Pawn

embedded scripting language

Datagram Exchange Module

August 2006

Abstract

The "Datagram Exchange Module" adds functions to the PAWN scripting language for sending and receiving "datagrams" (or *packets* over a communication link. Currently, only the UDP protocol from the TCP/IP suite is supported.

The software that is associated with this application note can be obtained from the company homepage, see section "Resources"

Typeset with TEX in the "Computer Modern" and "Palatino" typefaces at a base size of 11 points.

Introduction

The "PAWN" programming language depends on a host application to provide an interface to the operating system and/or to the functionality of the application. This interface takes the form of "native functions" —a means by which a PAWN script calls into the application, and "public functions" —for the host application calling into the script. The PAWN "core" toolkit neither mandates nor defines *any* native/public functions (the tutorial section in the manual uses only a *minimal* set of native functions in its examples). In essence, PAWN is a bare language to which an application-specific library must be added.

That non-withstanding, the availability of general purpose native-function libraries is desirable. The "Datagram Exchange Module" discussed in this document intends to be such a general-purpose module.

This application note assumes that the reader understands the PAWN language. For more information on PAWN, please read the manual "The PAWN booklet — The Language" which is available from the company homepage.

Network protocols

The TCP/IP protocol suite is a series of protocols for transferring data over a network connection, that partially live side by side and partially build upon each other. The low level protocols are IP, ICMP, ARP and a few others. At the medium level one finds TCP and UDP and among the high level protocols one finds the familiar HTTP, FTP, POP3, SMTP, etc.

UDP and TCP are both medium level protocols. The difference between UDP and TCP are that UDP only knows how to transfer individual packets (with some maximum size) and that it lacks a "acknowledge" mechanism to ensure that packets that are send to also arrive. In contrast to TCP, UDP is also a *connectionless* protocol, which means that it can talk to multiple clients at the same time, or broadcast packets. UDP has also less overhead than TCP, due to the absence of any delivery control. TCP is more suitable for exchanging files or (large) streams of data between two end-points, whereas UDP is more suitable to send short messages or packets to one or more recipients, especially is these packets must be transferred over the network quickly. However, if you wish to make sure that a packet that was sent also arrives, you must implement your own handshaking protocol with UDP.

The "Datagram Exchange Module" supports the UDP protocol. It allows sending a packet to a specific network address, or to broadcast the packet over the network.

In the TCP/IP protocol, each host has a unique IP address* which is a sequence of four values between 0 and 255. Optionally, a host also has a "domain name": a sequence of the name of the computer followed by that of the network or high level domain, with dots (".") between them.

Since any host can run more than one application that accesses the network, knowing the IP address of another computer is not enough: you must also know the *port number* of the process on the other computer that understands the data that you are sending. The port number is a single number between 1 and 65535, but many of the low numbers are already reserved by standard processes and protocols. For example, FTP is 21, SMTP is 25, HTTP is 80, POP3 is 110.

In the "Datagram Exchange Module", the IP address and the port number are combined into a single address, which has the format "`127.0.0.1:9930`". The first four numbers are the IP address (address 127.0.0.1 is the "loop-back address", by the way). The fifth number is the port number to send to; it is separated from the IP address by a colon (":"). The port number 9930 is the default port for the Datagram Exchange Module.

Either the IP address or the port number may be removed. If you send to "`192.160.0.1`", the packet goes to that address over the default port 9930. If you set the address "`:1234`", the packet will be broadcast over the network on port 1234 (note that the colon must be present). When neither the IP address nor the port number are present, the packet would be broadcast on the default port 9930.

The UDP protocol does not support streams of data, only individual packets. The maximum size of a packet is not standardized. On Ethernet, the maximum packet size is defined at 1500 bytes, but when the packet goes over the Internet a save maximum is 546 bytes.

---

* This address must be unique within the network that the host operates; depending on how the network is set up, the IP address need not be unique throughout the world. For example, a LAN is typically linked to the Internet through a router that does NAT-lookup, and thereby shields the "local" IP addresses from the outside world. Two LANs that are each behind a router may use the same local IP addresses, without this giving a conflict.

Implementing the library

The "Datagram Exchange Module" consists of the two files AMXDGRAM.C and DATAGRAM.INC. The C file may be "linked in" to a project that also includes the PAWN abstract machine (AMX.C), or it may be compiled into a DLL (Microsoft Windows) or a shared library (Linux). The .INC file contains the definitions for the PAWN compiler of the native and public functions in AMXDGRAM.C. In your PAWN programs, you may either include this file explicitly, using the `#include` preprocessor directive, or add it to the "prefix file" for automatic inclusion into any PAWN program that is compiled.

The "Implementer's Guide" for the PAWN toolkit gives details for implementing the extension module described in this application note into a host application. The initialization function, for registering the native functions to an abstract machine, is `amx_DGramInit` and the "clean-up" function is `amx_DGramCleanup`. Calling the clean-up function before terminating/unloading the extension module (or the host application using it) is recommended.

If the host application supports dynamically loadable extension modules, you may alternatively compile the C source file as a DLL or shared library. No explicit initialization or clean-up is then required. Again, see the Implementer's Guide for details.

Usage

Depending on the configuration of the PAWN compiler, you may need to explicitly include the STRING.INC definition file. To do so, insert the following line at the top of each script:

```
#include <datagram>
```

The angle brackets "<...>" make sure that you include the definition file from the system directory, in the case that a file called STRING.INC or STRING.P also exists in the current directory.

From that point on, the native functions from the datagram exchange library are available. Below is an example that comes from the "PAWN booklet — Language Guide". It implements a simplistic chat program where messages that you type are broadcasted over the network and received by other hosts that run this script. At the same time, you will also receive messages that these other hosts send.

Listing: **chat.p**

```
#include <datagram>

@receivestring(const message[], const source[])
    printf "[%s] says: %s\n", source, message

@keypressed(key)
    {
    static string[100 char]
    static index

    if (key == '\e')
        exit                    /* quit on 'Esc' key */

    echo key
    if (key == '\r' || key == '\n' || index char == sizeof string)
        {
        string{index} = '\0'    /* terminate string */
        sendstring string
        index = 0
        string[index] = '\0'
        }
    else
        string{index++} = key
    }

echo(key)
    {
    new string[2 char] = { 0 }
    string{0} = key == '\r' ? '\n' : key
    printf string
    }
```

For an explanation of the keyboard input functions of this script, please see the "Language Guide". All scripts that use the datagram exchange module will be event-driven. See the language guide for a tutorial on the event-driven programming model.

Several functions have a parameter that specifies the maximum number of *cells* that a destination buffer can hold. The purpose of this parameter is to avoid an accidental buffer overrun. Note that this parameter *always* gives the buffer size in *cells*, even for packed strings. The rationale behind this choice is that the `sizeof` operator of PAWN also returns the size of buffers in cells.

Public functions

@receivepacket A packet was received

Syntax: `@receivepacket(const packet[], size, const source[])`

`packet`	Contains the packet that was received.
`size`	Contains the number of *bytes* (not cells) that are in the packet.
`source`	Contains the IP address and the port number of the sender of this packet.

Returns: The return value of this function is currently ignored.

See also: `@receivestring`, `sendpacket`

@receivestring A packet that contains a string was received

Syntax: `@receivestring(const message[], const source[])`

`message`	Contains the message (a zero-terminated string) that was received.
`source`	Contains the IP address and the port number of the sender of this packet.

Returns: The return value of this function is currently ignored.

Notes: The string is in unpacked format if the original packet contained a string in UTF-8 format. Note that messages in the ASCII character set are also UTF-8 compliant.

See also: `@receivepacket`, `sendstring`

Native functions

listenport — Sets up the port number to listen at

Syntax: `listenport(port)`

`port` — The number of the port to listen at. This must be a value between 1 and 65535, but you should probably avoid to use any of the reserved port numbers.

Notes: You must call this function *before* receiving the first packet. In other words, you should set up a port in `main`.

If no port number has been explicitly chosen, the module will listen at port 9930.

Returns: This function always returns 0.

sendpacket — Sends a packet

Syntax:
```
sendpacket(const packet[], size,
           const destination[]="")
```

`packet` — The buffer that contains the packet to send.

`size` — The size of `packet` in *bytes* (not cells).

`destination` — The IP address and port number to which the packet must be sent. If absent or an empty string, this function will broadcast the packet and use the default port number 9930.

Returns: `true` on success, `false` on failure.

See also: `@receivepacket`, `sendstring`

sendstring — Sends a packet containing a string

Syntax: `sendstring(const message[], const destination[]="")`

`packet` The buffer that contains the string to send. If this is an unpacked string, it will be UTF-8 encoded before being transferred.

`destination` The IP address and port number to which the packet must be sent. If absent or an empty string, this function will broadcast the packet and use the default port number 9930.

Returns: `true` on success, `false` on failure.

See also: `@receivestring`, `sendpacket`

Resources

The PAWN toolkit can be obtained from **www.compuphase.com** in various formats (binaries and source code archives). The manuals for usage of the language and implementation guides are also available on the site in Adobe Acrobat format (`PDF` files).

Documentation on UTF-8 is in the File I/O extension module. For sending binary data, you may want to use UU-encoding; more information on UU-encoding and native functions to encode and decode are in the String Manipulation library. The File I/O module and the String Manipulation library are two other general purpose extension modules for PAWN.

Index

◇ Names of persons (not products) are in *italics*.
◇ Function names, constants and compiler reserved words are in `typewriter font`.