UDP Examples

COLLABORATORS

	TITLE : UDP Examples		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		October 9, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

UDP Examples

1.1 Contents

```
                          UDP_Chat and UDP Funcs
                  ----------------------
             Written by Anton Reinauer- 08/03/1999
                Copyright Anton Reinauer 1999
```

1.2 disclaimer

Let’s get the legal stuff over with!

The author takes NO responsibility for any damages caused, or loss of property, life, or whatever, by this Program and associated Functions.
Any use of the Program and Functions is purely at the user's risk, and the author is not liable for any damages incurred through the use of these Program and Functions.

1.3 introduction

These examples are useful for anyone who wants to write a internet multi-user, fast reaction game (platformers, Quake clones etc), that by their nature must have small communication delays (lags). These routines are for communicating information between machines connected through the Internet. These programs differ from Paul Burkey's TCP examples in that they use the UDP internet protocol instead of TCP for speed (TCP is ok for strategy or turn based games as a slight lag is acceptable with them). UDP is much faster, but unlike TCP there is no guarantee that your data packet will get to it's destination (UDP is about 90% reliable); so you have to write your communication code with this in mind. These routines use Peer-to-Peer comunications rather than Server-Client (although the latter is used for Login, and will be used for system messages like disconnecting).
You don't need a internet connection to run these programs, as you can run multiple copies on the same machine, because the TCP/IP protocol works the same, wether communicating on the same machine or to the other side of the world! You can even have one UDP_Chat program acting as Server and Client, and log into itself, for quick testing purposes- it needs fixing really :)

1.4 installation

Just put this directory where you want. You need TCP-to-Blitz.lha from dev/basic on Aminet, and recompile your Deflibs with bsdsocket.library1 in your amigalibs drawer in Blitzlibs: .
You'll obviously need an TCP/IP stack like Miami or AmiTCP running (an Internet connection isn't necessary if you're just running it locally). You have to have RexxMast running to use the Arexx port. You can use AmiComSys to control the logging into a Server if you want.
You have to turn overflow errors off in the debugger options in Compiler Options, and you have to have amigalibs.res resident in the compiler options as well.

1.5 requirements

Minumum Requirements:
An A500 with some extra fast mem (to run the TCP/IP stack in), and WB 2.0+.

1.6 udp_send and udp_receive

The UDP_Send and UDP_Receive are just simple examples of setting up a UDP socket and sending and receiving data. UDP_Chat is a IRC-like chat program that both sends and receives (it is a test bed for internet game communications). All programs open UDP sockets (domain AF_INET, and type of SOCK_DRAM). The Send and Receive progs and the earlier versions of UDP_Chat are in the Old_Stuff directory, so you can start out with the simpler programs first, if you find the lastest version of UDP_Chat a bit daunting :-)

The first Send program (UDP_Send) just simply sends each packet to the host address and port defined at the top of the program (this could be different for every data packet). UDP_Receive binds it's socket with bind_() to localhost at the port defined at the top of the program (this port can't have an entry in Services though). Then it just checks if any data is waiting on the socket and gets it.

The second Send program (UDP_Send2) does things slightly differently. It opens a socket and then does a connect_() to the port on the appropriate host (defined at the top of the program). Then it just uses send_() like in TCP, and all packets will be sent to the connected socket. UDP_Receive2 does the same as UDP_Receive, and binds the socket to the port, but because there's a TCP-like connection to the sending host, it just uses recv_() to receive data.

Just run UDP_Send and UDP_Receive (or UDP_Send2 and UDP_Receive2), and type some letters into the Send window and hit return, and they will appear in the receive window. To close the programs, hit the close button in their window. Note: UDP_Send works with UDP_Receive2, and vice-versa!

1.7 udp daemon

I haven't got the programs working as a Daemon yet- when I tried, it opened three programs for every packet sent (and AmiTCP said it had got into a loop)! It appears UDP works slightly differently than TCP, I think your Daemon has to signal back to the TCP/IP stack that it's running or something. I don't think you'll really need a UDP Daemon anyhow, as you'll probably being doing peer-to-peer connections, rather than using a Server. You could always use a small TCP Daemon, which puts up a window, to ask the user if they want to play a game, which could then run your game and exit.

1.8 udp_chat

UDP_Chat is a simple IRC chat type program using UDP. I've ↩ added some
horrible hacks to Alvaro Thompson's (once) nice font sensitive code :-)
- Typical bloody games programmer ;-) . I have to write more docs for it, but you should be able to work out what it's doing. I've written it as a game communication code test program.

```
 You have to have a TCP/IP stack running before you run UDP_Chat.
It can act as a Server or a Client- if it receives a correct connection
request, it becomes the Server, and can have up to 7 other players online
(you can set the total number of players with the #MAX_NUMBER_PLAYERS
constant). If it logs into another program (that isn't already online as a
client, and has a spare player position) it will be logged in as a Client
(as long as it isn't already a Server)- note it will still send messages
directly to all the other players (when logged on), not through the Server.
You can run up to #MAX_NUMBER_PLAYERS UDP_Chat programs locally for
testing, and all can log into program acting as a Server- they all get
different port numbers.
  It disconnects if the close button is pressed (or if it's the Server and
all clients have disconnected). If it's a Client it will tell the Server
it's disconnecting, and will wait until it receives confirmation from the
Server before exiting; if it's the Server it will wait until all players
have confirmed the game is closing before exiting. Also it will exit after
a certain time-out anyway if it doesn't receive the correct
confirmation(s).
   Your host address is printed beside the Localhost button. You have to
give this address to the other person through IRC, if your program is
going to act as Server (or vice-versa), and then they type the address
you give them in the Send To: gadget and hit return. You shouldn't have
to change the port number gadget (unless you've got two of them running
locally- the second one will be bound to a port number one number higher
than the first- the port UDP_Chat is bound to is printed up in the
top-right corner). You can log into a Server automatically with AmiComSys,
and an
                Arexx script
                . Also you could get info from
StrICQ, with some extra coding.
  Then just type into the Send: gadget and hit return. Your words will
be echoed on the screen above and will be sent to any players online.
If you can't connect to a host (it will tell you under the
Send To: gadget), you won't have your words echoed onscreen, and they
won't be sent. The numbers after R: and S: (received and sent), are
packet number, and the player the packet is sent or received from.
   You have to change the constant #NO_CONNECTION at the top of the
code if you're running your programs locally: 1 if you haven't been
online, or 0 if you have been/are online. 'Localhost' seems to work
differently in each case- don't ask me why :-) If you're testing
programs online it needs to be set to 0. Also you have to set the
#DEBUG constant to 1, which allows you to have multiple copies on one
machine (otherwise, if it finds that there is already a UDP_Chat Arexx
port it will exit). It's also for outputting all debug data onscreen.
   The UDP_Funcs are in a separate Include- these are the basic send,
receive, and Initialise UDP functions (probably most of this program
could be put into that include). UDP_Chat checks wether packets
have arrived at their destination and resends them if they haven't
(ie: no reply after a certain time- determined by 'max_wait.w'). After
'number_resends.w' resends it assumes the link is dead and the player is
considered offline.
  If it is the Server, it disconnects the player and tells all the other
players that that player is offline. If it is a Client it tells the Server
and if the Server can't connect to the player, the Server then disconnects
the player as above. If the Server can connect to the player then the
Client that claimed the other player was offline, is disconnected. It is
done this way as a security thing- to stop people cheating by claiming
```

someone is offline (the claims are stored in the player_honesty() array for later checking if the other player is found to be online)! This is for the rare case that two players have a connection to a third, but only one of those players loses connection to the third one. The connection to the Server is the one that is important.

The Ping times are printed out beside each player's number, and you can tell what your player number now is- 'Me' is printed in place of your Ping time. The Server is always Player 1.

The Delay 1s button stops the program for 1 second, and the Delay 1m stops the program for 1 minute- this is for simulating lags or lost links. The localhost button is for getting your local IP address if you go online after you've started the program.

Comms_Housekeeping{} does most of the work now, and returns true if the program should exit- ie: it's disconnection is confirmed by the Server.

Window_Events checks for input from the user and acts accordingly, it also returns true if the program should exit- ie: if the user clicks on the close gadget while offine.

The Net Protocol Header defines the hex numbers that tell what each packet is, so you can decode the data in each packet. I haven't fully documented the protocol on computer file yet, but it's pretty self-explanatory.

You'll notice I've put in some security measures, ie: it checks if the packet has come from a logged-in host (rather than having a player number in the packet- saves data too), so it stops people sending fake messages to try and cheat. People could still send fake IP packets if they knew what they were doing, so I might put in a security number in each packet, that is random for each connection between each of the players. Even basic encryption might be needed as well. Another (and the best) way, would be to use the Secure Sockets Layer available in Miami.

Also the Security_Warning{} statement allows people to have security messages printed out in their game- you can chose none, all, medium, or only serious security warnings. This allows people to keep an eye out for attempted cheating. This routine can be customised to print messages out however you like in your game- at the moment it just calls Print_String{}.

1.9 arexx port

It has an Arexx port (Arexx port name is 'UDP_Chat') so it can be told to log in to a Host from AmiComSys, using the small Rexx program UDP_Chat.rexx. All you have to do, to log into a Server from AmiComSys is click on the person's name, who is going to be the Server and go to the Rexx menu in AmiComSys and run UDP_Chat.rexx, and UDP_Chat will automatically log into the UDP_Chat program being run on the other person's machine. You can add this Rexx script to the Arexx menu in AmiComSys with 'Add Arexx Script' in the Arexx menu, for ease of use.

You need to change the location of your UDP_Chat executable at the top of the Arexx script, so it will run UDP_Chat if it's not already running (because the default directory seems to be AmiComSys:).

You could also use this Rexx port with programs like QAmiTrack (on Aminet, as is AmiComSys), or STRICQ <http://www.momo2000.com/~mclaughd> (you would need to do some extra coding to use it). Don't use DC-ICQ, as the group responsible are crackers/pirates, and may be using it to get passwords from your hard-drive.

```
  Arexx Commands:

CONNECTTOSERVER Hostname.s : this tells UDP_Chat to attempt to log
  into another UDP_Chat which is acting as Server. The second parameter
  (string) is the Hostname to connect to.

ISONLINE : This asks UDP_Chat it's online state. UDP_chat responds with
  a string- either 'Offline', 'Server' or 'Client' depending on it's
  online status. I haven't tested this yet, as I didn't have time to
  work out how to receive a string in a Rexx script.

QUIT :  This tells UDP_Chat to exit.

   More commands could be easily added- just add them to the
Get_Rexx_Message{} function. UDP_Chat's Arexx port is 'UDP_Chat'.
```

1.10 things to do

Each player needs to send out a 'heartbeat' if they haven't sent a packet for 5 seconds, so other players know they're still online.
Maybe it should check through all the unacknowledged packets for one that's overtime, rather than just the oldest one, as it does at the moment.
The messages() array might need to be made larger if you're sending lots of messages out to 7 other players (maybe to 256 or bigger- adjustible by the #MESSAGE_ARRAY_SIZE constant).

It needs CIA interupt code to make sure it has an accurate timer, using the highest priority CIA (CIA-B, timer B from memory). At the moment, the timer could be reset which would screw things up, and also the time taken for one tick changes, depending on screenmode (I think it's either NTSC (60 ticks/second) or PAL (50 ticks/second)- even if you're using a Graphics card).

Also, changing the string handling routines to Peeking/Poking or assem would speed up the routines a lot (although they seem fairly fast at the moment).

I'll have to put in checking routines, to make sure all packets are received in order- as you wouldn't want to execute packet 2, and then packet 1 (if 1 had to be resent), so I'll have to buffer the incoming packets until the earlier packet is resent- this is why Quake gets lagged badly if you have a bad connection (lots of lost/broken packets). Also putting all incoming packets into a buffer will allow me to do what Quake does, in that all single-player movements (the person playing on that machine) are put into the buffer as well, and then they're executed the same as the movements from the other players.

I'll add routines so you can either send messages by Peer-to-Peer, or by Client-Server, or both in the same game if you wanted.

Then it would be ready to put into a small test game- yay!

1.11 theory

```
                   See Paul Burkey’s Net web page for more info on Internet game
connections: <http://www.sneech.freeserve.co.uk/netlink.html> . It’s a
complex subject that requires some research to understand all the
problems faced.
   Also see the file included in this archive ‘Lag problems in Net games’,
it is a discussion from the Blitz-List on the problems caused by lags
in multiplayer games, and a couple of solutions to those problems.

   This section currently under construction :-/


                 Basic TCP/IP Theory

                 Multi-Player Connections

                 Security/Prevention from Cheating

                 Advanced Stuff
                   I will be adding in a lot more here, from simple Internet ↩
                      theory, to
multiplayer theory (Client-Server, and Peer-To-Peer connections), through to
advanced stuff, like protection from cheating/hacking, player prediction
to smooth out gameplay etc.
```

1.12 basic tcp theory

```
 Question: How does someone find somebody else on the Internet to have a
game with?

 Answer:  Everytime you go online an IP number (like 230.21.202.1) is
allocated to you (your ISP usually allocates you a free number every time
you log in, so each time you log in your internet address changes- if you
have a permanent connection you can have a permanent IP number). You can
also be found by a name (in my case port101.ww.co.nz-
the 101 number will change each time I log on- this is the same address
as the IP number- the TCP stack translates it to an IP number by checking
it up through a DNS server). You need to know this number or name (this is
your address on the Internet), of the person who is acting as Server (type
into the Send To: gadget in UDP_Chat), so you can log into their Server
program (UDP_Chat acting as Server).
   This number can be found out in a number of ways, if you’re talking
on IRC, you can find their address in the person’s info window (click on
their name and on the query button in AmIRC, and the name on the left-hand
of the @ symbol is their internet address- ie: Anton@max39.ww.co.nz
- max39.ww.co.nz is their address), or with STRICQ, AmiComSys- basically
through any server/program that is permanently online, which everyone can
og into, so people can find out other people’s IP addresses.
  When someone has a Server online, it waits for a log in, on a certain
port number (mine uses 27272), so you have to log into the Server IP
number at the correct port number, at that address. The port number
```

normally is always the same, so that’s not a problem.
Another way is to log in each IP number of an ISP (if you know what ISP the person you’re looking for is on; say to find me- port101.ww.co.nz-port302.ww.co.nz - or whatever the upper number limit is (basically how many lines in/modems they have)- note that the ’port’ in the above address has nothing to do with the port number I mentioned above- it could be any word(s) really), and try to log in at the correct port number, if you get a response (and correct one- it might be the correct Server program, but wrong person!), then you’ve found the right person, and can log in.

1.13 multi-player connections

There’s two ways of doing multi-player connections, Peer-to-Peer and Client-Server:

With Peer-to-Peer (each player sends messages out to all the other players), you’ll have to figure out how you’ll do collision detection - the best way to do it is for the collision detection for each player’s car to be done by the player’s program, as only one machine can make a decision for one object. For things like power-ups, maybe it would be best for the Server to handle those decisions (and effectively controlling all objects other than player’s cars, and messaging is done Client-Server). Also these messages would have to be sent reliably, as you couldn’t afford to lose information about picking up a power-up etc.
To prevent graphical problems with higher lags, you could add an artificial lag to your player’s movements, say based on the lag to the Server minus, say 50-100 ms.

With Client-Server, (you send all messages to the Server which then sends them all out to the players). This is good because your player doesn’t move until it receives the message back from the Server, so all objects move around the same on all machines (assuming lags don’t vary too much). (With a racing game this is less of a problem). Collision detection is easy because the server does it all, but if the server is an Amiga, then that would put extra load on the machine (as it would have to run the game as well)- you would need a powerful machine for it. Also you would need a decent modem because the server has a lot of data coming in and out- it takes a load off the individual player’s machines though.
The main problem is, that Quake servers etc are usually permanent at an ISP and have a direct link to the net, so they don’t add any appreciable lag to the messaging; but if the server is home computer with an analogue modem (which in your game is likely), then the message going through the analogue modem and out again will add a large amount to lag (for a 300ms lag, half of that will be the modem! So if your game was Client-Server it would add another 150ms to the lag!)

1.14 security

For a start, my UDP_Funcs check the address and port number of every packet sent, and discard it if it doesn’t match up to a player

that's already logged on- this stops any person from grabbing my UDP_Funcs and sending fake/destructive packets- they'd have to really know what they were doing, as they'd have to send RAW packets (make their own packets up with a fake header), to screw my routines over.
More stuff can be done to improve security obviously!

1.15 advanced stuff

One thing Quake does is send out packets 10-20 times a second with the player's position etc, this is good for games like Quake, so if you get a bad lag, your player doesn't carry on moving and end up in the Lava!! Obviously you wouldn't need to do that many packets a sec, maybe 1,2 or 5 a sec maybe; which would keep the bandwidth down- Quake was originally designed for LANs not the net, they made QuakeWorld so Quake was more playable on the net, and it reduces the bandwidth used so you can have more players- up to 64 now with MegaQuake.

1.16 history

```
 Version 2.3  7/3/1999
------------
```

I've put in the auto-disconnect code, for when a player stops responding. If a player can't communicate when another player it tells the server this- then the Server checks this fact and tells all the players if this is true. This is for the rare case that two players have a connection to a third, but only one of those players loses connection to the third one. If the server can communicate with the player that's thought to be offline, then the player that initiated the call gets disconnected instead. This is just some simple security, to stop cheating by getting the other person disconnected.
I've tidied up the main loop and taken most stuff out of it- most of the work is now done by the Comms_Housekeeping{} Function. It's now much easier to slot these functions into your game.
Added the Clear_Player_Arrays Statement to re-initialise a disconnected player's data.
Fixed some disconnecting bugs, and now Requested_Connection{} checks wether the host and port are already connected (it was possible before to let a host log in twice- if you got the timing wrong!)
Added Security_Warning{} to allow people to have security messages printed out in their game- you can chose all, medium, or only serious security warnings.
Some repeated code in Comms_Housekeeping{} and Acknowledge_Packet{} has been shifted to Resend_Message{}.

```
 Version 2.2  14/11/1998 (internal version- not released)
------------
```

The disconnect code has been put in, and I tidied up the Decode_Packet and Requested connection functions (no more passing variables between them through Global variables- naughty :).
Also I've changed the Packet protocol a bit- all packets now have a

```
packet number (.l) in the front, wether they need it or not, as it makes
the encoding and decoding of packets much simpler.
  Acknowledge_Packet now checks wether the packet number is in correct
bounds to prevent it getting in a loop, if a bad packet is received.
  If the Server closes, it sends the #GAME_END message to all players,
if a player closes, it sends #CP_REQ_PLAYER_DISCONNECT with it’s player
number to the Server, then the Server informs all the players (including
the one that sent the disconnect message) that the player has quit, then
the player quits when it receives the disconnect message with it’s
player number.
  Once a Client or Server has started to quit, it will quit after a certain
time-out, even if it doesn’t get all the responses it required.

 Version 2.1  02/10/1998
------------
  Fixed the dead link bug- if after 5 resends there’s no reply, it now
acknowledges that packet and stops resending-
( Acknowledge_Packet{message_number} on line 649 should have been
Acknowledge_Packet{last_message_number} )- doh
  Put docs into AmigaGuide- yay :-)  - and added some things to the
‘Things to Do’ list.

 Version 2.0  27/07/1998
------------
  Changed printing of Ping times, so they are now printed out correctly for
each player, and you can now tell what your player number now is (‘Me’ is
printed in place of your Ping time). Before it just printed out the
Ping time of the last packet received. The Server is always Player 1. I
fixed a bug which was numbering the clients wrongly, at the same time.
  Put in a ARexx interface, so you can log into a Server from AmiComSys,
by passing the host address to UDP_Chat from AmiComSys (through a small
Rexx program).
  The Function Connect_UDP{} has been renamed to the more logical
Initialise_UDP{}, as it doesn’t actually connect to anything.  It now
closes the bsdsocket.library in Initialise_UDP{} instead of at the end
of the program- Paul had done this in his TCP_Funcs, but I hadn’t
transfered his code across properly. :-/
  I renamed the programs to their actual version number, rather than
the quick filename I gave them (ie: UDP_ChatV1.9.bb2, rather than
UDP_Chat4.bb2)- doh!
  Changed Close_UDP{} to Exit{} and put it back in UDP_Chat. It now
deallocates the receive memory buffer (UDP_mem.l) allocated in
UDPHeader.bb2, and closes the Rexxport and UDP socket if they’ve been
opened, (tidier than having it all at the end). So you can just call
Exit{} to exit from anywhere (with an error string if needed), rather than
have those naughty Gotos! :)  Initialised ‘sock.l’ to -1 so Exit{} routine
can work
   Added in a #DEBUG constant to switch on debug info, and allow multiple
copies of UDP_Chat on one machine (currently all debug info is still
printed out).
   Changed default Port number to 27,272- (3,001 was too low for practical
use). Put ‘ypos’ above ‘GoSub Init_Gui’, so can now call Exit{} from
Init_Gui.
   Added this History File!

 Version 1.9   24/04/1998
------------
```

```
  Original Aminet release. Several bugs known, and a several features
not yet implemented.
```

1.17 thanks to

```
Thanks to Paul Burkey for TCP_Funcs.

        <paulb@sneech.demon.co.uk>

        <burkey@bigfoot.com>

        <http://www.sneech.demon.co.uk>


Alvaro Thompson for TCP_GUI

        <alvaro@enterprise.net>


David Newton for the Arexx example

      <dave@nbsamiga.demon.co.uk>


and to Dr. Ercole Spiteri for TCP-to-Blitz.

        <ercole@maltanet.omnes.net>

        <ercole@usa.net>
```

1.18 author and usage

```
   Feel free to use these routines in your program- but give me a mention
in your credits for these routines :-)
  If your game's shareware, can you send me a full copy of your game
as well please :-)
  If your game's commercial, can you contact me first, and we can come to
some arrangement- don't worry, I don't expect much considering the current
Amiga games scene- maybe it'll be just the same as for shareware :-)


         UDP_Chat is written by  Anton Reinauer.

 Any questions/comments to:   <anton@ww.co.nz>

   My web page:  <http://www.ww.co.nz/home/anton/>


  Have fun now kiddies :-)
```