, compressing transferred files, security, and file operations.
- IO completion ports: With a IO completion port, the network card can read and write from/to memory. This improves performance in most situations. Because this is a MS Windows(NT, 2000 and XP)-only feature, I will not use it in this course.

2.3 Setting up RakNet

RakNet is available for Windows, Unix/Linux and Mac OS X. It supports the common 32Bit CPU architectures and with some patches the new Athlon

64 CPUs in native 64Bit mode (you can use RakNet in 32Bit on any Athlon 64 of course). You can get the RakNet library with three different licenses:

- GPL License: You can use the library in any GPL project. There are no restrictions and it is freely distributable.
- Shareware License: This license is compatible with other proprietary licenses and closed. You can get a shareware license for free, when you apply for one here:
<http://www.rakkarsoft.com/LicenseApplication.html>
- Commercial license: This is a special license for commercial games. It goes beyond the shareware license. You can apply for a free commercial license.

We will use the GPL version here. If you need another license, please apply for one and use the GPL license until you get the new.

2.3.1 Windows

First step is to download the source archive from www.rakkarsoft.com. Scroll the page down to the Download section and get the GPL Version (or use this link: <http://www.rakkarsoft.com/raknet/downloads/RakNet.zip>). After downloading, extract the archive into any directory. Open the appropriate workspace for your IDE: RakNet.dev for Bloodshet Dev-cpp, RakNet.dsw for MS VC++ 6 and RakNet.sln for MS VC++ 7 and above. Compile at least the following projects: DLL, LibStatic, RakVoice, RakVoiceDLL. Now you need to setup your compiler: First you need to copy the Include/-directory, name it "raknet". Open an instance of your IDE. For MS VC++ go to Extra->Options->Projects->VC++-Directories. Choose "include files" on the right side. Select an empty line from the list in the middle, open the "Directory Search" and select RakNet's root directory. Choose "library files" at the right side. Again, go to the middle and select "lib/"-directory in RakNet's root. That's it!

2.3.2 Linux/Unix

First step is to download the source archive from www.rakkarsoft.com. Scroll the page down to the Download section and get the GPL Version (or use this link: <http://www.rakkarsoft.com/raknet/downloads/RakNet.zip>). After downloading, extract the archive into any directory. Open “makefile.defs” in the directory and change the variables, so that they fit your needs. Make sure you have kernel headers or the source installed, they are required. Run “make” and “make install” (as root) to install the headers and library. When linking your programs, make sure you link RakNet and pthread library!

Note for Linux/Unix Users: Please send me a message, if you do not use Windows. I need to adjust the assignments for you. This is not a problem; you will get the same tasks and points as the rest.

When I compiled RakNet on my Debian unstable box I got some errors with lower/upper cases. If you have them, fix the Makefile so it corresponds to your directory layout. Or send me a message and I will send you my fixed Makefiles.

2.4 First Example: Sending data over a network

After this long theoretical part we finally reach the first practice now. In this chapter you will be learning the basics of sending data with RakNet. The sample program will take a string given via command line and send it to another computer. The program needs two modes: sending a string and receiving + displaying it. In the next chapter we will be extending this program to a client/server chat system.

2.4.1 Creating the project for Windows

Open your IDE and go to File -> New project and select “win32 console program”. Choose name and directory. Make sure, you select “Empty project” in the next window! Now add the first source file, I named mine

“main.cpp”. RakNet requires threading support, so go to Project -> Properties -> C/C++/Code Creating -> Runtime Library: Choose “Multithreaded Debug” for your debug build and “Multithreaded” for your release build. You need to link against the RakNet libraries, therefore select Project -> Properties -> Linker -> Input and add “*RakNetLibStaticDebug.lib ws2_32.lib*” in the “Additional dependencies”-field. BloodShed users have to link with “-lraknet -lpthreads” and set there include dir to the same directory as MS VC++ users. Do not forget to create the main function.

2.4.2 Creating the project for Linux/Unix

Just open a new source file with your preferred editor or IDE and name it. After your compile, link with “-lraknet -lpthreads “. That’s all. But do not forget to create the main function.

2.4.3 Developing the program

As mentioned further above, a network is organized as Peer-To-Peer or Client/Server. The first network example will show you how to send with a peer-to-peer system and a client/server system. After that, the later program will extended to show some of the advanced features of RakNet.

2.4.3.1 Peer-To-Peer

To send data with RakNet with a peer-to-peer system, you always need an instance of RakPeerInterface for every peer. Such an instance is created with the RakNetworkFactory-Singleton. To use both classes, you need the right headers:

```
#include <raknet/RakNetworkFactory.h>
#include <raknet/RakPeerInterface.h>
#include <raknet/PacketEnumerations.h>
```

For network connections, we always need a port on the local system and the remote system (remember: between both ports, a “virtual”

connection/data tunnel is established). To simplify matters, these ports are constant and defined:

```
const int iRemotePort=60001;  
const int iLocalPort=60002;
```

In the main function, the program reads the operation mode into a string. Operation mode can be “send” or “listen”. In listen-mode the program waits for an incoming connection and displays the received data. If the user selects “send”, the program asks for a remote IP address (where to send the data) and the message. Now it comes to initializing and setting up the network interfaces.

2.4.3.1.1 Listen-mode

Every time you want to send data with RakNet, you need a network interface. This is created with RakNetworkInterface:

```
RakPeerInterface *receiver = RakNetworkFactory::GetRakPeerInterface();
```

After creating the interface, it must be initialized:

```
receiver->Initialize(100, iRemotePort, 0);
```

The first parameter limits the number of connections for this peer. Next is the port, where the listener waits for connections. The last parameter specifies a sleep time for the receiving thread: RakNet uses a lot of threads; keep that in mind. One of these threads is the listener thread: Network packets are received and preprocessed here. The sleep time, how much a thread is inactive/sleeps, before it can continue work. The higher the value, the less CPU time the network thread will sleep. If you set this value too high, the thread will sleep too long and there is a good chance, that you dont receive all packets. So be careful. For games a good value is '0', so all packets receive their target. The next function call limits the number of incoming, parallel connections:

```
receiver->SetMaximumIncomingConnections(100);
```

Now, the main part of message processing: the receive loop:

```
while(!done)
{
    Packet *packet = receiver->Receive();
    while(packet)
    {
        switch(packet->data[0])
        {
            case ID_REMOTE_DISCONNECT_NOTIFICATION:
                cout << "A client has disconnected" << endl;
                break;
            case ID_REMOTE_CONNECTION_LOST:
                cout << "A client has lost the connection" << endl;
                break;
            case ID_REMOTE_NEW_INCOMING_CONNECTION:
                cout << "A client has connected" << endl;
                break;
            case ID_CONNECTION_REQUEST_ACCEPTED:
                cout << "Connection request has been accepted" << endl;
                break;
            case ID_NEW_INCOMING_CONNECTION:
                cout << "Incoming connection from " << receiver-
>PlayerIDToDottedIP(packet->playerId) << endl;
                break;
            case ID_NO_FREE_INCOMING_CONNECTIONS:
                cout << "Server is full" << endl;
                break;
            case ID_DISCONNECT_NOTIFICATION:
                cout << "Somebody has disconnected: " << receiver-
>PlayerIDToDottedIP(packet->playerId) << endl;
                break;
            case ID_CONNECTION_LOST:
                cout << "Connection lost" << endl;
                break;
            case ID_RESERVED9+1:
                {
                    cout << "Received reserved package: " << packet->length <<
endl;

                    bitStream.Reset();
                    bitStream.Write((char*) packet->data, packet->length);
                    // skip identifier
                    bitStream.IgnoreBits(8);
                    // read length: every byte/char has 8 bits, so convert the
count

                    unsigned int length = (unsigned int) (packet->bitSize >> 3);
                    length--; // ignore the ID byte at the beginning
                    cout << "\tlength: " << length << endl;
                }
            }
        }
    }
}
```

```

        // Now get the send string
        char *d = new char[length];
        bitStream.Read(d, length);

        cout << "\tdata: \"" << d << "\" << endl;
        if(strcmp(d, "quit") == 0)
            done = true;
        delete []d;
    }
    break;
case ID_RECEIVED_STATIC_DATA:
    {
        cout << "Received static data: " << flush;
        // RakNet sends empty packets on connecting
        if(packet->length == 1)
            cout << "contains no data" << endl;
    }
    break;
default:
    cout << "Message identifier: " << (int)packet->data[0] << endl;
    break;
}
receiver->DeallocatePacket(packet);
packet = receiver->Receive();
}
}

```

This loop is similar to the event loop in a windows program. At first we need to look, if a packet is waiting:

```

Packet *packet = receiver->Receive();
while(packet)

```

If not, you can skip the loop; there is nothing to do. As long as there are packets waiting, we want to process them. The packet structure looks like this:

```

struct Packet
{
    PlayerID playerId;
    unsigned long length;
    unsigned long bitSize;
    char* data;
};

```

The first byte (8 bit) data define the type of the packet. There are approx. 30 predefined values you cannot use for your own packets. Have a look at

them in `raknet/PacketEnumerations.h`! The rest of the numbers can be used for your own packet types. The sample program only identifies the most important types like incoming connection or disconnection. Length and bitsize tell you the size of the data. "Length" is obsolete and only kept there for compatibility issues. Don't use it; use `bitSize` instead. Every packet contains a player id; this number is a unique number and identifies every player/peer. The id consists of the IP address and the port of the peer. To translate this id into an IP address you can read, the `RakPeerInterface` has a function to return the address as string:

```
receiver->PlayerIDToDottedIP(packet->playerId)
```

When you want to send your own data, you need to define your own packet type. The sample program just adds a "one" to the last predefined type: `ID_RESERVED9+1`

Data is sent via a `BitStream` or a `char*`-array. For a `BitStream` (as in this example) you do the following: You write all your data to a bitstream and send it with the network interface (I will describe how you do this in a moment). To decode your data, you create a new bitstream or reset a used one. Now you write the received packet data to the bitstream:

```
RakNet::BitStream bitStream;  
bitStream.Write((char*) packet->data, packet->length);
```

Now your data is in the bitstream and you can read specific values from it. We only want to print the received string without the identifier, so it is skipped: `bitStream.IgnoreBits(8);`

The length of the user data in a packet is saved in the attribute "bitSize". To get the length of the string, you divide the bit size by 8 (or shift it right by 3 positions, which is a lot faster!) and subtract one (for the identifier). Allocate some memory for the string and read it from the bitstream:

```
unsigned int length = (unsigned int) (packet->bitSize >> 3);  
length--;  
char *d = new char[length];  
bitStream.Read(d, length);
```

Finally you have to pop the packet from the packet queue and check if there is another one:

```
receiver->DeallocatePacket(packet);  
packet = receiver->Receive();
```

Dont forget to disconnect and destroy the RakPeerInterface at the end:

```
receiver->Disconnect(1000);  
RakNetworkFactory::DestroyRakPeerInterface(receiver);
```

2.4.3.1.2 Send-Mode

To send data you just need a few lines of code. At first you have to get an interface and initialize it. This time you only need one connection:

```
RakPeerInterface *sender = RakNetworkFactory::GetRakPeerInterface();  
sender->Initialize(1, iLocalPort, 0);
```

To send you have to perform three steps: Connect, send, disconnect. Ok, first the connection:

```
sender->Connect((char*)strRemoteIP.c_str(), iRemotePort, 0, 0)
```

As RakNet uses multithreading, the connect()-function only sends a request for a connection. If it returns “true”, that only means that the request was send. This is very important! To make sure you have a connection, you must wait for a packet with the id “ID_CONNECTION_REQUEST_ACCEPTED”. To do this, use a while()-loop to wait the packet from the remote peer: Normally you would implement a time-out for this; this timeout mechanism is missing here:

```
bool hasConnection = false;  
while(!hasConnection)  
{  
    Packet* p = sender->Receive();  
    if(p)  
    {  
        if(p->data[0] == ID_CONNECTION_REQUEST_ACCEPTED)  
            hasConnection = true;  
    }  
}
```

```

        sender->DeallocatePacket(p);
    }
}

```

After that particular package, some more packages can be received in the local packet queue and you have to fetch them:

```

Packet *p = sender->Receive();
while(p)
{
    sender->DeallocatePacket(p);
    p = sender->Receive();
}

```

You can send your data either with a bitstream or cast your data structure to a char* array and send that. You can cast your structure to a character array, write it to a bitstream and send that. The advantage of creating a structure and casting is: it is very easy to change the structure and to see, which data you are actually sending. Since both the sender and the recipient can share the same source file defining the structure, you avoid casting mistakes. There is also no risk of getting the data out of order, or using the wrong types. The disadvantage of creating a structure is that you often have to change and recompile many files to do so. You also lose the compression, which you can automatically perform with the bitstream class. The advantage of using a bitstream is, that you do not have to change any external files to use it. Simply create the bitstream, write the data you want in whatever order you want, and send it. You can use the 'compressed' version of the read and write methods to write using fewer bits. You can also write out data dynamically, writing certain values if certain conditions are true. The disadvantage of a bitstream is: you are now susceptible to make mistakes. You can read data in a way that does not complement how you wrote it - the wrong order, a wrong data type, or other mistakes. I prefer a bitstream therefore the sample programs will use them. Feel free to use the array version. Enough theory; back to the sample program. We can prepare the BitStream to send data, now. If you dont use a new bitstream, reset the old first. The first byte of every packet data is the identifier. As stated above, I added one to the last predefined identifier for a new ID. Do not forget to cast it to an unsigned char/byte:

```

RakNet::BitStream bitStream;

```

```
bitStream.Write( (unsigned char) (ID_RESERVED9+1) );
```

Directly after the id byte follows your user data. In this case, it is the string to send. Here, RakNet offers another utility class: StringCompressor. When you use this class, your strings are automatically compressed and StringCompressor handles the '\0'-terminator for you. But you should be careful with compression: This is a “huffman compression”, so small texts can increase the data size instead of decreasing it! Huffman compression needs a dictionary for decompressing. For small texts this can increase the data size by ca. 40%, so choose well, when you use it. In a normal chat/game there isn't much text at once (by one player), so compression only increases CPU load. I have implemented both options (with stringcompressor and with normal bitstream) in the sample program. For a normal bitstream you write your data directly to the bitstream:

```
bitStream.Write((char*)strData.c_str(), strData.length()+1 );
```

The first parameter is the actual data, casted to a char*. The second parameter is the length of that. Be aware of the '\0'-terminator at the end of every string! Dont forget to add it in the size. There are some overloaded functions to send single characters, integers and floats. Now you can send the data:

```
sender->Send(&bitStream, HIGH_PRIORITY, RELIABLE_ORDERED, 0,  
           UNASSIGNED_PLAYER_ID, true);
```

The first parameter is the bitstream with your data; the second defines the priority of that data. The lower the priority, the later your data is sent. You can choose between three priority levels: HIGH_PRIORITY, MEDIUM_PRIORITY and LOW_PRIORITY. High priority packets will be sent before medium and medium before low. The third parameter defines, how your data is sent. It can have one of five values: UNRELIABLE, UNRELIABLE_SEQUENCED, RELIABLE, RELIABLE_ORDERED, RELIABLE_SEQUENCED. The best way to explain matters is doing it with an example: Imagine you send the data packets A,B,C,D,E and F. This is what you get for every mode:

- UNRELIABLE: C, F, A

- UNRELIABLE_SEQUENCED: D
- RELIABLE: F, D, C, A, B, E
- RELIABLE_ORDERED: A, B, C, D, E, F
- RELIABLE_SEQUENCED: C, D, E, F

You see, sequenced packets drop older packets. RELIABLE and RELIABLE_ORDERED are ok for the most cases. The fifth parameter (here 0) tells which ordering stream to use. You can think of the stream as a relative ordering stream, where all packets of the same ordering type are ordered to each other. RakNet supports 32 ordering streams from 0 to 31. You can use any stream you want. Ordered streams are not important for now, so we skip them here. The sixth parameter is for the playerId. With UNASSIGNED_PLAYER_ID, the packet is sent to all connected peers. The last parameter (here it is "true") is whether to broadcast the data or not. It works together with the playerId parameter. If broadcasting is true, the player id parameter means: Send the data to all connected peers, except the one specified as sixth parameter. In this example, we send the data to everybody, who is connected. Even to the original sender! In a game you wouldn't want to do this: Only send the data to all peers, which do not have information. The sender has this information and does not need it again, so you set the player id parameter to the original sender and send. If broadcast is false, the player id tells, whom it sent the data to. If the Send()-functions return false, the data could not be sent. If the data is sent, you get "true" as return value. But again, this does not mean that your data is really sent! It is only put in a message queue waiting for transmission. So before we can disconnect from the remote peer, we better wait a second. This makes sure, the data is sent. Now we can disconnect from the remote peer:

```
Sleep(1000);
sender->Disconnect(1000);
```

The argument for Disconnect() sets a waiting time, until all waiting packets are sent. If a packet is not sent in that period, it is dropped. Do not forget to destroy the peer interface:

```
RakNetworkFactory::DestroyRakPeerInterface(sender);
```

This was a long text, again. In short, you need only four function calls to process your data: Initialize the interface, connect, send/receive, disconnect. That's it!

2.4.3.2 Client/Server

The description and overview of this sample program will be much shorter, because the steps are the same: Get an interface, connect, send/receive, disconnect. The main difference is the network interface names: RakClientInterface and RakServerInterface. Both are created with RakNetworkFactory. The sample program uses the same mechanism for determining the operation mode and IP address. The ports for server and client are now named "ClientPort" and "ServerPort".

The server part of the program uses exactly the same message processing loop as the listener part above. But before you can enter this loop, you need to initialize the server interface:

```
RakServerInterface *server = RakNetworkFactory::GetRakServerInterface();  
server->Start(100, 0, 0, iServerPort);
```

The first parameter sets the maximum number of connected clients, maximum is 65535 per server object. The second parameter is always unused. The third parameter sets the thread's sleep time, see description above. The last parameter is the port, where the server sends and receives data.

The client is the same as above, too. In the Connect()-method, the fourth parameter is the same unused parameter as in RakServerInterface::Start() the second. The last parameter is the sleep time, as always.

That was all to setup a client/server environment. Easy, isn't it?

2.5 Second Example: Peer-to-Peer chat

The second sample program is a client/server chat system. We begin with a similar program as above. But this time, we will add more and more advanced features like compression, encrypted connections, ban list and kicking, Remote Procedure Calls.

But before we start coding, we need to think about the design of our programs: At first, the clients will send and receive chat messages from other clients. So you need a new message type for this. Chatters can leave and enter a chat, so two more types. And another one for sending the list of already connected chatters. At the beginning you need 4 message types in total:

- `ID_CHAT_MESSAGE`: When a client receives it, it gets the chat message from the packet and displays it in any form. The server will get the message from the client, add the clients name/id to it and broadcasts it to all connected clients, except the original sender.
- `ID_CHAT_NEW_CHATTER`: The server will inform all connected clients, that a new person has connected. With this packet comes the name for the new chatter.
- `ID_CHAT_REMOVE_CHATTER`: When a client receives this packet type, it removes the name from the local list.
- `ID_CHAT_LIST`: When a client received `ID_CHAT_NEW_CHATTER`, it has to request the list of all connected chatters. With this packet are the names/ids sent.

Now it is time to code it ...

2.5.1 Server

The server is a command line program. It receives commands like “shutdown” from typing them into console. Entering keys is detected with kbhit() and then the whole input is saved into a string. There are some predefined commands like “quit”, which will shutdown the server and exit the program.

The first step is to include the necessary headers and create an instance of RakServerInterface and let it listen on a specific port:

```
#include <raknet/RakNetworkFactory.h>
#include <raknet/RakServerInterface.h>
#include <raknet/PacketEnumerations.h>
#include <raknet/BitStream.h>
RakServerInterface *server = RakNetworkFactory::GetRakServerInterface();
server->Start(100, 0, 0, serverPort);
```

The variable serverPort is defined as a constant in the program; you can use any number you want as long as it is above 60000. Now you can enter the message loop:

```
while(!done)
{
    Packet *p = server->Receive();
    while(p)
    {
    }
}
```

To make debugging easier, it is useful to add some output. In this case, we print all incoming packets together with the IP address and port of the sender. You get both with RakServerInterface::GetPlayerFromID():

```
char ip[22];
unsigned short port=60002;
server->GetPlayerIPFromID(p->playerId, ip, &port);
```

Now it is easy to identify a certain client. If a packet ID is unknown, the program just prints, who sent it. The same goes for ID_NEW_INCOMING_CONNECTION, ID_RECEIVED_STATIC_DATA, ID_MODIFIED_PACKET. ID_CHAT_MESSAGE is the first new chat message you need to handle. Calculate the message length (in bytes) first, this is bitlength of the packet * 8 (of a right shift of three):

```
size_t length = p->bitSize >> 3;
```

Before reading the chat message from the packet, you need to write it to a bitstream. The chat message's length is length-1 byte (for the identifier). In every packet is a PlayerID structure saved. The id in this structure is used to identify the client. With this id, the server builds a new string. This string is broadcasted to all clients and these will display it:

```
// Write received data to bitstream
RakNet::BitStream bitStream;
bitStream.Write((char*)p->data, length);

// Ignore identifier
bitStream.IgnoreBits(8);

// Read chat message from bitStream
char *msg = new char[length-1];
bitStream.Read(msg, length-1);

// Reuse bitstream
bitStream.Reset();
bitStream.Write((unsigned char)ID_CHAT_MESSAGE);
// build string
string s = string("<") + convertNumberToString(p->playerIndex) + string(">") +
string(msg);
// dont forget to count the newline character
bitStream.Write((char*)s.c_str(), s.length()+1);
server->Send(&bitStream, HIGH_PRIORITY, RELIABLE, 0, p->playerId, true);
```

Of course the client has to connect to the server before sending chat messages. After connecting, the server informs all other clients, that somebody new has joined. The packet also contains the client's id. A client's id is copied from RakNet: every new connected client gets an internal unique ID. So we just copy it and use it. You could use the index into the vector as ID, but it is not guaranteed that both are the same. This vector is made up from this structure:

```
typedef struct sChatter
{
    unsigned int binaryAddress;
    size_t index;
    sChatter(void):binaryAddress(0),index(0) {}
} Chatter;
```

The binary address is a RakNet internal and used to identify a client. When somebody connects, we copy his connection details and save them:

```
vector<Chatter> vChatters;
Chatter c;
c.binaryAddress = p->playerId.binaryAddress;
c.index = p->playerIndex;
vChatters.push_back(c);
```

Finally, all clients need to know, that there is somebody:

```
RakNet::BitStream bitStream;
bitStream.Write((unsigned char)ID_CHAT_NEW_CHATTER);
bitStream.Write((int)vChatters.size()-1);
server->Send(&bitStream, HIGH_PRIORITY, RELIABLE, 0, p->playerId, true);
```

After somebody has connected, he can request a list of chatters with ID_CHAT_LIST. Instead of sending real nicknames, we just send the ID of all connected clients as “name”. The server prepares a packet with that ID and writes all IDs as integer into the bitstream:

```
RakNet::BitStream bitStream;
bitStream.Write((unsigned char)ID_CHAT_LIST);
for(int c=0; c < vChatters.size(); c++)
    if(p->playerIndex != vChatters[c].index)
        bitStream.Write(c);
server->Send(&bitStream, HIGH_PRIORITY, RELIABLE, 0, p->playerId, false);
```

Pay attention that the server does not broadcast this packet! To achieve this, you specify “false” as last parameter of Send(). When a client can connect, he needs the possibility to disconnect, too. On disconnect, the client sends a packet with ID_CHAT_REMOVE_CHATTER. This will cause the server to disconnect the client, remove him from the local chatter list (this is just an iteration over the vector and not shown here) and inform all other clients. Disconnecting or “kicking” a client is done with RakServerInterface::Kick(PlayerID id):

```
server->Kick(p->playerId);
```

Do not forget to inform all clients, that somebody has left:

```
RakNet::BitStream bitStream;
bitStream.Write((unsigned char)ID_CHAT_REMOVE_CHATTER);
```

```
bitStream.Write(index);
server->Send(&bitStream, HIGH_PRIORITY, RELIABLE, 0, id, true);
```

The message is broadcasted to all connected clients.

Before you read the next part, please open the “chatserver” example and read through the code. If you have any questions or if something is unclear, please ask in the forum!

2.5.2 Clients

The client has a graphical GUI, made with WinAPI:

<Place screenshot here, but my VMWare Windows makes no screenshots ☹ >

The window is divided into several parts: In the center left you find the main part, where the chat and other important messages appear. At the right side the user list is displayed. At the bottom there is an input field for typing messages and aside a button to send them. The client accepts commands just like the server. In front of the commands there is a slash, like in Quake. The client supports some default commands: connect, disconnect and quit. Only “connect” accepts one argument: The server’s ip address. When you hit the “send”-button, the content of the input field is read in. If it begins with a slash, the internal list of commands is checked. If the command exists, it will be executed. Getting the arguments is made with STL strings (just a search for a whitespace). When the program is started, an instance of RakClientInterface and the GUI is created:

```
client = RakNetworkFactory::GetRakClientInterface();
```

hEdit1, hButton1, hList1 (message list) and hList2 (chatter list) are the handles for the GUI elements. When the button is clicked, the real action takes place. The text from the edit field is fetched and if the first character is a slash, it is interpreted as a command. Before a client can chat with others, he must connect to a server first. The preliminary command is separated into the real command and the arguments. The argument for

“connect” is used to connect to the server. If no argument is given, the client connects to localhost (127.0.0.1):

```
client->Connect((char*)arg.c_str(), serverPort, 0, 0, 0);
```

So, there is no port given. If you specify '0' as port, RakNet chooses a free port automatically. So you do not need to bother with it. Alternatively, you can specify a fixed port, of course. If the Connect()-function call was successful, you need to wait for the server to send a “connection established”:

```
while(!client->IsConnected())
{
    Packet *p = client->Receive();
    if(p)
        client->DeallocatePacket(p);
}
```

For better communication the client needs the list of connected users. And all other clients need to be informed about the new client:

```
// Send a message, that we are there
RakNet::BitStream bitStream;
bitStream.Write((unsigned char)ID_CHAT_NEW_CHATTER);
// Write some data to the string. RakNet check internal
// for packet size and raises an assertion if no data is
// send
bitStream.Write("something", strlen("something"));
client->Send(&bitStream, HIGH_PRIORITY, RELIABLE, 0);

// request chatter list
bitStream.Reset();
bitStream.Write((unsigned char)ID_CHAT_LIST);
bitStream.Write("something", strlen("something"));
client->Send(&bitStream, HIGH_PRIORITY, RELIABLE, 0);
```

RakNet does not allow sending empty packets with only just an ID. So you need to attach some data to the request packages. Sending this string is a waste of bandwidth, better is just sending a Boolean (1 bit). Before the client exits the program, he has to disconnect. To make sure, he disconnects, cleanup()-function is called:

```
void cleanUp(void)
{
    if(client == NULL)
        return;
```

```

// make sure, that there are now packets waiting
Packet *p = client->Receive();
while(p)
{
    client->DeallocatePacket(p);
    p = client->Receive();
}
if(client->IsConnected())
    client->Disconnect(0);
RakNetworkFactory::DestroyRakClientInterface(client);
client = NULL;
}

```

When the user wants to disconnect, it just has to send a packet with ID_CHAT_REMOVE_CHATTER:

```

RakNet::BitStream bitStream;
bitStream.Write((unsigned char)ID_CHAT_REMOVE_CHATTER);
bitStream.Write(true);

client->Send(&bitStream, HIGH_PRIORITY, RELIABLE, 0);

```

Now only one important feature is still missing: Chatting. When the program fetches the text from the edit field, it checks the string, if it is a command. If not, it is automatically a chat message. With the string you get the length of the message. All you need to do is to prepare a packet with ID_CHAT_MESSAGE and write the chat message to it. The server will handle the rest:

```

RakNet::BitStream bitStream;
bitStream.Write((unsigned char)ID_CHAT_MESSAGE);
bitStream.Write(buffer, length);
client->Send(&bitStream, HIGH_PRIORITY, RELIABLE, 0);

```

That's it ☺

Now to the message receiving part: When a packet with ID_CHAT_MESSAGE is received, the program fetches the string from the packet and displays it. The server has built the complete message with the original sender for us:

```

const size_t length = p->bitSize >> 3;
RakNet::BitStream bitStream;
bitStream.Write((char*)p->data, length);

```

```

// Ignore identifier
bitStream.IgnoreBits(8);

char *msg = new char[length-1];
memset(msg, '\0', sizeof(char) *(length-1));
bitStream.Read(msg, length-1);

// construct message for display
SendMessage(hList1, LB_ADDSTRING, 0, (LPARAM)msg);

```

The length of the chat message is the length of the whole packet minus the one byte for the identifier. This identifier is ignored, the message copied and displayed in the GUI list. The process, when somebody new has entered, is similar: This time we read an integer as “name”:

```

RakNet::BitStream bitStream;
bitStream.Write((char*)p->data, length);

// Ignore identifier
bitStream.IgnoreBits(8);

// read id
int id=0;
bitStream.Read(id);

```

The user is added to the list (hList2) and a message is displayed in the main field:

```

// inform client, that somebody is here
string s = convertNumberToString(id) + string(" has entered!");

// Display new user
SendMessage(hList1, LB_ADDSTRING, 0, (LPARAM)s.c_str());
SendMessage(hList2, LB_ADDSTRING, 0,
(LPARAM)convertNumberToString(id).c_str());

```

When a new client joins the server, he needs to know who is already connected. The server provides this information with an ID_CHAT_LIST packet (remember: when the client connects, he sends a packet with the same ID to the server to request this list). The packet consists of the packet identifier and all IDs. An ID is an integer (4 bytes), the number of connected chatters is packet’s byte length – 1 byte, divided by 4 (because of 4 bytes per ID). Instead of the division you better make a shift, that is a

lot faster. Now you can iterate through the packet's data and add the user name to the local list:

```
// the ids are saved as integer. Every int has 4 bytes
bitStream.IgnoreBits(8); // ignore packet identifier
const size_t n = (length-1)>>2;
string d = convertNumberToString(length) + string(" ") + convertNumberToString(n);

for(size_t c=0; c < n; c++)
{
    int id=0;
    bitStream.Read(id);
    SendMessage(hList2, LB_ADDSTRING, 0,
        (LPARAM)convertNumberToString(id).c_str());
}
```

The last possible packet is the exit of a chatter. This packet has the ID ID_CHAT_REMOVE_CHATTER. With this packet comes the name of the user, which has left. The only thing you need to do is removing the chatter from the list (hList2). Everything else is server's business. The ID is read from the packet and then a loop starts through the list of names. If the name is found, it will be removed and the user will get a message about this.

That is all you need to do for a small chat system. Now I want to show you, how you can use some advanced features of RakNet. The recently developed programs serve as a basis; so make sure you have understood them! If you haven't yet, please go to the forums and let me help you.

2.5.3 Extending both programs

RakNet has some more features, which I want to show you now. All of them are useful for game programming.

2.5.3.1 Remote Procedure Calls

Remote procedure calls allow you to call a function on another system, the remote computer. This is particular useful, if you want to rapidly develop a network based program, e.g. a chat. Register the functions, which are

allowed to be called by another computer and you are already done. No more packet handling.

The sample programs implement a new server command: shutdown. When this is invoked, all clients display a message box that the server is going to shutdown. This message box is created in a RPC function. The server will call this function on all clients. So let us implement it.

An RPC-function has always the same prototype:

```
void <functionname>(RPCParameters *rpcParameters)
```

The only argument is a structure, which holds several variables like the sender, to sent data and the size of the data. The client function has the name „shutdown“ and displays a message box with the send string („input“-argument):

```
void shutdown(RPCParameters *rpcParameters)
{
    // Display server's shutdown message
    MessageBox(NULL, rpcParameters->input, "Shutdown!", 0);
}
```

Now you need to register this function by RakNet, this is done right after the network interface is created:

```
RakClientInterface *client = RakNetwokFactory::GetRakClientInterface();
REGISTER_STATIC_RPC(client, shutdown);
```

The second line calls a macro with the network interface as first parameter and the function's name as second. Please note, that the name is not given as string. The macro will convert it automatically. The server gets a new command, called "shutdown". This will call the function "shutdown" on all clients and then exit the server program.

```
if(command == "shutdown")
{
    string shutdownMessage = "Server is going down now...";
    server->RPC("shutdown", (char*)shutdownMessage.c_str(),
    (shutdownMessage.length()+1)*8, HIGH_PRIORITY, RELIABLE, 0,
    UNASSIGNED_PLAYER_ID, true, false, UNASSIGNED_OBJECT_ID);

    // wait 3 seconds, to make sure the disconnects succeed
```

```
    sleep(3);
    done = true;
}
```

You can change the shutdown message to whatever you want. This string is displayed to all the clients in the message box. RakServerInterface::RPC() takes the function name (this is very important!) as first parameter. It follows the data, which is taken as second argument of every remote procedure call (see function prototype above). In our case, it is the exit message. The third argument is the size of this data in bits (!). As always, when you send a string, do not forget to send the correct size with the trailing zero character. The next three arguments are the standard arguments for every Send()-function. The function should be called on all clients, so we broadcast the function call to all clients. This is done with UNASSIGNED_PLAYER_ID as player id argument and activating broadcasting. When you set the last parameter to “true”, the first four bytes of the given are interpreted as a timestamp (see further below for timestamps).

2.5.3.2 Ban list and kicking

Sometimes players are evil or cheating and you do not want them playing on your server (or in our case: not chatting anymore). Or maybe as server administrator you want to bother somebody; then you can kick him from the server. This disconnects the client and he has to reconnect, if you allow it to him. If not, he is “banned” from your server. This can be a temporary or lifelong ban. RakNet implements a server side ban list, so you just need to add or remove clients. This list works with IP addresses, so it is not very useful for clients with a dial-up connection. When a client connects, the instance of RakServerInterface will look into the ban list for the IP address. If it is not in the list, the client will be allowed to connect. If it is part of the ban list, the server replies with an ID_CONNECTION_BANNED packet and disconnects the client. So the client has to check for this packet when connecting.

This is just a small change in the client code: We want to inform the user, that he is banned. Remember the place, where the client waits for

connection establishment. This is the place, where some new code is placed:

```
// Make sure, we have a connection
while(!client->IsConnected() && !banned)
{
    Packet *p = client->Receive();
    if(p)
    {
        // new
        if(p->data[0] == ID_CONNECTION_BANNED)
        {
            MessageBox(hWnd, "Go away, you are banned!!", "Banned!", 0);
            banned = true;
        }
        client->DeallocatePacket(p);
    }
}
if(!banned)
    SendMessage(hList1, LB_ADDSTRING, 0, (LPARAM)("Connection established"));
```

The new thing is the check for the packet with ID_CONNECTION_BANNED. “banned” is Boolean with a default value of false. When the client gets the “banned”-message, the Boolean is set to true and a message box informs the user about this. If he is not banned, the usual “connection established” message appears. If the user now types a chat message, the client network interface will ignore it (because he is not connected).

To add a banned IP, you have to call RakServerInterface::AddToBanList(). The only argument is the ip address as a string. To remove somebody from the ban list, you call RakServerInterface::RemoveFromBanList() with the IP as argument. RakServerInterface::ClearBanList() will remove all entries in the ban list. The server gets two new commands: “kick” and “ban”. Both commands take the player index as argument. With this index being the IP address fetched and the client gets kicked/banned. At first the short “kick” command:

```
if(cmd == "kick")
{
    // first and only argument is index to kick
    PlayerID id = server->GetPlayerIDFromIndex(atoi(args.c_str()));
    server->Kick(id);
}
```

```

char ip[22];
unsigned short port=0;
server->GetPlayerIPFromID(id, ip, &port);
cout << "Kicked client: " << ip << "." << port << endl;
}

```

To kick a client, you need his PlayerID structure. You get this with RakServerInterface::GetPlayerIDFromIndex(). Just pass this id to Kick() and you are done. The server administrator is informed, which IP was kicked. The “ban” command is only one function call longer:

```

if(cmd == "ban")
{
    // first and only argument is index to ban
    PlayerID id = server->GetPlayerIDFromIndex(atoi(args.c_str()));
    char ip[22];
    unsigned short port=0;
    server->GetPlayerIPFromID(id, ip, &port);
    server->AddToBanList(ip);
    server->Kick(id);
    cout << "Banned and kicked client: " << ip << "." << port << endl;
}

```

The client is added to the server’s ban list and then kicked. The kicking mechanism is the same as above. New is the call to RakServerInterface::AddToBanList(). This time the administrator will be informed too.

2.5.3.3 Passwords

Often you do not have personal access to your server and you need to administrate it over a network. An administrator has to reveal oneself with a password. The easiest way to do this is with a new client-side command. This command takes the potential password as argument. To identify, that a client wants to login a new packet type is created: ID_CHAT_LOGIN. The packet contains the password as a string. When the login command is invoked, the argument is read in and the packet is prepared. This is similar to the connect-command:

```

RakNet::BitStream bitStream;
bitStream.Write((unsigned char)ID_CHAT_LOGIN);
bitStream.Write((char*)arg.c_str(), arg.length()+1);

```

```
client->Send(&bitStream, HIGH_PRIORITY, RELIABLE, 0);
```

The server receives this packet, reads the password and compares it with the administrator password. If both are the same, the server replies with an ID_CHAT_LOGIN packet and an integer with a value of 1. If the passwords are not equal, the client gets a 0. The user of the client program gets the feedback string in the main window.

The server receives the packet with the password from the client and reads it into a std::string. This string is compared with the server-side saved administrator password (the loginPassword-variable). After this comparison, a packet with ID_CHAT_LOGIN is prepared. If the comparison was successful, a one as integer will be written to the packet. In any other case a zero will be written:

```
char *password = new char[length-1];
bitStream.Read(password, length-1);

bitStream.Reset();
bitStream.Write((unsigned char)ID_CHAT_LOGIN);
int matches=0;
if(loginPassword.compare(password) == 0)
    matches = 1;
bitStream.Write(matches);

// Send reply
server->Send(&bitStream, HIGH_PRIORITY, RELIABLE, 0, p->playerId, false);
```

Now the user could invoke other commands like kicking users from the server or shutting it down.

2.5.3.4 Encrypted connections

In the previous example the password has been sent as a usual string. This way everybody in your network can receive this packet and easily read the password. With this password, one could hijack your server. Not a good situation at all. One way to improve this is the usage of encryption. RakNet has a built-in encryption mechanism. Encryption in RakNet is based on the RSA algorithm. The security of the RSA cryptosystem is based on two mathematical problems: the problem of factoring very large

numbers and the RSA problem. I do not want to go too deep into the mathematical details; they are boring ;-)

The basic idea behind RSA is that the factoring of very large numbers into two numbers is a very time consuming issue. But creating such a large number is easy; just multiply two prime numbers. Imagine Alex and Bob want to secure their communication. Alex generates a private and a public key using RSA (both keys depend on each other). He gives the public key to Bob, but keeps the private key private. Bob wants to tell Alex something really important, so he takes the public key he got and encrypts his message with it. Now he can send the message to Alex and nobody else can read it. Alex uses his private key to decrypt the received message and read the plain text. To reply Bob, Alex needs a public key from Bob. The key's length is determined in bits; the bit size is often stated. Common values are 2048, shorter values are not recommended. If the bitsize is 256 or shorter the keys can be deciphered within several hours. But the longer the key, the more CPU power it needs. RakNet's built-in encryption uses bitsizes of 256 and 512. This is supposed to be enough for games. Longer keys would use too much CPU time. Now let us code it. With RakNet you can only encrypt a whole connection (and all its packets) or not. If you need to encrypt single packets, you need to do it on your own. To enable encryption you need only one additional function call on the server and one on the client:

```
client->InitializeSecurity(NULL, NULL);
```

and

```
server->InitializeSecurity(NULL, NULL);
```

The server's `InitializeSecurity()`-function takes the private key as arguments. For easier handling, this key is divided into two variables. The client's function takes the public key as arguments; this key is also divided into two variables. When you pass `NULL` to all these arguments, RakNet will generate the keys internally. But key generation is a time consuming task, so maybe you want to precompute this. You can utilize RakNet's `RSACrypt-template` to do it. To instantiate this template, you have to specify the bit size of the desired keys. Now you only just need to call

generateKeys(). There are two get()-methods to save both keys into your own variables: getPublicKey() and getPrivateKey(). Now, all you need to do is passing the keys to the server and client. The server takes the private key variables and the client the public key variables. To transmit the public key on the server to the client the key is saved into a file (when passing NULL to the initialization functions, the keys are automatically transmitted to the clients on connection) and the client reads this file. The new server code:

```

u32 p; // public key
RSA_BIT_SIZE q; // public key
BIGHALFSIZE(RSA_BIT_SIZE, e); // private key
BIGHALFSIZE(RSA_BIT_SIZE, n); // private key

big::RSACrypt<RSA_BIT_SIZE> rsacrypt;

rsacrypt.generateKeys();
rsacrypt.getPublicKey(p,q);
rsacrypt.getPrivateKey(e,n);

server->InitializeSecurity((char*)e, (char*)n);

```

The saving part is skipped here, the four variables are just written into a file. The client will read this file and pass the variables to his initialization function:

```

u32 p;
RSA_BIT_SIZE q;

FILE *f = fopen(publicKeyFile.c_str(), "rb");
fread((char*)&p, sizeof(p), 1, f);
fread((char*)&q, sizeof(q), 1, f);
fclose(f);

client->InitializeSecurity((char*)&p, (char*)q);

```

Keep attention to the pointers at InitializeSecurity(). When connecting and the client's public key does not match the server's key; a message box will be displayed and the connection attempt will be stopped:

```

bool keyCorrect=true;
while(!client->IsConnected() && keyCorrect)
{
    Packet *p = client->Receive();
}

```

```

    if(p)
    {
        if(p->data[0] == ID_RSA_PUBLIC_KEY_MISMATCH)
        {
            MessageBox(hWnd, "You dont have a valid public key. Please call
the administrator!", "", 0);
            keyCorrect = false;
        }
        client->DeallocatePacket(p);
    }
}
if(!keyCorrect)
    client->Disconnect(0);

```

You do not need anything else to have a secure connection. This connection also helps against cheaters, so keep that in mind.

2.5.3.5 Timestamps

RakNet can use timestamps to adjust the timers on all connected systems. Imagine you shoot on you local system at the time 8000. The packet receives on time 2000 on the server and receives at your opponent's computer at (his local) time 16000. Timestamping adjusts all these times to one value, which means your shoot happens at time 8000 on your computer and needs 5 ticks to travel to the server. It receives the server at 2005 and the second client at 16014 (the client's local time). For games this synchronisation is very important for inter-/extrapolation of different values like movement (you will learn about this in the game programming chapter). The timestamp is specified at the beginning of your packet, so you need to rearrange the reading and writing part. The new packet identifier is ID_TIMESTAMP. After this identifier comes the real timestamp, this is a 32 bit unsigned long (4 bytes). You get this value with `RakNet::GetTime()`.

If you use timestamping, you have to use `RakNet::GetTime()` on all systems. You can NOT use your own timers!

After the timestamp you can put in your packet identifier and data, just like in a normal packet.

The edited server takes the time from the packet and appends it to the generated chat message. When a client sends a chat message, the timestamp is written to the packet. This is all the client has to do to use timestamping:

```
// Send the chat message to the server
RakNet::BitStream bitStream;
// new
bitStream.Write((unsigned char)ID_TIMESTAMP);
bitStream.Write((unsigned long)RakNet::GetTime());
// Now the normal data
bitStream.Write((unsigned char)ID_CHAT_MESSAGE);
bitStream.Write(buffer, length);
client->Send(&bitStream, HIGH_PRIORITY, RELIABLE, 0);
```

The packet handling in the server code needs a small change, because the chat messages now arrive with ID_TIMESTAMP:

```
case ID_TIMESTAMP:
{
    size_t length = p->bitSize >> 3;
    RakNet::BitStream bitStream;
    bitStream.Write((char*)p->data, length);

    // Ignore ID_TIMESTAMP
    bitStream.IgnoreBits(8);

    // Ignore timestamp
    bitStream.IgnoreBits(sizeof(unsigned long)*8);

    // Ignore identifier
    bitStream.IgnoreBits(8);

    size_t messageLength = length-1-sizeof(unsigned long)-1;

    // Dont forget to remove the timestamp before reading the chat message
    char *msg = new char[messageLength];
    memset(msg, '\0', messageLength*sizeof(char));
    bitStream.Read(msg, messageLength);

    bitStream.Reset();
    bitStream.Write((unsigned char)ID_CHAT_MESSAGE);
    // new: Add the server time to the message
    string s = string("<") + convertNumberToString(p->playerIndex) + string(":") +
                convertNumberToString((unsigned
long)RakNet::GetTime())+ string(">") + string(msg);
    bitStream.Write((char*)s.c_str(), s.length()+1);
```

```

server->Send(&bitStream, HIGH_PRIORITY, RELIABLE, 0, p->playerId, true);

delete []msg;
msg = NULL;
}
break;

```

The first four lines should be common now: The packet's data is written to a bitstream and the packet identifier is ignored. After this identifier follows the timestamp and the data's identifier. They are also both ignored. The chat message's length is the whole packet length minus the ID_TIMESTAMP identifier, minus the timestamp and minus the data identifier (ID_CHAT_MESSAGE in this case). The chat message is read from the bitstream and the complete message for the clients is prepared. The server's local time is saved into the chat message string and then it's send to the clients. These will display the message. The client has a similar change in his message handling code. The only difference is that the client will display the chat message. There's one important thing you have to do before you can use this feature: Timestamping requires sending of occasional packets to publish a system's time. RakNet can do this automatically; you just have to enable it. Before starting a server or connecting a client to a server, call RakClientInterface::StartOccasionalPing() or RakServerInterface::StartOccasionalPing():

```
server->StartOccasionalPing();
```

```
client->StartOccasionalPing();
```

2.5.3.6 Statistics

I will keep the last part of this rather short. RakNet generates some statistics about the connection quality. You get this information with the GetStatistics()-method. It takes a PlayerID structure as argument and returns a structure containing information about the connection quality between the requesting system and the system from the PlayerID. For our chat program returns the client's method the quality to the server. This information is displayed in a separate message box. The most important part is the "packet loss" line: This line says, how many of your send

packets are send several times to reach their destination. To display the message box, the client program got a new command called “stats”. A utility function takes the above-mentioned statistics structure and creates a string from it. The last argument of this function is the verbose level. You can choose 0, 1 or 2. The higher the value, the more information you will get. 1 is a good compromise:

```
// make sure we are connect, else we have no statistics to show
if(client->IsConnected())
{
    char stats[2048];
    memset(stats, '\0', sizeof(char) * 2048);
    StatisticsToString(client->GetStatistics(), stats, 1);
    MessageBox(hWnd, stats, "network statistics", 0);
}
```

2.6 Third Example: Simple Game

Now it is time to show you how to develop your first game with network support. I chose the simplest game you can imagine (so the code has only a bit of game code): Pong. Additionally, the program has no advanced features; every network game has usually like dead reckoning. And it does not handle packages. Instead the program uses the RPC feature of RakNet to control the game. All in all, the game code is just some lines long and the network code is only some more lines longer. The game is based on the client/server architecture. The client sends, if the player presses or releases a key. Each player has only two keys to control his paddle: Up and down. The server takes this input and computes the paddle movement for each client. But before I show you some code, take out a sheet of paper and pencil. You need to make some maths before. But do not be afraid, it is just additions and subtractions ☺

2.6.1 The gamefield

Before you can start coding, you need to think about this game. When a paddle collides with the walls or what happens, when the ball collides with a wall/paddle. So let us get started. At first, draw a rectangle. This will be

the game field. In order that the players can see the borders, make four rectangles from the big one. This should look like this (I made mine with GIMP):



Figure 4: Basic game field made from 4 rectangles

Now add the two paddles and the ball:

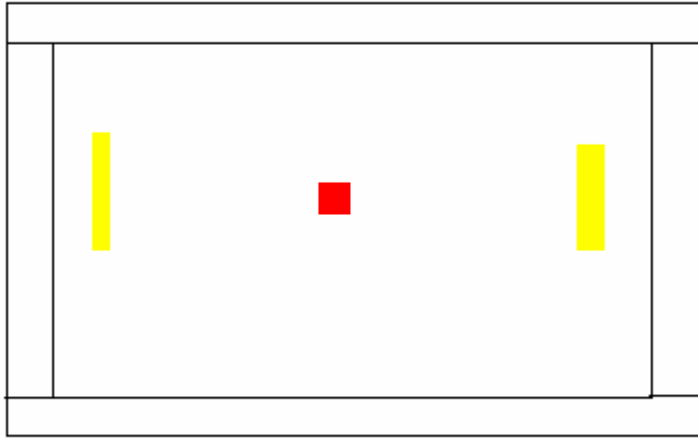


Figure 5: Game field with paddles and ball

The sizes are missing, so add them:

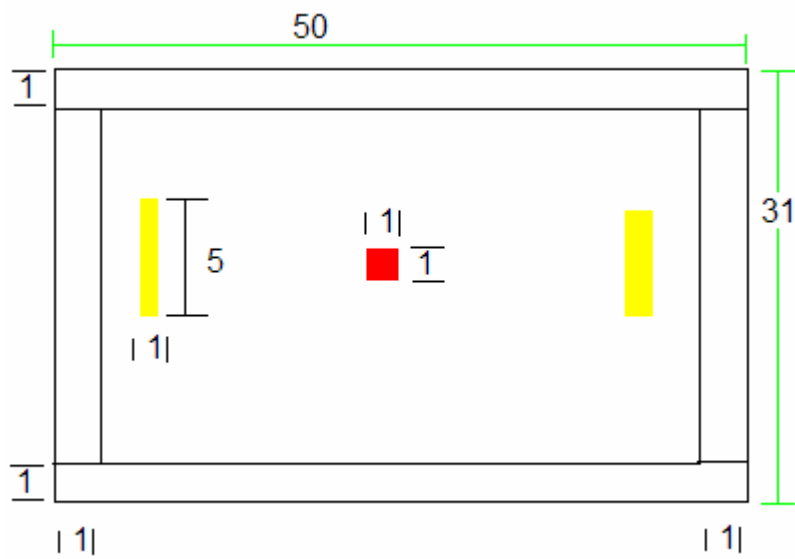


Figure 6: Complete game field with sizes

When you look at the images, you know why I am a programmer and not an artist ;-)

Each paddle is five units high and one unit wide. The ball is a quad with a side length of one. Each of the borders has a thickness of one. The width of the complete gamefield is 50 units and it is 31 units high.

The ball and the paddles are controlled via the center. That means, if the ball's position is (5;0) it is *centered* around this point. The gamefield itself is centered around the point (0;0). I have added these center points to the image:

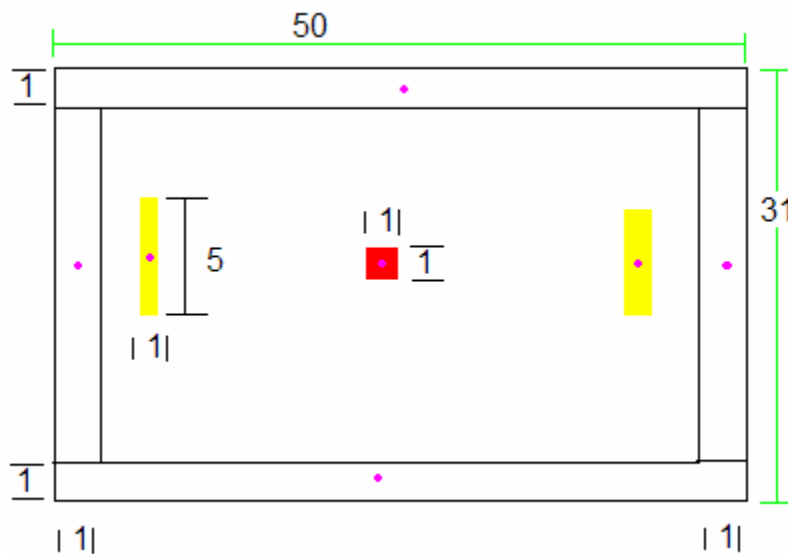


Figure 7: Pong gamefield with center points added

The “real” gamefield width is 50 units – 1 unit (left border) – 1 unit (right border) = 48 units. Analogue for the height: 31 units – 1 unit (top border) – 1 unit (bottom border) = 29 units. Each paddle has 2.5 units (5 units / 2) size in both (top and bottom) directions. With this information, you can start to think, when a paddle is colliding with a border. I have zoomed into the figure 7:

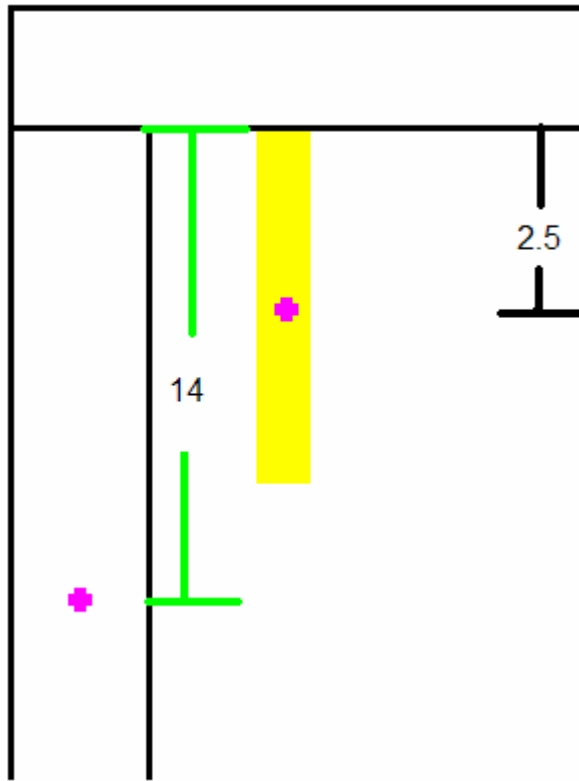


Figure 8: Zoomed gamefield for collision calculation

The gamefield is centered around (0;0) so it goes 14.5 units above and down. The figure shows the part, which goes above. Simple math ($14.5 - 2.5$) gives you: If the paddle is at height 12 or higher, it collides with the top border. If the height is -12 or smaller, it collides with the bottom border. A similar calculation is used for the ball-wall collision detection, but this time the ball has a size of 0.5 units (from the center). Thus the ball collides with the bottom/top border, if it is at height $14.5 \text{ units} - 0.5 \text{ units} = 14 \text{ units}$. The players try to move the ball behind the enemy's paddle. Both paddles are placed at width +/- 20 units. So you can choose any width you want for the "target zone", as long as it beyond 20 units. I have taken +/- 21 units. Now to the hardest part of this game code: ball-paddle collision. Another image first:

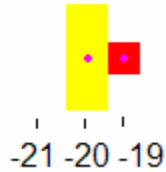


Figure 9: Ball and paddle, zoomed

This image is zoomed a lot. It shows a yellow paddle (the left one) and the red ball. The units are inscribed below. The paddle's center is at unit -20 and has a width of 0.5 units (from the center). The ball's center is at -19 units and has a width of 0.5 units, too. The ball and paddle collide if the ball is at position $-20 \text{ units} + 0.5 \text{ units (paddle width)} + 0.5 \text{ units (ball width)} = -19 \text{ units}$. But sometimes, the ball moves very fast and the ball's center is at a smaller position than -19 units. So it is better to define: The ball collides with the paddle is between -21 and -19 units (remember, at -21 units begins the target zone). But wait! You have to check the height of the ball, too! Ok, this is similar to the calculation before. The ball can only collide with the paddle, if the ball's height is within the range of the paddle.

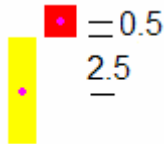


Figure 10: Ball and paddle, zoomed

This time, you see the paddle and the ball with its sizes. The ball can collide with the paddle, if it has a height of: $\text{paddle height} + 2.5 \text{ units (paddle size in that direction)} + 0.5 \text{ units (ball size)}$. And for the side below: $\text{paddle height} - 2.5 \text{ units} - 0.5 \text{ units}$. You see, the collision result depends on the position of the paddle (as expected). But what happens, when the ball or a paddle collides? First the paddle-border collision: Set paddle's height is reset to $\pm 12 \text{ units}$, so it can enter the wall. Before you can define, what happens with the ball on collision, you need to know how the ball moves. This movement is made with a formula, taken from physics:

*New position = old position + velocity * elapsed time*

The positions and velocities are vectors. The velocity is scaled with the elapsed time since the last position update and added to the old position to get the new position. When a ball hits a wall, it is reflected. In two dimensions means this (horizontal wall, as in this game): new velocity $y = \text{old velocity } y * (-1)$:

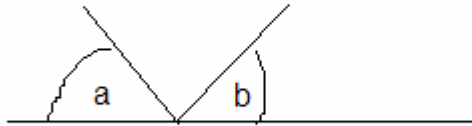


Figure 11: Reflection of a vector on a wall

Or in other words: The angle a between the wall and the incoming vector is the same angle b as between the outgoing vector and the wall. For a paddle-ball collision you modify the x component of velocity with the same formula: new velocity $x = \text{old velocity } x * (-1)$.

That's it! Now you can start coding your first game with network support. So let's do it!

2.6.2 The client

The client code is only a few lines long. The program has to inform the server, that a key was pressed or released. This is the first optimisation I use: The client sends only updates, if paddle movement has started or stopped. You do not need to send this, e.g. every half second. The change in key state is the only information you need. The second task of the client

is to display the complete. To do this, it receives regular game state (ball and paddle positions) from the server and displays this.

2.6.2.1 The client's code

The sample uses GameVersity's Sipogen engine. This engine has a built in class named *"CEngineApplication"*. When you derive your class from this, you do not need to initialize the whole engine. This saves some time and work. Before you can call your own init function, you have to ask the player, if he wants to be server (and client) or only client (which connects to the server). Depending on his answer, the correct network init function is called. If the player chooses "server", a server is started on port 64444 and the client is connected is connected to this server (using localhost or 127.0.0.1 as IP address). Acting as a server without playing the game is not possible. I do not want to show this network, because it is rather simple and similar to the network code in the first sample. The only new is the timeout in the clients connection try. When only one client is connected, the game is rather boring: You can play (start a game with space), but your enemy does not react. So another client has to connect for a playing this game. To receive game state updates, the client registers the function "positionUpdate" as a RPC. This is called from the server and gives the client new information about the game:

```
// receive position update for all two paddles and ball from server
void positionUpdate(char *input, int numberOfBitsOfData, PlayerID sender)
{
    RakNet::BitStream bitStream;
    bitStream.Write(input, numberOfBitsOfData/8);

    bitStream.Read(paddle1Height);
    bitStream.Read(paddle2Height);
    bitStream.Read(ballPosition.x);
    bitStream.Read(ballPosition.y);
    bitStream.Read(ballPosition.z);
}
```

The head of the function is the standard RPC function head. A bitstream from RakNet is used to read the data. The most important function is used, when a player presses or releases a key:

```

void paddleMove(bool start, bool up)
{
    RakNet::BitStream bitStream;
    int s=(int)start;
    int u=(int)up;
    bitStream.Write(s);
    bitStream.Write(u);
    if(!client->RPC("playerMove", &bitStream, HIGH_PRIORITY, RELIABLE, 0, 0))
        MessageBox(NULL, "Failed to send player movement!", "", 0);
}

```

The first parameter tells, if a movement has begun or is stopped (if the key was pressed or released). The second says, in which direction the paddle is moving (if it moves). A paddle can move only in two directions: up or down. If it does not move up, it is moving down. Both values are written to a bitstream and a function is called on the server, to send him this information. To get the positions for rendering, the main program uses *clientGetPosition()*. This function copies the values from the server into the arguments and returns.

2.6.3 The server

The server code is a little bit longer, but nothing complicated there. I have decided to include another small optimisation: The game's status is updated only every 33ms and the clients get only every 100ms game status updates. This reduces the used CPU time and decreases network traffic. But this method has a drawback: The paddle and ball movement is not very smooth. Both are jumping around. I will show you later, how you can make a smooth movement!

The server manages the connected clients with a vector of these structures:

```

typedef struct SPlayer
{
    float height;
    bool moving;
    bool movingUp;
    PlayerID playerID;
    SPlayer(float h, bool m, bool u)
    {}
}

```

```

} Player;
std::vector<Player> players;

```

For every client is the current paddle saved, if the paddle is moving and in which direction. The host program has to call the function *serverUpdate()* to perform necessary updates to the game. The function checks at first, if any network packets are waiting and process them. New clients are added to the player list or removed, if they have disconnected. Now comes the “magic”: The function has two static variables, which determine when the last game and network update was:

```

static unsigned long lastGameUpdate=0;
static unsigned long lastNetworkUpdate=0;

```

If the current time (remember: Use RakNet’s timer!) is bigger than the last update time + 33ms, a game update is performed and lastGameUpdate gets a new values. The same procedure for the network updates, except they are only every 100ms:

```

// update game every 33 ms
if(RakNet::GetTime() > lastGameUpdate+33)
{
    const float time = (RakNet::GetTime()-lastGameUpdate)*0.001f;
    gameUpdate(time);
    lastGameUpdate = RakNet::GetTime();
}
// but send network updates only every 100ms
if(RakNet::GetTime() > lastNetworkUpdate+100)
{
    const float time = (RakNet::GetTime()-lastNetworkUpdate)*0.001f;
    networkUpdate(time);
    lastNetworkUpdate = RakNet::GetTime();
}

```

GameUpdate() performs all the calculation, you have affiliated above. This code is just some if/else statements and position resets, so I do not want to show this here. *NetworkUpdate()* writes the paddle heights from the first two clients and the complete ball position to a bitstream and broadcast this message to all connected clients. This is rather simple, so I do not want to show it here either ;-)

2.6.4 Final words

This looks like a lot of complicated code, but it is not! Step through the code and you will see, that it is quite easy to write a simple network game. Ok, this Pong clone has not a lot of features and the movement is not smooth, but you learn to correct these flaws in the next chapters. For now, sit back and enjoy 😊

