The basics of file systems

Neapolis University Pafos

JET BRAINS

Implementing a connection pool

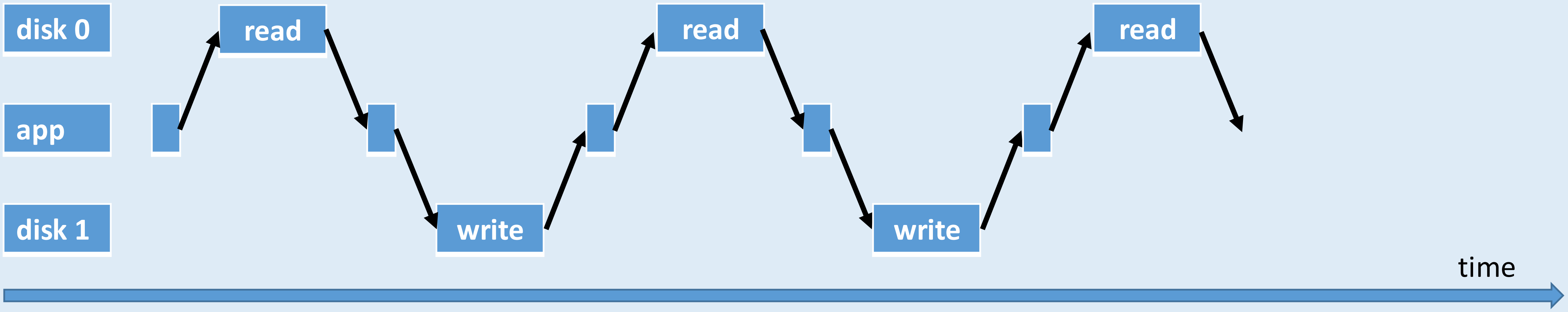
```
func (upw *UnreliableProxyWriter) WriteAt(…) (int64, error) {
    …
    header := RequestHeader{
        SequenceNumber: upw.sequenceNumber,
        RequestType:    MessageTypeUploadPart,
    }
    writeAtReq := WriteAtRequestHeader{…}
    …
    if _, err := conn.Write(buf.Bytes()); err != nil {
    …
    if err := upw.receiveResponse(); err != nil {
    …
    upw.currentOffset = chunkEnd
    return n, nil
}
```

Synchronous and asynchronous IO, pipelining and multiplexing

Consider a naïve implementation of a routine that copies a file from one disk to another:

```
while (!done) {
  r = read(fd_in, buf, sizeof(buf));
  r0 = write(fd_out, buf, r);
  ...
}
```

Let us draw time intervals when each disk is accessed:



Implementing a connection pool

```
func (upw *UnreliableProxyWriter) WriteAt(...) (int64, error) {
    ...
    header := RequestHeader{
        SequenceNumber: upw.sequenceNumber,
        RequestType:    MessageTypeUploadPart,
    }
    writeAtReq := WriteAtRequestHeader{...}
    ...
    if _, err := conn.Write(buf.Bytes()); err != nil {
    ...
    if err := upw.receiveResponse(); err != nil {
    ...
    upw.currentOffset = chunkEnd
    return n, nil
}
```

Connections must be kept busy at all times.

Implementing a connection pool

```
func (upw *UnreliableProxyWriter) WriteAt(…) (int64, error) {
    …
    header := RequestHeader{
        SequenceNumber: upw.sequenceNumber,
        RequestType:    MessageTypeUploadPart,
    }
    writeAtReq := WriteAtRequestHeader{…}
    …
    if _, err := conn.Write(buf.Bytes()); err != nil {
    …
    if err := upw.receiveResponse(); err != nil {
    …
    upw.currentOffset = chunkEnd
    return n, nil
}
```

Connections must be kept busy at all times.

One more reason why this is a must:

- The Linux kernel tracks the average speed of writes to a connection, and will not grow socket buffers if there are not enough writes on average. Connections with few writes on average will stay slow.

Implementing a connection pool

```
func (upw *UnreliableProxyWriter) WriteAt(…) (int64, error) {
    …
    header := RequestHeader{
        SequenceNumber: upw.sequenceNumber,
        RequestType:    MessageTypeUploadPart,
    }
    writeAtReq := WriteAtRequestHeader{…}
    …
    if _, err := conn.Write(buf.Bytes()); err != nil {
    …
    if err := upw.receiveResponse(); err != nil {
    …
    upw.currentOffset = chunkEnd
    return n, nil
}
```

Connections must be kept busy at all times.

One more reason why this is a must:

- The Linux kernel tracks the average speed of writes to a connection, and will not grow socket buffers if there are not enough writes on average. Connections with few writes on average will stay slow.

One may try to solve is this way:

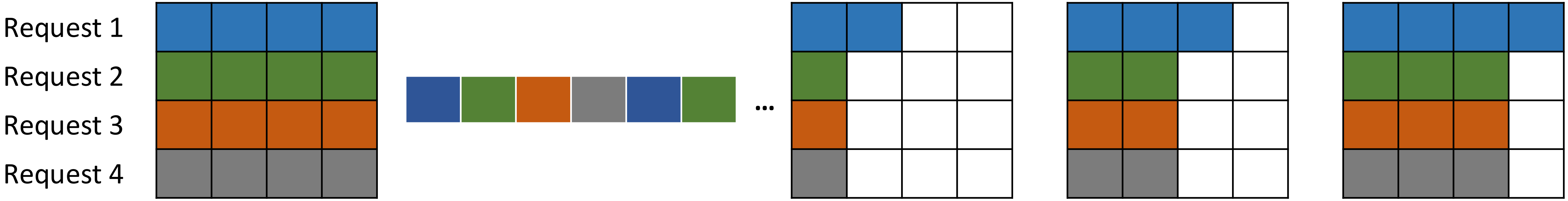
- suppose a chunk has 32M worth of data,
- let us issue 32 `WriteAt()`s, each with 1M payload,
- let us have 32 connections to send all `WriteAt()`s in parallel.

How the network interacts with concurrent requests

A client that send multiple requests over multiple network connections

The network

The server's request queue



From the point of view of the network different connections are independent streams of bytes and each must get an equal share of the bandwidth.

The server needs a lot of RAM to buffer all incoming requests. If it does not, it will throttle the client.

Implementing a connection pool

```
func (upw *UnreliableProxyWriter) WriteAt(…) (int64, error) {
    …
    header := RequestHeader{
        SequenceNumber: upw.sequenceNumber,
        RequestType:    MessageTypeUploadPart,
    }
    writeAtReq := WriteAtRequestHeader{…}
    …
    if _, err := conn.Write(buf.Bytes()); err != nil {
    …
    if err := upw.receiveResponse(); err != nil {
    …
    upw.currentOffset = chunkEnd
    return n, nil
}
```

Connections must be kept busy at all times.

Implementing a connection pool

```
func (upw *UnreliableProxyWriter) WriteAt(…) (int64, error) {
   …
   header := RequestHeader{
      SequenceNumber: upw.sequenceNumber,
      RequestType:    MessageTypeUploadPart,
   }
   writeAtReq := WriteAtRequestHeader{…}
   …
   if _, err := conn.Write(buf.Bytes()); err != nil {
   …
   if err := upw.receiveResponse(); err != nil {
   …
   upw.currentOffset = chunkEnd
   return n, nil
}
```

Connections must be kept busy at all times.

In go one can achieve this by assigning 3 goroutines to each connection:

1. a goroutine that writes requests to the connection,
2. a goroutine that receives requests from the connection,
3. a control goroutine that closes the connection and cancels the context of the reader and the writer.

Implementing a connection pool

When we have a pool of connections we need to decide how to assign requests to them.

Implementing a connection pool

When we have a pool of connections we need to decide how to assign requests to them.

This pairs nicely with writer goroutines.

```
type ConnGroup struct {
   …
   messages chan *message
}

func (c *Conn) writerGor(ctx context.Context) {
   for req := range c.connGroup.messages {
      … send the request …
   }
}

func (cg *ConnGroup) SendMessage(ctx context.Context, msg
*message) (err error) {
   select {
   case cg.messages <- message:
      return nil
   case <-ctx.Done():
      return ctx.Err()
   }
}
```

A connection sends a message as soon as it becomes available.

The channel throttles writers. If all connections are busy then `SendMessage()` is put to sleep until there is a connection that can send more requests.

Implementing a connection pool

When we have a pool of connections we need to decide how to assign requests to them.

This pairs nicely with writer goroutines.

```
type ConnGroup struct {
   …
   messages chan *message
}

func (c *Conn) writerGor(ctx context.Context) {
   for req := range c.connGroup.messages {
      … send the request …
   }
}

func (cg *ConnGroup) SendMessage(ctx context.Context, msg
*message) (err error) {
   select {
   case cg.messages <- message:
      return nil
   case <-ctx.Done():
      return ctx.Err()
   }
}
```

A connection sends a message as soon as it becomes available.

The channel throttles writers. If all connections are busy then `SendMessage()` is put to sleep until there is a connection that can send more requests.

Note: `SendMessage()` does not work like HTTP client's Do`()`. It only sends a message. It does not wait for the response.

Implementing a connection pool

When we have a pool of connections we need to decide how to assign requests to them.

This pairs nicely with writer goroutines.

```
type ConnGroup struct {
   …
   messages chan *message
}

func (c *Conn) writerGor(ctx context.Context) {
   for req := range c.connGroup.messages {
      … send the request …
   }
}

func (cg *ConnGroup) SendMessage(ctx context.Context, msg
*message) (err error) {
   select {
   case cg.messages <- message:
      return nil
   case <-ctx.Done():
      return ctx.Err()
   }
}
```

A connection sends a message as soon as it becomes available.

The channel throttles writers. If all connections are busy then `SendMessage()` is put to sleep until there is a connection that can send more requests.

Note: `SendMessage()` does not work like HTTP client's `Do()`. It only sends a message. It does not wait for the response.

Question: how do we implement responses?

Implementing a connection pool

When we have a pool of connections we need to decide how to assign requests to them.

This pairs nicely with writer goroutines.

```
type ConnGroup struct {
   …
   messages chan *message
}

func (c *Conn) writerGor(ctx context.Context) {
   for req := range c.connGroup.messages {
      … send the request …
   }
}

func (cg *ConnGroup) SendMessage(ctx context.Context, msg
*message) (err error) {
   select {
   case cg.messages <- message:
      return nil
   case <-ctx.Done():
      return ctx.Err()
   }
}
```

A connection sends a message as soon as it becomes available.

The channel throttles writers. If all connections are busy then `SendMessage()` is put to sleep until there is a connection that can send more requests.

Note: `SendMessage()` does not work like HTTP client's `Do()`. It only sends a message. It does not wait for the response.

Question: how do we implement responses?
Idea: a response is a message "this is the result of request #N".

Note: this way responses may use any connection in the pool. Compare this to IP packets in a TCP connection using any available path between endpoints.

Implementing a connection pool

```
func (upw *UnreliableProxyWriter) WriteAt(…) (int64, error) {
    …
    header := RequestHeader{
        SequenceNumber: upw.sequenceNumber,
        RequestType:    MessageTypeUploadPart,
    }
    writeAtReq := WriteAtRequestHeader{…}
    …
    if _, err := conn.Write(buf.Bytes()); err != nil {
    …
    if err := upw.receiveResponse(); err != nil {
    …
    upw.currentOffset = chunkEnd
    return n, nil
}
```

Implementing a connection pool

```
func (upw *UnreliableProxyWriter) WriteAt(…) (int64, error) {
   …
   header := RequestHeader{
      SequenceNumber: upw.sequenceNumber,
      RequestType:    MessageTypeUploadPart,
   }
   writeAtReq := WriteAtRequestHeader{…}
   …
   if _, err := conn.Write(buf.Bytes()); err != nil {
   …
   if err := upw.receiveResponse(); err != nil {
   …
   upw.currentOffset = chunkEnd
   return n, nil
}
```

Do we need to wait for replies to all `WriteAt()`s?

Implementing a connection pool

```
func (upw *UnreliableProxyWriter) WriteAt(...) (int64, error) {
  ...
  header := RequestHeader{
    SequenceNumber: upw.sequenceNumber,
    RequestType:    MessageTypeUploadPart,
  }
  writeAtReq := WriteAtRequestHeader{...}
  ...
  if _, err := conn.Write(buf.Bytes()); err != nil {
  ...
  if err := upw.receiveResponse(); err != nil {
  ...
  upw.currentOffset = chunkEnd
  return n, nil
}
```

Do we need to wait for replies to all `WriteAt()`s?

This is not necessary. Only the last `WriteAt()` to a chunk returns the result of a call to GCS. It decides whether a chunk upload was successful or not. Replies from other requests may be used to fail fast if there was a network error.

Implementing a connection pool

```
func (upw *UnreliableProxyWriter) WriteAt(…) (int64, error) {
    …
    header := RequestHeader{
        SequenceNumber: upw.sequenceNumber,
        RequestType:    MessageTypeUploadPart,
    }
    writeAtReq := WriteAtRequestHeader{…}
    …
    if _, err := conn.Write(buf.Bytes()); err != nil {
    …
    if err := upw.receiveResponse(); err != nil {
    …
    upw.currentOffset = chunkEnd
    return n, nil
}
```

Do we need to wait for replies to all `WriteAt()`s?

This is not necessary. Only the last `WriteAt()` to a chunk returns the result of a call to GCS. It decides whether a chunk upload was successful or not. Replies from other requests may be used to fail fast if there was a network error.

In general, waiting for a reply is a point where a client and a server synchronise. Synchronisation limits the scalability. One must try to have as few synchronisation points as possible.

Implementing a connection pool

```
func (upw *UnreliableProxyWriter) WriteAt(…) (int64, error) {
   …
   header := RequestHeader{
      SequenceNumber: upw.sequenceNumber,
      RequestType:    MessageTypeUploadPart,
   }
   writeAtReq := WriteAtRequestHeader{…}
   …
   if _, err := conn.Write(buf.Bytes()); err != nil {
   …
   if err := upw.receiveResponse(); err != nil {
   …
   upw.currentOffset = chunkEnd
   return n, nil
}
```

Do we need to wait for replies to all `WriteAt()`s?

This is not necessary. Only the last `WriteAt()` to a chunk returns the result of a call to GCS. It decides whether a chunk upload was successful or not. Replies from other requests may be used to fail fast if there was a network error.

In general, waiting for a reply is a point where a client and a server synchronise. Synchronisation limits the scalability. One must try to have as few synchronisation points as possible.

This introduces a problem, though: a write #K may fail, but the client sends writes # K+1, K+2 and so on without waiting for a reply to write #K. The server must be able to handle writes to an already failed upload.