Background

I have done a survey of various go packages and most of them have been written with specific use cases (tcp, unixdomain) and specific versions of 9P2000.*

Design goal #1: multiple versions

Currently **p9** only speaks **9P2000.L 9P2000**

Design goal #2: multiple transports

As mentioned before, **p9** is heavily optimized for linux, this theory is based on the number of linux specific build tags in the project :), which is great but I think we can come up with a design that would allow us to add support for even more languages and more transport mechanisms.

Design goal #3: access to messages for tools

p9 provides two very simple interfaces **File** and **Attacher** , which cover 90% of all use cases, but hides the messages from the consumers which makes it hard to implement things that aren't servers or clients, things like proxies, multiplexers and 9p debugging tools where the application doesn't really care about the contents of the message but only cares about how to parse the header and pass it on to the next thing.

message package

At a very high level the message package is quite simple.

```
// Message is the interface all 9P messages must implement
type Message interface {
        encoding.BinaryUnmarshaler
        encoding.BinaryMarshaler
        io.WriterTo
        io.ReaderFrom

        Type() MType
        Tag() Tag
        String() string
        Size() int64
}

type ResponseWriter interface {
        Write(m Message) error
}

type TransportHandlerFunc func(ctx context.Context, w ResponseWriter, req Message) error

type ProtoCaller interface {
        Proto(version string) interface{}
}

type TransportHandler interface {
        Serve9P(ctx context.Context, w ResponseWriter, req Message) error
}
```

ResponseWriter pattern is lifted right out of the HTTP package, which is what almost all go developers are familiar with, so it should have a very reasonable learning curve. **ResponseWriter** would also allow us to implement streams (we'll get to that in a bit) while still being backwards compatible with 9P transactions.

p9/net , **p9/ssh** and friends

The ability to have custom message types allows us to extend 9P functionality, for instance the 9P2000 implementation of the **Rread** message might look like this

```
type ReadResp struct {
        Header
        Count uint32
        Data  []byte
}

func wbit32(w io.Writer, x uint32) (int64, error) {
        n, err := w.Write([]byte{byte(x), byte(x >> 8), byte(x >> 16), byte(x >> 24)})
        return int64(n), err
}

// WriteTo implements io.WriterTo interface for ReadResp
func (r *ReadResp) WriteTo(w io.Writer) (wrote int64, err error) {
        var n int64
        n, err = r.Header.WriteTo(w)
        if err != nil {
                return n, err
        }

        // Data []byte
        // size of the byte slice
        n, err = wbit32(w, uint32(len(r.Data)))
        if err != nil {
                return n + wrote, err
        }
        wrote += n
        // bytes
        a, err := w.Write(r.Data)
        if err != nil {
                return int64(a) + wrote, err
        }
        wrote += int64(a)
        return
}
```

The **p9/net** package could extend that functionality and do some TCP specific optimizations.

```
type tcpmsg struct {
        payload message.Message
}

// WriteTo implements io.WriterTo interface for ReadResp
func (m *tcpmsg) WriteTo(w io.Writer) (wrote int64, err error) {
        conn, ok := w.(net.Conn)
        if !ok {
                return r.payload.WriteTo(w)
        }
        readResp, ok := m.payload.(*proto.ReadResp)
        if !ok {
                return r.ReadResp.WriteTo(w)
        }
        b, _ = readResp.Header.MarshalBinary()
        net.Buffers.Read(b)
        net.Buffers.Read(r.ReadResp.Data)
        a, err := net.Buffers.Write(w)
        if err != nil {
                return 0, err
        }
        return int64(a), nil
}
```

As previously mentioned the message package isn't concerned about protocol nuances. Just being able to parse the header information is fine for it's purposes.

So how do we handle multiple different protocols at the same time?

```
func Register(protocol string, HandlerFunc)
```

I am not a massive fan of package level globals but in this case, having this global registry makes sense, the agreement on what a version string means has to be universal. Ie, **9P2000.L** should be universally agreeed on by all implementors otherwise things will start getting messy.

multiple protocol versions

Let's talk over a concrete example. 9P2000 message package about a specific protocol we import the package.

```
import (
        _ "harvey-os.org/9p/9P2000"
)
```

When imported the package uses the init func to register it self as the handler for the specific version of the protocol.

```
package proto

const Version = "9P2000"

func init() {
        message.Register(Version, ProtoHandler)
}

func ProtoHandler(ctx context.Context, w message.ResponseWriter, req message.Message,handler
        p92000handler, ok := handler.(Proto)
        if !ok {
                return fmt.Errorf("%s is not supported", Version)
        }
        switch m.(type) {
        case r *AttachReq:
                resp, err =p92000handler.Version(ctx, r)
                // -----snip---------->B
        }
}

type Proto interface {
        Attach(ctx context.Context, r *AttachReq) (*AttachResp, error)
        // -----snip---------->B
}
```

9P2000.??R

Inorder to extend the protocol, we'd create a new packages as necessary, for instance harvey-os.org/9p/9P2000.Harvey

```
package proto

import (
        base "harvey-os.org/9p/9P2000"
)

const Version = "9P2000.???"

func init() {
        message.Register(Version, ProtoHandler)
}

func ProtoHandler(ctx context.Context, w message.ResponseWriter, req message.Message,handler
        p92000handler, ok := handler.(Proto)
        if !ok {
                return base.ProtoHandler(ctx, w, req, handler)
        }
        switch m.(type) {
        case r *GlobReq:
                resp, err =p92000handler.Glob(ctx, r)
        default:
                return base.ProtoHandler(ctx, w, req, handler)
        }
}

type Proto interface {
        base.Proto
        Glob(ctx context.Context, r *GlobReq) (*GlobResp, error)
}

type GlobReq struct {
        message.Header
}
type GlobResp struct {
        message.Header
}
```

We can create as many different versions of the protocol as we want without breaking other protocols. For instance if Graham wants to create his version it would be easy to fork the repo change the version number to **9P2000.???Graham** , teach file to handle messages of a certain type. And that's it really

Putting it all together

The end users should still use File to implement their 9P servers.

File should implement the which ever versions of the protocol it wants to implement. It should then return a specific implementation for that version of the protocol.

```
import (
        base "harvey-os.org/9p/9P2000"
        harvey "harvey-os.org/9p/9P2000.Harvey"
        linux "harvey-os.org/9p/9P2000.L"
)

type File struct{}

func (f *File) Stat(r *base.StatReq) (*base.StatResp, error) {}

type LinuxFile struct {
        File
}

type Harvey struct {
        File
}

// weird Rlopen something
func (f *LinuxFile) Open(r linux.OpenReq) (*linux.OpenResp, error) {}

// regular Ropen
func (f *HarveyFile) Open(r harvey.OpenReq) (*harvey.OpenResp, error) {}

func (f *file) Proto(version string) interface {
        switch version {
                case linux.Version:
                return &LinuxFile{f}
                case harvey.Version:
                return &HarveyFile{f}
        }
}
```

This makes sure that File can support as many different versions of protocol even if they reuse message numbers.

Aside: Streams

Johnmeasured the effects of netwowrk lag in his thesis "FTP-like streams for the 9P file protocol" (2010). He went on to propose and implemented a streaming mechanism for 9P with the adition of two message types Tstream and **Rstream**

```
size[4] Tstream tag[2] fid[4] isread[1] offset[8]
size[4] Rstream tag[2] count[4] data[count]
```

As mentioned previously the **ResponseWriter** interface is to support streaming responses. **ResponseWriter** when combined with custom message types can be quite powerful.

For instance here is a custom fsnotify server might be implemented.

```
type fsnotifyStreamMessage struct {
        event fsnotify.Event
}

func (r *inotifyStreamMessage) WriteTo(w io.Writer) (n int64, err error) {
        b := make([]byte, 4)
        binary.LittleEndian.PutUint16(b[0:], r.Op)
        binary.LittleEndian.PutUint16(b[0:], uint16(len(r.Op.Name)))
        w.Write(b)
        io.WriteString(w, r.Op.Name)
}

func fsnotify(ctx context.Context, w ResponseWriter, r Message) error {
        _, ok := r.(*StreamRequest)
        if ok {
                return errors.New("...")
        }
        watcher, err := fsnotify.NewWatcher()
        if err != nil {
                log.Fatal(err)
        }
        defer watcher.Close()
        defer w.Close()
        for {
                select {
                case event, ok := <-watcher.Events:
                        if !ok {
                                return
                        }
                        if err := w.Write(&inotifyStreamMessage{event: event}); err != nil {
                                return err
                        }
                case err, ok := <-watcher.Errors:
                        return err
                }
        }
}
```

This pattern can be used to create 9p live video servers, device drivers and so on.

access to messages

opentracing example.

```
import "github.com/opentracing/opentracing-go"

type Tracer struct {
        h message.TransportHandler
        opentracing.Tracer
}

type responseWriter struct {
        message.ResponseWriter
        opentracing.Tracer
}

func (t *Tracer) Serve9P(ctx context.Context, w message.ResponseWriter, req message.Message)
        t.StartSpan(m.Type(), m.Tag())
        return t.h.Handle(ctx, &responseWriter{w,t.Tracer}, req)
}
```

FAQ

Isn’t this too much work?

probably

Is this a smart thing to do?

I’m not smart enough to answer that.

You didn’t cover [x]

This isn’t a complete design, I’m sure there are inconsistencies and some parts of this sound half baked. Because they are, I just wanted to write down my thoughts somewhere before doing a major refactor. I am hoping ideas on here can serve as a starting point to discuss some changes to p9. That’s it! I’m sure by the the we decide if this is worth pursuing, names of packages and interfaces will change, so do not spend too much time thinking about those.