Network Working Group
Internet-Draft
Intended status: Experimental
Expires: 27 October 2026

G. G00se
Independent

25 April 2026

The HONK Protocol: Word-Counting over TCP with Honk-Based Responses and Optional Obfuscation

draft-g00se-honk-00

Abstract

This document defines the HONK protocol, a simple application-layer protocol operating over TCP. A HONK client submits a stream of UTF-8 encoded text terminated by a CRLF sequence, and the HONK server responds with a sequence of HONK tokens. The token count equals twice the number of whitespace-delimited words detected in the input. If no words are detected, the server responds with exactly three HONK tokens. The protocol includes an optional Privacy Mode in which message content is obfuscated using single-byte XOR with the fixed key value 0x48 (the ASCII code point for the letter H). This document requests that IANA assign TCP port 24565 to the HONK service.

This document is an individual submission. It is NOT an April Fools Day publication. The author requests serious technical consideration from the IETF community.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 27 October 2026.

Table of Contents

1. Introduction

Many application-layer protocols accept text from a client and return structured feedback. The HONK protocol defines one such exchange: a client submits a line of UTF-8 text, and the server acknowledges the submission by emitting a number of HONK tokens proportional to the word count of that line.

The design goals of the HONK protocol are:

* Simplicity: A conformant implementation requires only a TCP socket, a UTF-8 decoder, a whitespace tokenizer, and the ability to write a line of ASCII text.

* Clarity of signal: The HONK response is distinctive and human-readable. An operator monitoring a session can immediately confirm the server is functioning.

* Interoperability: TCP transport and UTF-8 encoding make the protocol accessible to any platform capable of opening a TCP connection.

* Optional obfuscation: Privacy Mode reduces the legibility of HONK traffic to casual observers. Privacy Mode is explicitly NOT a security mechanism and does not provide cryptographic confidentiality; see Section 8.

Practical applications include network-accessible word-count feedback services, smoke-test endpoints in text processing pipelines, and educational demonstrations of application-layer protocol design and implementation.

2. Conventions and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

HONK token The four-character ASCII string "HONK" (0x48 0x4F 0x4E 0x4B).

Request A single line of UTF-8 encoded text submitted by a HONK client, terminated by CRLF. In Privacy Mode, the line content is XOR-obfuscated before transmission.

Response A server-generated line of one or more HONK tokens

separated by SP and terminated by CRLF. In Privacy Mode, the line content is XOR-obfuscated before transmission.

Word A maximal non-empty sequence of Unicode code points none of which have the Unicode property White_Space=Yes per Unicode Standard Annex 44. The CRLF terminator is not part of the Request body and MUST NOT be counted.

Honk count (N) The number of HONK tokens in a Response, computed per Section 4.4.

CRLF The two-octet sequence CR LF (0x0D 0x0A). CRLF is the line terminator for all HONK messages and is never obfuscated, even in Privacy Mode.

Privacy Mode An optional session mode negotiated at connection open. Request and Response content bytes are XOR'd with the key K = 0x48. Privacy Mode does not provide cryptographic confidentiality. See Section 5.

Obfuscation key (K) The fixed single-octet value 0x48, the ASCII code point for the letter H. Chosen for its mnemonic association with the word HONK.

3. Protocol Overview

The HONK protocol uses a request-response model over a persistent TCP connection. Two session modes exist: Standard Mode and Privacy Mode. The mode is set at connection open and applies for the full lifetime of that connection.

3.1. Standard Mode

1. Client establishes a TCP connection [RFC0793] to server port 24565.

2. Client transmits a Request: UTF-8 text followed by CRLF.

3. Server counts words, computes N, and sends a Response of N HONK tokens.

4. Steps 2 and 3 MAY repeat. Either party MAY close the connection at any time.

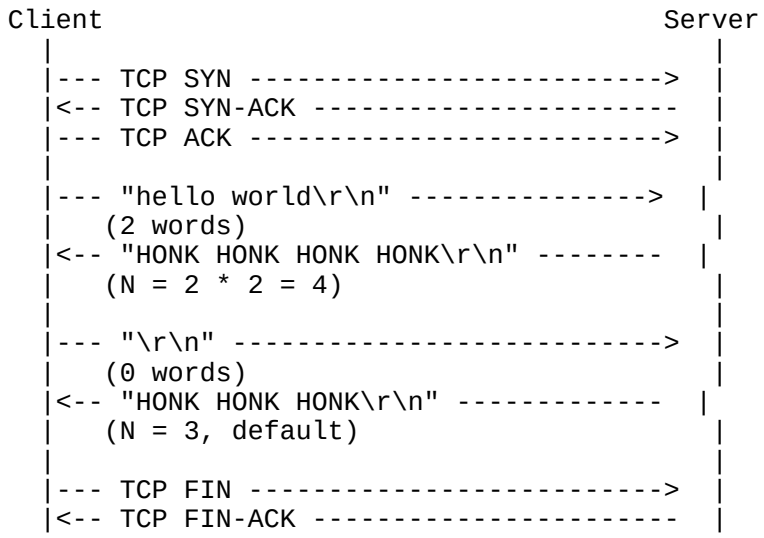

Figure 1: Standard Mode Exchange

3.2. Privacy Mode

The client sends "HONK PRIV\r\n" as the very first transmission. The server acknowledges with "HONK PRIV\r\n". Both lines are cleartext. All subsequent Request and Response content on that connection is XOR-obfuscated with K = 0x48. CRLF terminators are never obfuscated.

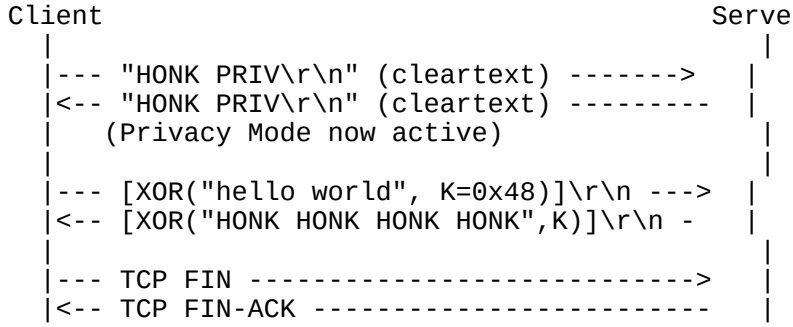

Figure 2: Privacy Mode Exchange

Privacy Mode MUST be negotiated before any Request is sent. A server receiving any line other than "HONK PRIV\r\n" as the first line operates in Standard Mode for the lifetime of that connection.

4. Message Format

4.1. Request Format

A Request is a UTF-8 encoded line terminated by CRLF. The body (everything before the CRLF) MUST be valid UTF-8 per [RFC3629]. The CRLF is not part of the body and MUST NOT be included in word counting. In Privacy Mode, the body bytes are obfuscated per Section 5.3 before transmission; the CRLF is always transmitted as cleartext.

The following grammar uses the ABNF notation defined in [RFC5234]:

```
request      = request-body CRLF
request-body = *UTF8-char
UTF8-char    = UTF8-1 / UTF8-2 / UTF8-3 / UTF8-4
UTF8-1       = %x00-7F
UTF8-2       = %xC2-DF UTF8-cont
UTF8-3       = %xE0 %xA0-BF UTF8-cont /
               %xE1-EC 2UTF8-cont /
               %xED %x80-9F UTF8-cont /
               %xEE-EF 2UTF8-cont
UTF8-4       = %xF0 %x90-BF 2UTF8-cont /
               %xF1-F3 3UTF8-cont /
               %xF4 %x80-8F 2UTF8-cont
UTF8-cont    = %x80-BF
CRLF         = %x0D %x0A
```

Implementations SHOULD impose a maximum request body length. The RECOMMENDED maximum is 65535 octets excluding the CRLF. A server MAY close the connection if this limit is exceeded. A server receiving invalid UTF-8 (after de-obfuscating in Privacy Mode) MUST close the connection without a Response.

4.2. Response Format

A Response is N HONK tokens separated by SP and terminated by CRLF, where N is computed per Section 4.4. In Privacy Mode, the response content bytes are obfuscated per Section 5.3 before transmission.

```
response    = honk-token *(SP honk-token) CRLF
honk-token  = "HONK"
SP          = %x20
CRLF        = %x0D %x0A
```

4.3. Word Counting

1. Interpret the request body as Unicode code points in UTF-8.

2. Partition the sequence into runs of whitespace and non-whitespace, where a code point is whitespace if and only if it has Unicode property White_Space=Yes per Unicode Standard Annex 44.

3. Each maximal non-empty run of non-whitespace code points is one word.

4. The word count W is the total number of such runs.

A body consisting entirely of whitespace has W = 0 and MUST receive the default response of three HONK tokens.

4.4. Honk Count Computation

Let W be the word count. The honk count N is:

* If W is greater than zero: N = 2 * W

* If W equals zero: N = 3

Implementations SHOULD impose a maximum N. The RECOMMENDED maximum is 65535. A server MAY cap N at its configured maximum without signaling an error to the client.

5. Privacy Mode

5.1. Purpose and Limitations

Privacy Mode provides lightweight obfuscation of HONK message content. It is designed solely to reduce readability of HONK sessions to casual, non-targeted observers such as automated log scanners not specifically looking for HONK traffic.

Privacy Mode does NOT provide cryptographic confidentiality, integrity, or authentication. The key K = 0x48 is fixed and publicly specified in this document; any party with access to this specification can immediately de-obfuscate all Privacy Mode traffic. Furthermore, because HONK Responses consist entirely of predictable tokens, Privacy Mode is trivially broken by a known-plaintext attack using a single observed Response. Operators requiring genuine confidentiality MUST use TLS [RFC8446] or an equivalent mechanism.

5.2. Mode Negotiation

A client wishing to use Privacy Mode MUST send the following as the very first transmission on the connection:

```
priv-request  = "HONK PRIV" CRLF
```

A server receiving "HONK PRIV\r\n" as the first line MUST respond with:

```
priv-response = "HONK PRIV" CRLF
```

Both lines are cleartext. After the server sends "HONK PRIV\r\n", Privacy Mode is active for the remainder of that connection. A client MUST NOT send "HONK PRIV\r\n" after sending any Request on the same connection.

5.3. Obfuscation Transform

Every octet of a Request or Response body is XOR'd with K = 0x48 before transmission. The transform is its own inverse: applying it twice recovers the original content.

Let B = (b_0, b_1, ..., b_{n-1}) be the body octets. The obfuscated sequence B' is defined as:

```
    b'_i = b_i XOR 0x48     for all i in [0, n-1]
```

The CRLF terminator is appended to B' after the transform and is never obfuscated, preserving line framing regardless of body content. The following table shows the obfuscated form of selected characters for implementor reference:

Cleartext	Hex	Obfuscated hex	Obfuscated char
H	0x48	0x00	NUL
O	0x4F	0x07	BEL
N	0x4E	0x06	ACK
K	0x4B	0x03	ETX
SP	0x20	0x68	h

Table 1: XOR Transform Reference (K = 0x48)

Note that the obfuscated form of H is the NUL byte (0x00). Implementations MUST NOT treat NUL bytes received in Privacy Mode as string terminators or error conditions.

6. Session Management

A session begins when a client establishes a TCP connection. The client MAY send multiple sequential Requests after any Privacy Mode negotiation. The server MUST process each Request and send the corresponding Response before processing the next. Pipelining is NOT RECOMMENDED.

The server SHOULD implement an idle connection timeout. A RECOMMENDED default is 30 seconds from the time the last Response was sent. Either party MAY close the connection at any time. The server SHOULD transmit any pending Response before initiating a close.

7. Implementation Considerations

7.1. Unicode Whitespace Handling

Implementations MUST use the Unicode White_Space property for word boundary detection and MUST NOT limit whitespace detection to ASCII characters only. Implementors SHOULD verify that the chosen library function covers the full Unicode White_Space property set.

7.2. Line Ending Handling

CRLF is the canonical line terminator. Server implementations MAY additionally accept a bare LF for compatibility with clients that do not emit CR. In Privacy Mode the bare LF MAY also be accepted, since CRLF bytes are never obfuscated.

7.3. NUL Bytes in Privacy Mode

As shown in Table 1, the obfuscated form of the letter H is the NUL byte 0x00. Because every HONK token begins with H, the obfuscated form of every Response token begins with NUL. Implementations MUST NOT treat NUL bytes in Privacy Mode content as string terminators. Implementations using null-terminated string APIs MUST account for this explicitly.

7.4. Concurrency

A HONK server MUST handle multiple simultaneous client connections. Implementations SHOULD use concurrent I/O mechanisms appropriate to their runtime environment.

8. Security Considerations

8.1. Privacy Mode Provides No Cryptographic Security

Privacy Mode MUST NOT be construed as providing cryptographic confidentiality, authentication, or integrity. The key K = 0x48 is fixed and publicly documented in this specification. Any observer with access to this document can de-obfuscate all Privacy Mode traffic immediately. Privacy Mode is trivially broken by a known-plaintext attack using a single observed Response. Operators requiring genuine confidentiality MUST use TLS [RFC8446] or an equivalent cryptographically secure transport.

8.2. Resource Exhaustion

Servers MUST implement a maximum request body length per Section 4.1 and a maximum honk count N per Section 4.4. Servers SHOULD limit concurrent connections and MUST implement idle timeouts per Section 6.

8.3. UTF-8 Input Validation

Servers MUST validate that the request body is well-formed UTF-8 after de-obfuscating in Privacy Mode. Servers receiving invalid UTF-8 MUST close the connection without sending a Response.

8.4. Network Exposure

The HONK protocol provides no authentication or authorization. Operators deploying a HONK server on a publicly accessible interface SHOULD restrict access using network-layer controls. Deployment on a public interface without access controls is NOT RECOMMENDED.

9. IANA Considerations

IANA is requested to assign the following entry in the Service Name and Transport Protocol Port Number Registry [RFC6335]:

Field	Value
Service Name	honk
Port Number	24565
Transport Protocol	TCP
Description	HONK: word-counting service with honk-based responses
Assignee	G00se (me@elaine.is)
Contact	G00se (me@elaine.is)
Reference	[This document]

Table 2: IANA Port Assignment Request

At the time of this writing, port 24565 does not appear in the IANA Service Name and Transport Protocol Port Number Registry as an assigned or reserved value. UDP port 24565 is not requested at this time. Per [RFC6335], IANA SHOULD mark UDP port 24565 as Reserved when assigning TCP port 24565.

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC0793] Postel, J., "Transmission Control Protocol", RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

10.2. Informative References

[RFC6335] Cotton, M., Eggert, L., Touch, J., Westerlund, M., and S. Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry", BCP 165, RFC 6335, DOI 10.17487/RFC6335, August 2011, <https://www.rfc-editor.org/info/rfc6335>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

Go Reference Implementation (Informative)

The following is the reference implementation of the HONK protocol, providing both server and client in a single Go source file. It requires Go 1.21 or later. The compiled binary is named "honk".

Usage:

```
honk server [-addr :24565]
honk send   [-addr host:24565] [-priv] [text...]
```

When invoked as "honk send", text may be supplied as command-line arguments or read line-by-line from standard input if no arguments are given.

<CODE BEGINS>

```
// honk - HONK protocol client and server (draft-g00se-honk-00)
package main

import (
    "bufio"
    "bytes"
    "flag"
    "fmt"
    "log"
    "net"
    "os"
    "strings"
    "unicode"
    "unicode/utf8"
)
```

```
const (
    DefaultAddr     = ":24565"
    DefaultHonks    = 3
    MaxRequestBytes = 65535
    MaxHonkCount    = 65535
    PrivLine        = "HONK PRIV"
    XORKey          = byte(0x48) // ASCII 'H'
)

func main() {
    if len(os.Args) < 2 {
        usage()
    }
    switch os.Args[1] {
    case "server":
        runServer(os.Args[2:])
    case "send":
        runClient(os.Args[2:])
    default:
        usage()
    }
}

func usage() {
    fmt.Fprintln(os.Stderr, "usage:")
    fmt.Fprintln(os.Stderr, "  honk server [-addr :24565]")
    fmt.Fprintln(os.Stderr, "  honk send   [-addr host:24565] [-priv] [text...]")
    os.Exit(1)
}

// ---- Server ----------------------------------------------------------

func runServer(args []string) {
    fs := flag.NewFlagSet("server", flag.ExitOnError)
    addr := fs.String("addr", DefaultAddr, "TCP listen address")
    fs.Parse(args)

    ln, err := net.Listen("tcp", *addr)
    if err != nil {
        log.Fatalf("honk server: listen %s: %v", *addr, err)
    }
    defer ln.Close()
    log.Printf("honk server: listening on %s (draft-g00se-honk-00)", *addr)

    for {
        conn, err := ln.Accept()
        if err != nil {
            log.Printf("honk server: accept: %v", err)
```

```
            continue
        }
        go serveConn(conn)
    }
}

func serveConn(conn net.Conn) {
    defer conn.Close()
    remote := conn.RemoteAddr().String()
    log.Printf("honk server: connection from %s", remote)

    buf := make([]byte, MaxRequestBytes+2)
    sc := bufio.NewScanner(conn)
    sc.Buffer(buf, len(buf))

    priv := false
    first := true

    for sc.Scan() {
        line := bytes.TrimRight(sc.Bytes(), "\r")

        // Privacy Mode negotiation: always cleartext on the first line.
        if first {
            first = false
            if string(line) == PrivLine {
                priv = true
                fmt.Fprintf(conn, "%s\r\n", PrivLine)
                log.Printf("honk server: %s Privacy Mode active", remote)
                continue
            }
        }

        content := line
        if priv {
            content = xor(line)
        }

        if !utf8.ValidString(string(content)) {
            log.Printf("honk server: %s invalid UTF-8; closing", remote)
            return
        }

        n := honkCount(string(content))
        resp := buildResponse(n)

        if priv {
            conn.Write(append(xor([]byte(resp)), '\r', '\n'))
        } else {
```

```
                fmt.Fprintf(conn, "%s\r\n", resp)
            }

            log.Printf("honk server: %s priv=%v words=%d honks=%d",
                remote, priv, wordCount(string(content)), n)
        }

        if err := sc.Err(); err != nil {
            log.Printf("honk server: %s: %v", remote, err)
        }
        log.Printf("honk server: %s closed", remote)
    }

    // ---- Client ------------------------------------------------------------

    func runClient(args []string) {
        fs := flag.NewFlagSet("send", flag.ExitOnError)
        addr := fs.String("addr", DefaultAddr, "server address")
        priv := fs.Bool("priv", false, "use Privacy Mode")
        fs.Parse(args)

        conn, err := net.Dial("tcp", *addr)
        if err != nil {
            log.Fatalf("honk client: connect %s: %v", *addr, err)
        }
        defer conn.Close()
        sc := bufio.NewScanner(conn)

        if *priv {
            fmt.Fprintf(conn, "%s\r\n", PrivLine)
            if !sc.Scan() {
                log.Fatalf("honk client: no priv ack")
            }
            if sc.Text() != PrivLine {
                log.Fatalf("honk client: unexpected priv ack: %q", sc.Text())
            }
            log.Printf("honk client: Privacy Mode active (K=0x48)")
        }

        send := func(text string) {
            if *priv {
                conn.Write(append(xor([]byte(text)), '\r', '\n'))
            } else {
                fmt.Fprintf(conn, "%s\r\n", text)
            }
            if !sc.Scan() {
                log.Fatalf("honk client: connection closed before response")
            }
```

```
            resp := sc.Text()
            if *priv {
                resp = string(xor([]byte(resp)))
            }
            fmt.Println(resp)
        }

        if rest := fs.Args(); len(rest) > 0 {
            send(strings.Join(rest, " "))
            return
        }
        stdin := bufio.NewScanner(os.Stdin)
        for stdin.Scan() {
            send(stdin.Text())
        }
    }

    // ---- Protocol helpers ----------------------------------------------------

    // xor applies single-byte XOR with K=0x48 to every byte of b.
    // The transform is its own inverse.
    func xor(b []byte) []byte {
        out := make([]byte, len(b))
        for i, v := range b {
            out[i] = v ^ XORKey
        }
        return out
    }

    // wordCount returns the number of Unicode-whitespace-delimited words.
    func wordCount(s string) int {
        return len(strings.FieldsFunc(s, unicode.IsSpace))
    }

    // honkCount computes N per Section 4.4:
    //   N = 2*W if W > 0; N = 3 if W == 0. Capped at MaxHonkCount.
    func honkCount(s string) int {
        w := wordCount(s)
        if w == 0 {
            return DefaultHonks
        }
        n := 2 * w
        if n > MaxHonkCount {
            return MaxHonkCount
        }
        return n
    }
```

```
   // buildResponse returns n SP-separated HONK tokens.
   func buildResponse(n int) string {
       tokens := make([]string, n)
       for i := range tokens {
           tokens[i] = "HONK"
       }
       return strings.Join(tokens, " ")
   }
   <CODE ENDS>

   To build:

   $ go build -o honk ./honk.go
   $ ./honk server
   2026/04/25 00:00:00 honk server: listening on :24565 (draft-g00se-honk-00)

   $ ./honk send "hello world"
   HONK HONK HONK HONK

   $ ./honk send -priv "hello world"
   HONK HONK HONK HONK
```

Author’s Address

G00se
Independent
Email: me@elaine.is