Gemini UDP Protocol Specification

Introduction

This document is Gemini's UDP protocol specification and establishes the processes that clients must implement in order to use the protocol. The protocol has been developed by René Goerlich, Paul Kanevsky and Peter Simpson for use with the Gemini 2 telescope control system, it is not and cannot be supported by the earlier Gemini 1 hardware and v4.1 firmware.

The UDP network protocol is described as are the reasons for its selection over TCP; some .NET example code is also provided in Appendix 2. For support please post on the Gemini II Yahoo group at: http://tech.groups.yahoo.com/group/Gemini-II/messages

What is UDP?

This section has been reproduced from WikiPedia, please see document licensing section for further information.

The **User Datagram Protocol** (**UDP**) is one of the core members of the Internet Protocol Suite, the set of network protocols used for the Internet. With UDP, computer applications can send messages, in this case referred to as *datagrams*, to other hosts on an Internet Protocol (IP) network without requiring prior communications to set up special transmission channels or data paths. The protocol was designed by David P. Reed in 1980 and formally defined in RFC 768.

UDP uses a simple transmission model without implicit handshaking dialogues for providing reliability, ordering, or data integrity. Thus, UDP provides an unreliable service and datagrams may arrive out of order, appear duplicated, or go missing without notice. UDP assumes that error checking and correction is either not necessary or performed in the application, avoiding the overhead of such processing at the network interface level. Time-sensitive applications often use UDP because dropping packets is preferable to waiting for delayed packets, which may not be an option in a real-time system.[1] If error correction facilities are needed at the network interface level, an application may use the Transmission Control Protocol (TCP) or Stream Control Transmission Protocol (SCTP) which are designed for this purpose.

UDP's stateless nature is also useful for servers answering small queries from huge numbers of clients. Unlike TCP, UDP supports packet broadcast (sending to all on local network) and multicasting (send to all subscribers).[2]

Common network applications that use UDP include: the Domain Name System (DNS), streaming media applications such as IPTV, Voice over IP (VoIP), Trivial File Transfer Protocol (TFTP), IP tunneling protocols and many online games.

UDP applications use datagram sockets to establish host-to-host communications. An application binds a socket to its endpoint of data transmission, which is a combination of an IP address and a service port. A port is a software structure that is identified by the port number, a 16 bit integer

value, allowing for port numbers between 0 and 65535. Port 0 is reserved, but is a permissible source port value if the sending process does not expect messages in response.

Why UDP, why not TCP?

Two key requirements for the Gemini protocol are:

- It is lightweight: it places low demand on the network and Gemini hardware / OS
- It is responsive: Gemini receives commands and clients receive responses quickly

Early testing using TCP revealed two issues:

1. Slow responses when using Windows clients
2. High impact on the Gemini CPU when using the built-in TCP stack

These were traced to:

1. The default value of the Microsoft TCP stack TcpAckFrequency setting (2), resulted in a 200ms delay in the round trip time to Gemini because the stack was waiting for a second packet from Gemini before sending the ACK packet for the first packet. Gemini on the other hand was waiting for an ACK to the first packet before sending the second packet... deadlock! The deadlock was always broken after 200ms by the Microsoft stack sending an ACK, even if it had not received a second packet within that time. So the system worked reliably but with frequent delays of 200ms resulting in low throughput.

   When TcpAckFrequency is set to a value of 1, performance is fast, however, it was not felt appropriate to require all Gemini uses to change the Microsoft default setting and therefore to carry the risk that some other aspect of Windows would not perform as expected.

2. The TCP stack provided in the software development kit for the Gemini chipset was found to have a relatively high impact on the Gemini processor, impacting overall responsiveness and throughput.

Testing with UDP was much more successful, providing a higher throughput than TCP, even with TcpAckFrequency set to 1, due to the lower protocol overhead for Gemini and because it bypassed the higher levels of the Microsoft TCP stack at the client. It should be noted that during testing René did not see the 200ms ACK delay because he was using a Linux client, this issue just derived from the Microsoft TCP/IP stack.

Use of UDP does resolve the two key issues above, but it also brings some issues of its own since it does not provide the robust delivery capability and guaranteed byte stream features that TCP offers. The following comparison shows key differences in these areas.

	TCP	UDP
Reliability	TCP manages message acknowledgment, retransmission and timeout. Multiple attempts to deliver the message are made. If it gets lost along the way, the server will re-request the lost part. In TCP, there's either no missing data, or, in case of multiple timeouts, the connection is dropped.	When a message is sent, it cannot be known if it will reach its destination; it could get lost along the way. There is no concept of acknowledgment, retransmission or timeout.
Ordering	if two messages are sent over a connection in sequence, the first message will reach the receiving application first. When data segments arrive in the wrong order, TCP buffers the out-of-order data until all data can be properly re-ordered and delivered to the application.	If two messages are sent to the same recipient, the order in which they arrive cannot be predicted.

In the world of TCP/IP, with which we are most familiar, TCP provides reliable data communication and as application users we do not need to concern ourselves with issues such as packets arriving out of order or being lost in transit. In adopting UDP we lose the benefits of reliability that TCP provides and consequently need to provide these at the application level.

Implementation is straightforward since we only need a simple, low overhead process because we don't need all the frills that TCP provides. The following sections describe Gemini's use of the UDP protocol and it's error handling process flow.

Access to Gemini's UDP Server

Gemini provides a UDP server connection by default on port 11110, there is no security access control, login or similar concept. If you can reach port 11110 then you can just start sending commands to Gemini. The UDP port number used by the Gemini server can be changed through the Network Settings page of the Gemini's built-in web application.

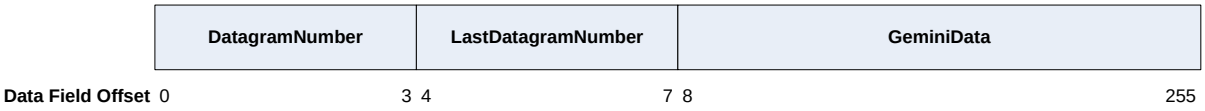
UDP Datagram Format

The UDP protocol datagram format is shown below

Your UDP protocol stack should take care of the first 8 bytes (0 to7), which record the source and destination IP ports, the length of the datagram and a checksum over the entire datagram. The Data

field (shown in blue), starting at offset 8 is what we are really interested in and is what we are working with below.

The Gemini protocol defines its own format for use within the UDP datagram Data field as shown below:



Field	Offset	Purpose
DatagramNumber	0	On sending a command, this will be a unique number for this datagram from this client. It sequentially increments by one for each datagram starting at 0. On receipt of a datagram from Gemini, this will contain the DatagramNumber of the command for which this is the response.
LastDatagramNumber	4	This should be set to zero for commands to Gemini and will normally be zero in responses from Gemini. It is set to the DataGramNumber of the last received command from the client, in response to a NAK command from the client. (See error handling protocol section)
GeminiData	8	Command to Gemini, using the Gemini Serial Command syntax or response from Gemini, in both cases terminated by the NULL character (0x00). The maximum size of the GeminiData field is 256 bytes TBC

Client Protocol

The overall Gemini client - server protocol is set out diagrammatically in Appendix 1.

Sending Individual Serial Commands to Gemini

In these cases the data to be sent is an individual serial command as defined in the serial protocol, which must be terminated with a NULL (0x00) character. In this example you want to send the :GR# command. So, you would determine the next unused DataGramNumber by incrementing that last used DataGramNumber and then construct a GeminiData field comprising:

- `4 bytes of DatagramNumber field`
- `4 bytes of 0x00 for the LastDatagramNumber field`
- `4 characters of serial command :GR#`
- `1 byte of NULL character(0x00) as terminator`

and send it to Gemini as a UDP datagram.

In response, you will receive a datagram containing something like:

- 4 bytes of DatagramNumber field (same as the one you sent)
- 4 bytes of 0x00 for the LastDatagramNumber field
- 9 characters of response 13:45:23#
- 1 byte of NULL character(0x00) as terminator

Sending Multiple Gemini Serial Commands to Gemini

The UDP protocol supports sending multiple Gemini Serial Commands in one datagram. For example, if you send the commands :

- 4 bytes of DatagramNumber field
- 4 bytes of 0x00 for the LastDatagramNumber field
- 17 characters of serial command :GR#:GD#:GS#:GVP#
- 1 byte of NULL character(0x00) as terminator

in one datagram you will receive a response datagram containing something like:

- 4 bytes of DatagramNumber field (same as the one you sent)
- 4 bytes of 0x00 for the LastDatagramNumber field
- 43 characters of response 13:45:23#75:34:09#09:56:09#Losmandy Gemini#
- 1 byte of NULL character(0x00) as terminator

Serial Commands with no Response

Some serial protocol commands have no response e.g. :Q#, :RS# If one of these is sent on its own, there should be no response for the client and the client will be unable to determine that the command has actually been received and acted upon.

To cater for this situation, Gemini will respond with a datagram containing an ACK (0x06) character terminated with a NULL (0x00) character whenever there is no serial command response to return. Thus the client should wait for a response datagram from Gemini under all circumstances. It will either contain the response or the ACK character if there is no serial command response.

If multiple commands are sent, one of which does not have a response, the ACK character will not be returned as the response to the other commands indicates that the responseless command was received and processed OK.

Datagram Reliability - Arrival Order

Since UDP does not guarantee packet arrival order, critical sequences must be managed at the application level. An example of a critical sequence is the "Equatorial Slew" sequence where the Serial Protocol specifies that the :Sr command must be sent before the :Sd command.

This can be achieved in two ways:

1. Ensure that your client always waits for the response to a command before sending the next command, i.e. always have only one datagram "in flight" at a time.
2. Place all critical sequence commands into one datagram using the "Multiple Gemini Serial Commands approach described above. This will ensure that they all either arrive together

and are processed in the order that you placed them in the datagram, or they do not arrive at all and are resent together through the "Errors and Losses" process described below.

Either method will work on its own but using both together is recommended.

A list of critical sequence commands is shown in Appendix 3.

Datagram Reliability - Errors or Losses

The Gemini UDP protocol specifies that every command datagram will receive a response datagram, even if the Gemini serial command does not have an application response. When things go wrong we need to consider two scenarios:

1. Command never reaches Gemini, so no response is sent
2. Command is received by Gemini but response is lost in transit and never reaches the client

From the client's perspective, both scenarios look the same: I sent a UDP datagram and I never received a response datagram so it has no idea whether or not Gemini received and acted on the command. To detect that something has gone wrong, the client needs to have a timeout on its "wait for response UDP datagram" routine.

When this is activated, the client needs to query Gemini to find out why no response was received, fortunately Gemini can supply information that allows the client to determine whether scenario, 1. or 2. above applies.

So, on timing out the client should send a 9 byte NACK datagram to Gemini:

- `4 bytes of DatagramNumber field`
- `4 bytes of 0x00 for the LastDatagramNumber field`
- `1 byte of NACK character(0x15)`

Note 1 - there is no NULL terminator character in this datagram
Note 2 - The DatagramNumber field should be the next unused value in sequence, it must not be any previously used value

Gemini will respond with a datagram like this:

- `4 bytes of DatagramNumber field (same as supplied in NACK datagram)`
- `4 bytes of LastDatagramNumber (DatagramNumber of the last command received)`
- `Response to the command in the LastDatagramNumber datagram`
- `1 byte of NULL character(0x00) as terminator`

The client should examine the LastDatagramNumber field and compare it with the DatagramNumber of the command that timed out.

- If it is the same, Gemini did receive the command and the response was lost en-route from Gemini to the client. The required response is now in this datagram so it can be returned to the client application method that initiated the command. Job done!

- If it is different, then Gemini never received the command in the first place, so the client should resend the command that timed out, using a new DatagramNumber, and return the response to that datagram to the calling method. Job done!

Under network loss conditions, Gemini may time out on several consecutive attempts to retrieve the command response. Under these conditions, the communications level should throw an exception back to the application indicating loss of communication to Gemini, so it can trigger appropriate messages to higher level applications or the user interface and logs.

Macro Commands

For convenience, if you send a datagram that just contains the ACK commands shown below, you will receive the composite responses shown.

Command	Response
0x05	Coordinates PRA, PDEC (as integer values) , RA, DEC, AZ, EL (as double values), separated by a semicolon. e.g.: 1113128;1152000;3.805914;+90.000000;360.000000;+51.078611;T;W;
0x05 0x01	Do we have any other macros defined?

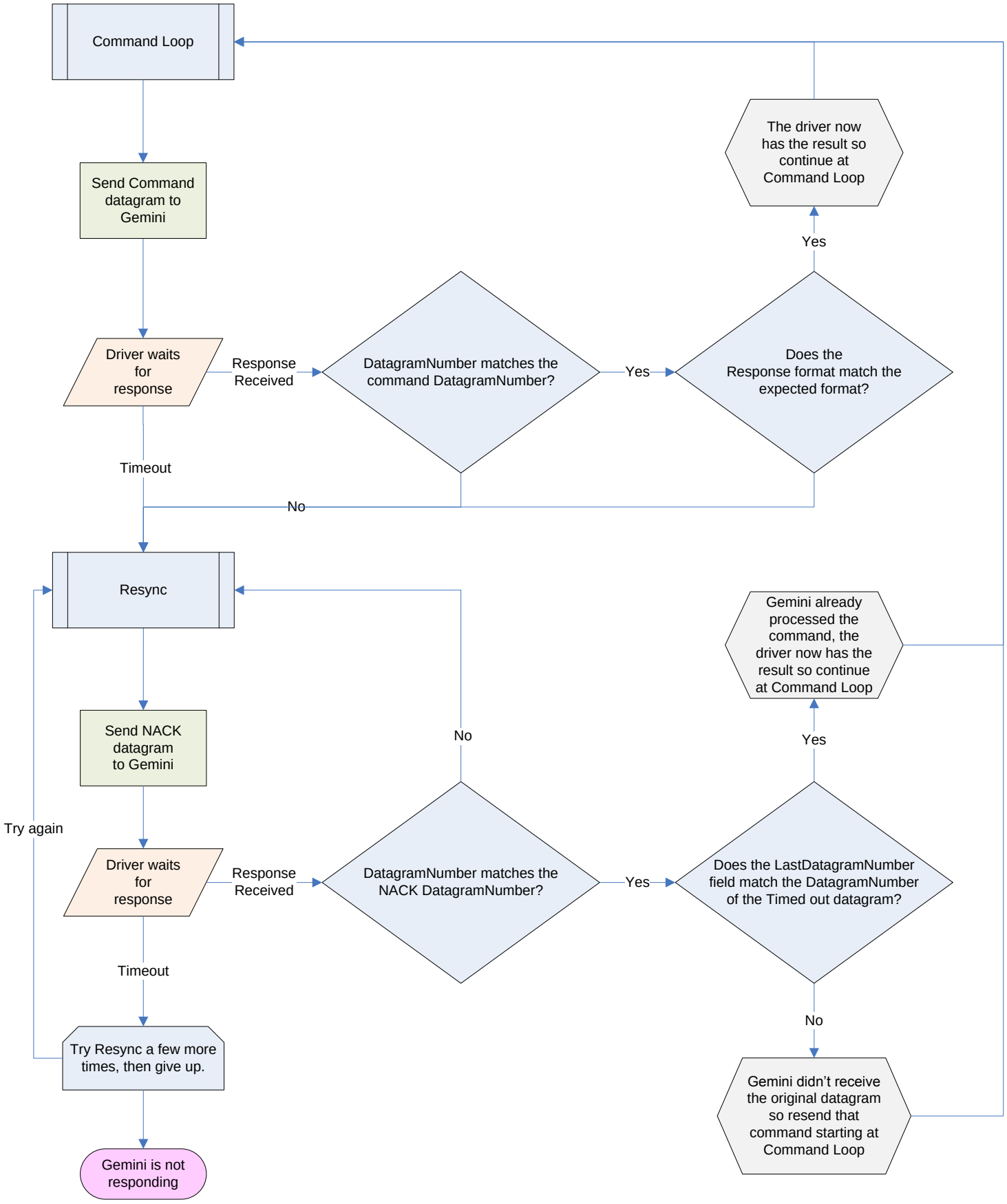
Document Licensing

This document contains text from WikiPedia http://en.wikipedia.org/wiki/User_Datagram_Protocol

Document History

Version	Changes
0.1 Draft	Initial draft for review

Appendix 1 - Client Protocol Flowchart



Appendix 2 - Some .NET Example Code

```
Imports System.Net
Imports System.Net.Sockets
Imports System.Text

Dim hostinfo As IPHostEntry, UDPSocket As Socket
Dim TransmitBytes(255), ReceiveBytes(255) As Byte, NumberOfBytes As Integer, ReceiveString As String

Const GeminiCommand As String = ":GVP#"
Const DatagramNumber As Integer = 0
Const LastDatagramNumber As Integer = 4
Const GeminiData As Integer = 8
Const LengthOfDatagramNumber As Integer = 4

Try
    'Open a UDP socket to the Gemini server
    hostinfo = Dns.GetHostEntry("gemini") ' Find the IP address of Gemini
    UDPSocket = New Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp) ' Create a socket
    UDPSocket.Connect(hostinfo.AddressList, 11110) 'Connect the socket to the Gemini server
    UDPSocket.ReceiveTimeout = 2000 'Set the receive timeout to 2 seconds

    'Set the DatagramNumber to 1 and the LastDatagramNumber to 0
    Array.Copy(BitConverter.GetBytes(1), 0, TransmitBytes, DatagramNumber, LengthOfDatagramNumber)
    Array.Copy(BitConverter.GetBytes(0), 0, TransmitBytes, LastDatagramNumber, LengthOfDatagramNumber)

    'Add the command to be sent
    Array.Copy(Encoding.ASCII.GetBytes(GeminiCommand), 0, TransmitBytes, GeminiData, GeminiCommand.Length)

    'Add the trailing CHR(0)
    TransmitBytes(GeminiCommand.Length + GeminiData) = 0

    'Send the command to Gemini
    UDPSocket.Send(TransmitBytes, DatagramNumber, GeminiCommand.Length + GeminiData + 1, SocketFlags.None)

    'Receive the returned informaiton from Gemini
    NumberOfBytes = UDPSocket.Receive(ReceiveBytes)

    'Convert the response to a string and display it
    ReceiveString = Encoding.ASCII.GetString(ReceiveBytes, GeminiData, NumberOfBytes - GeminiData)
    ReceiveString = ReceiveString.TrimEnd(Chr(0)) 'Remove the trailing CHR(0) added by Gemini
    MsgBox("Gemini response to " & GeminiCommand & ": " & ReceiveString)

    'Close the UDP socket connection
    UDPSocket.Close()

Catch ex As Exception
    MsgBox(ex.ToString)
End Try
```

Appendix 3 - Critical Sequence Commands

- Os with :Od
- :OS with :OR
- :RC with :MA
- :RC/RG/RM/Rm/RS with :Me/Mw/Mn/Ms/Ma/Mi/Mg
- :SG with :SL/SC
- :Sr with :ON with :Sd (This order is recommended in the serial commands description)
- :Sz with :ON with :Sa

ToDo: There are probably some Gemini native commands as well...