The basics of file systems

Improvements to Andrei's ReliableWriter

```
func (rw *ReliableWriter) launchWriting(ctx context.Context) {
   go func() {
      defer close(rw.resultChan)
      for {
         select {
         case <-rw.writeEventsChan:
      …
}

func (rw *ReliableWriter) Complete(ctx context.Context) error {
   …
   rw.notifyWriteEvent()
   return <-rw.resultChan:
}

func (rw *ReliableWriter) Abort(ctx context.Context) {
   rw.unreliableWriter.Abort(ctx)
   rw.isComplete = false
   rw.isAborted = true
   rw.notifyWriteEvent() // To resume launch
}
```

Improvements to Andrei's ReliableWriter

```
func (rw *ReliableWriter) launchWriting(ctx context.Context) {
   go func() {
      defer close(rw.resultChan)
      for {
         select {
         case <-rw.writeEventsChan:
      …
}

func (rw *ReliableWriter) Complete(ctx context.Context) error {
   …
   rw.notifyWriteEvent()
   return <-rw.resultChan:
}

func (rw *ReliableWriter) Abort(ctx context.Context) {
   rw.unreliableWriter.Abort(ctx)
   rw.isComplete = false
   rw.isAborted = true
   rw.notifyWriteEvent() // To resume launch
}
```

1. Abort() does not wait for the chunk writer goroutine to exit.
2. The lifecycle of the chunk writer goroutine is overly complicated.
3. There is useless locking in unreliable writers. It needs to be removed.

Improvements to Andrei's ReliableWriter

```
var buf ScatterGatherBuffer
if canBeLast {
   buf, err = rw.data.TakeBytes(0, rw.MaxChunkSz)
} else {
   buf, err = rw.data.TakeBytes(rw.MinChunkSz, rw.MaxChunkSz)
}

...

written, err := rw.attemptWriteWithRetries(ctx,
   buf.ToBytes(),
   chunkBegin, chunkEnd,
   isLast)
```

Improvements to Andrei's ReliableWriter

```
var buf ScatterGatherBuffer
if canBeLast {
   buf, err = rw.data.TakeBytes(0, rw.MaxChunkSz)
} else {
   buf, err = rw.data.TakeBytes(rw.MinChunkSz, rw.MaxChunkSz)
}

...

written, err := rw.attemptWriteWithRetries(ctx,
   buf.ToBytes(),
   chunkBegin, chunkEnd,
   isLast)
```

This defeats the whole purpose of scatter-gather lists.
The chunk writer goroutine is not zero-copy.

Two venues for further development

1. Add fault injections to unreliable writers to test whether ReliableWriter correctly recovers from retryable errors.
2. Implement a custom protocol and a proxy that converts its requests to GCS requests.

Fault injections

The current implementation of ReliableWriter is broken because it does not even retry many of possible errors.

How to proceed:

1. Read about `net.Listener`, `net.Dialer` and `net.Conn`.
2. Make an implementation of `net.Conn` that injects errors at random.
3. Make an implementation of `net.Dialer` that wraps all outgoing connections into a fault-injecting `net.Conn`.
4. Read about `http.Transport` and learn how to configure HTTP clients with custom dialers.
5. Learn how to create a GCS client with a custom HTTP transport.
6. Create a GCS client with your fault-injecting `http.Transport`.
7. Run test uploads with this client, find bugs in `ReliableWriter`, and fix them.

Custom protocol

We will need a custom protocol to run uploads over multiple connections.

However, it makes sense to start with a very trivial protocol that uses only 1. Requests in this protocol should map 1-to-1 to GCS requests.

How to proceed:

1. Learn how to use `net.Listener` and `net.Dial` to make a server socket and connect to it.
2. Read about `encoding/binary` and learn to serialise and deserialise structs that represent request headers and arguments.
3. Make structs with arguments for all GCS requests, and learn to send and receive those structs.
4. Learn to convert request arguments from your custom format to GCS requests and issue GCS requests.
5. Make sure all of your requests have a common header that contains the sequence number, and the request size.
   Quiz: why are these important?
6. Make sure that you can proxy `WriteAt()` without receing the whole of its body first.