```
// Unbounded buffer, where underlying values are byte arrays

package bufb

// Linked list element
type BufEle struct {
        val []byte
        next *BufEle
}

type Buf struct {
        head *BufEle         // Oldest element
        tail *BufEle         // Most recently inserted element
}

func NewBuf() *Buf {
        return new(Buf)
}

func (bp *Buf) Insert(val []byte) {
        ele := &BufEle{val : val}
        if bp.head == nil {
                // Inserting into empty list
                bp.head = ele
                bp.tail = ele
        } else {
                bp.tail.next = ele
                bp.tail = ele
        }
}

func (bp *Buf) Front() []byte {
        if bp.head == nil { return nil }
        return bp.head.val
}

func (bp *Buf) Remove() []byte {
        e := bp.head
        if e == nil { return nil }
        bp.head = e.next
        // List becoming empty
        if e == bp.tail { bp.tail = nil }
        return e.val
}

func (bp *Buf) Empty() bool {
        return bp.head == nil
}

func (bp *Buf) Flush() {
        bp.head = nil
        bp.tail = nil
}
```

```
// Testing code for buffer

package bufb

import (
        "testing"
        "json"
        "rand"
        )

// Convert integer to byte array
func i2b(i int) []byte {
        b, _ := json.Marshal(i)
        return b
}

// Convert byte array back to integer
func b2i(b []byte) int {
        var i int
        json.Unmarshal(b, &i)
        return i
}

// How many repetitions
var ntest int = 10
// How many elements per test
var nele int = 50


func TestBuf(t *testing.T) {
        // Run same test ntest times
        for i := 0; i < ntest; i++ {
                bp := NewBuf()
                runtest(t, bp)
                if !bp.Empty() {
                        t.Logf("Expected empty buffer")
                        t.Fail()
                }
        }
}

func runtest(t *testing.T, bp *Buf) {
        inserted := 0
        removed := 0
        emptycount := 0
        for removed < nele {
                if bp.Empty() { emptycount ++ }
                // Choose action: insert or remove
                insert := !(inserted == nele)
                if inserted > removed && rand.Int31n(2) == 0 {
                                insert = false
                }
                if insert {
                        bp.Insert(i2b(inserted))
                        inserted ++
                } else {
                        v := b2i(bp.Remove())
                        if v != removed {
                                t.Logf("Removed %d.  Expected %d\n", v, removed)
                                t.Fail()
                        }
                        removed ++
                }
```

```
        }
}
```

```
// Unbounded buffer, where underlying values are byte arrays

package bufi

// Linked list element
type BufEle struct {
        val interface{}
        next *BufEle
}

type Buf struct {
        head *BufEle         // Oldest element
        tail *BufEle         // Most recently inserted element
}

func NewBuf() *Buf {
        return new(Buf)
}

func (bp *Buf) Insert(val interface{}) {
        ele := &BufEle{val : val}
        if bp.head == nil {
                // Inserting into empty list
                bp.head = ele
                bp.tail = ele
        } else {
                bp.tail.next = ele
                bp.tail = ele
        }
}

func (bp *Buf) Front() interface{} {
        if bp.head == nil { return nil }
        return bp.head.val
}

func (bp *Buf) Remove() interface{} {
        e := bp.head
        if e == nil { return nil }
        bp.head = e.next
        // List becoming empty
        if e == bp.tail { bp.tail = nil }
        return e.val
}

func (bp *Buf) Empty() bool {
        return bp.head == nil
}

func (bp *Buf) Flush() {
        bp.head = nil
        bp.tail = nil
}
```

```
// Testing code for buffer

package bufi

import (
        "testing"
        "json"
        "rand"
        "fmt"
        )

// How many repetitions
var ntest int = 10
// How many elements per test
var nele int = 50


// Convert integer to byte array
func i2b(i int) []byte {
        b, _ := json.Marshal(i)
        return b
}

// Convert byte array back to integer
func b2i(b []byte) int {
        var i int
        json.Unmarshal(b, &i)
        return i
}

func btest(t *testing.T, bp *Buf) {
        inserted := 0
        removed := 0
        emptycount := 0
        fmt.Printf("Byte array data: ")
        for removed < nele {
                if bp.Empty() { emptycount ++ }
                // Choose action: insert or remove
                insert := !(inserted == nele)
                if inserted > removed && rand.Int31n(2) == 0 {
                                insert = false
                }
                if insert {
                        bp.Insert(i2b(inserted))
                        inserted ++
                } else {
                        x := bp.Remove()  // Type = interface{}
                        b := x.([]byte)   // Type = []byte
                        v := b2i(b)
                        if v != removed {
                                t.Logf("Removed %d.  Expected %d\n", v, removed)
                                t.Fail()
                        }
                        removed ++
                }
        }
        fmt.Printf("Empty buffer %d/%d times\n", emptycount, nele)
}

func itest(t *testing.T, bp *Buf) {
        inserted := 0
        removed := 0
        emptycount := 0
```

```
        fmt.Printf("Integer data: ")
        for (removed < nele) {
                if bp.Empty() { emptycount ++ }
                // Choose action: insert or remove
                insert := !(inserted == nele)
                if inserted > removed && rand.Int31n(2) == 0 {
                        insert = false
                }
                if insert {
                        bp.Insert(inserted)
                        inserted ++
                } else {
                        x := bp.Remove()  // Type = interface{}
                        v := x.(int)      // Type = int
                        if v != removed {
                                t.Logf("Removed %d.  Expected %d\n", v, removed)
                                t.Fail()
                        }
                        removed ++
                }
        }
        fmt.Printf("Empty buffer %d/%d times\n", emptycount, nele)
}

func mtest(t *testing.T, bp *Buf) {
        inserted := 0
        removed := 0
        emptycount := 0
        fmt.Printf("Mixed data: ")
        for (removed < nele) {
                if bp.Empty() { emptycount ++ }
                // Choose action: insert or remove
                insert := !(inserted == nele)
                if inserted > removed && rand.Int31n(2) == 0 {
                        insert = false
                }
                if insert {
                        if rand.Int31n(2) == 0 {
                                // Insert as integer
                                bp.Insert(inserted)
                        } else {
                                // Insert as byte array
                                bp.Insert(i2b(inserted))
                        }
                        inserted ++
                } else {
                        x := bp.Remove()  // Type = interface{}
                        var iv int
                        switch v := x.(type) {
                        case int:
                                iv = v
                        case []byte:
                                iv = b2i(v)
                        default:
                                t.Logf("Invalid data\n")
                                t.Fail()
                        }
                        if iv != removed {
                                t.Logf("Removed %d.  Expected %d\n", iv, removed)
                                t.Fail()
                        }
                        removed ++
                }
```

```
        }
        fmt.Printf("Empty buffer %d/%d times\n", emptycount, nele)
}


type TestFun func(*testing.T, *Buf)

func testBuf(t *testing.T, f TestFun) {
        // Run same test ntest times
        for i := 0; i < ntest; i++ {
                bp := NewBuf()
                f(t, bp)
                if !bp.Empty() {
                        t.Logf("Expected empty buffer")
                        t.Fail()
                }
        }
}

func Testb(t *testing.T) {
        testBuf(t, btest)
}

func Testi(t *testing.T) {
        testBuf(t, itest)
}

func Testm(t *testing.T) {
        testBuf(t, mtest)
}
```

```
// Implementation of a UDP proxy

package main

import (
        "log"
        "os"
        "flag"
        "fmt"
        "net"
        "strings"
        "sync"
)

// Information maintained for each client/server connection
type Connection struct {
        ClientAddr *net.UDPAddr // Address of the client
        ServerConn *net.UDPConn // UDP connection to server
}

// Generate a new connection by opening a UDP connection to the server
func NewConnection(srvAddr, cliAddr *net.UDPAddr) *Connection {
        conn := new(Connection)
        conn.ClientAddr = cliAddr
        srvudp, err := net.DialUDP("udp", nil, srvAddr)
        if checkreport(1, err) { return nil }
        conn.ServerConn = srvudp
        return conn
}

// Global state
// Connection used by clients as the proxy server
var ProxyConn *net.UDPConn

// Address of server
var ServerAddr *net.UDPAddr

// Mapping from client addresses (as host:port) to connection
var ClientDict map [string] *Connection = make(map[string] *Connection)

// Mutex used to serialize access to the dictionary
var dmutex *sync.Mutex = new(sync.Mutex)

func setup(hostport string, port int) bool {
        // Set up Proxy
        saddr, err := net.ResolveUDPAddr("udp", fmt.Sprintf(":%d", port))
        if checkreport(1, err) { return false }
        pudp, err := net.ListenUDP("udp", saddr)
        if checkreport(1, err) { return false }
        ProxyConn = pudp
        Vlogf(2, "Proxy serving on port %d\n",port)

        // Get server address
        srvaddr, err := net.ResolveUDPAddr("udp", hostport)
        if checkreport(1, err) { return false }
        ServerAddr = srvaddr
        Vlogf(2, "Connected to server at %s\n", hostport)
        return true
}

func dlock() {
        dmutex.Lock()
}
```

```
func dunlock() {
        dmutex.Unlock()
}

// Go routine which manages connection from server to single client
func RunConnection(conn *Connection) {
        var buffer [1500]byte
        for {
                // Read from server
                n, err := conn.ServerConn.Read(buffer[0:])
                if checkreport(1, err) { continue }
                // Relay it to client
                _, err = ProxyConn.WriteToUDP(buffer[0:n], conn.ClientAddr)
                if checkreport(1, err) { continue }
                Vlogf(3, "Relayed '%s' from server to %s.\n",
                        string(buffer[0:n]), conn.ClientAddr.String())
        }
}

// Routine to handle inputs to Proxy port
func RunProxy() {
        var buffer[1500]byte
        for {
                n, cliaddr, err := ProxyConn.ReadFromUDP(buffer[0:])
                if checkreport(1, err) { continue }
                Vlogf(3, "Read '%s' from client %s\n",
                        string(buffer[0:n]), cliaddr.String())
                saddr := cliaddr.String()
                dlock()
                conn, found := ClientDict[saddr]
                if !found {
                        conn = NewConnection(ServerAddr, cliaddr)
                        if conn == nil {
                                dunlock()
                                continue
                        }
                        ClientDict[saddr] = conn
                        dunlock()
                        Vlogf(2, "Created new connection for client %s\n", saddr)
                        // Fire up routine to manage new connection
                        go RunConnection(conn)
                } else {
                        Vlogf(5, "Found connection for client %s\n", saddr)
                        dunlock()
                }
                // Relay to server
                _, err = conn.ServerConn.WriteToUDP(buffer[0:n], ServerAddr)
                if checkreport(1, err) { continue }
        }
}

var verbosity int = 6

// Log result if verbosity level high enough
func Vlogf(level int, format string, v ...interface{}) {
        if level <= verbosity {
                log.Printf(format, v...)
        }
}

// Handle errors
func checkreport(level int, err os.Error) bool {
```

```
        if err == nil {
                return false
        }
        Vlogf(level, "Error: %s", err.String())
        return true
}

func main() {
        var ihelp *bool = flag.Bool("h", false, "Show help information")
        var ipport *int = flag.Int("p", 6667, "Proxy port")
        var isport *int = flag.Int("P", 6666, "Server port")
        var ishost *string = flag.String("H", "localhost", "Server address")
        var iverb *int = flag.Int("v", 1, "Verbosity (0-6)")
//      var idrop *float64 = flag.Float64("d", 0.0, "Packet drop rate")
        flag.Parse()
        verbosity = *iverb
        if *ihelp {
                flag.Usage()
                os.Exit(0)
        }
        if flag.NArg() > 0 {
                ok := true
                fields := strings.Split(flag.Arg(0), ":")
                ok = ok && len(fields) == 2
                if ok {
                        *ishost = fields[0]
                        n, err := fmt.Sscanf(fields[1], "%d", isport)
                        ok = ok && n == 1 && err == nil
                }
                if !ok {
                        flag.Usage()
                        os.Exit(0)
                }
        }
        hostport := fmt.Sprintf("%s:%d", *ishost, *isport)
        Vlogf(3, "Proxy port = %d, Server address = %s\n",
                *ipport, hostport)
        if setup(hostport, *ipport) {
                RunProxy()
        }
        os.Exit(0)
}
```