Batch processing in RESTful web services

MASTER'S THESIS

Wilhelm Svenselius <wsv@kth.se>
2015-03-18
# Table of Contents

1. INTRODUCTION ........................................................................................................... 1
   - Problem Statement .................................................................................................. 1
   - Purpose and Delimitations ..................................................................................... 2
   - Method and Conclusions ....................................................................................... 3

2. BACKGROUND ............................................................................................................ 4
   - Batch Processing in General Computing .............................................................. 4
   - Network Latency — An Example ............................................................................. 4
   - Batching and REST ............................................................................................... 6

3. METHODOLOGY ........................................................................................................... 8
   - Not a Field Study ................................................................................................... 8
   - Measuring Performance ......................................................................................... 9
   - Implementation .................................................................................................... 9

4. APPROACHES TO BATCHING ................................................................................... 11
   - 4.1. HTTP Protocol Features .................................................................................. 11
       - Connection keep-alive ........................................................................................ 11
       - Pipelining ........................................................................................................... 12
       - Practical issues with pipelining ......................................................................... 12
   - 4.2. Composing Multiple Resources into a Single Response .................................. 13
       - Composition with complex operations ............................................................... 14
       - Composite representation using multipart/mixed ............................................. 15
       - Complex request/response scenarios ................................................................. 16
       - Headers of the top-level document ................................................................... 18
       - 207 Multi-Status .................................................................................................. 19
   - 4.3 An Alternative Approach: Batching Via Proxy ................................................. 20

5. IMPLEMENTATION ....................................................................................................... 22
   - 5.1. Clients .............................................................................................................. 22
   - 5.2. Server ................................................................................................................ 25

6. PERFORMANCE AND EXPERIMENTAL RESULTS .................................................... 28
   - 6.1. HTTP Pipelining ............................................................................................... 28
   - 6.2. Batching using multipart/mixed ....................................................................... 31

7. CONCLUSION ............................................................................................................... 35

8. REFERENCES AND ACKNOWLEDGEMENTS ............................................................. 36
   - REFERENCES ......................................................................................................... 36
Abstract

Network latency is one of the most significant causes of poor performance in web services, as it is not uncommon for latency to take up a greater share of the total time from request to response than the actual processing time on the server. Working within the well-established REST architectural style for web services, we examine HTTP pipelining and composite representation using multipart/mixed as potential means of reducing the effects of latency on batched operations, using experimental implementations of both approaches to test their performance in different scenarios. The results indicate that performance improvements of up to 50% are possible using pipelining, and up to 80% when using composite representation, under controlled conditions. This supports the conclusion that significant improvements in performance are achievable in existing RESTful web services given a reasonable development effort.
1. Introduction

The field of Information Technology concerns itself greatly with the development and standardization of network protocols, providing some balance between the mostly opposite goals of general applicability, performance, reliability and ease of implementation. As the World Wide Web has evolved to encompass a huge variety of content and services, HTTP (Hyper-Text Transfer Protocol) over TCP/IP has become the foundation for application-level communications (1). Furthermore, the REST (Representative State Transfer) architecture, which is implemented using HTTP in the vast majority of cases (2), is widely considered a best practice for designing web services.

HTTP was not built for speed, however, and can easily become a bottleneck as the number of operations per second grows. This is an issue for websites, which are intended for use by humans, but especially so for web services, that are primarily accessed by other computers. Of particular concern are batch operations, characterized by a large number of operations to be carried out in sequence or parallel. Such operations are common in use cases pertaining to data import/export.

Problem statement

The problem with the HTTP protocol in batch operations is essentially one of inefficient resource use. There simply is no standard method – de facto or otherwise – for performing operations in bulk. Although late additions to HTTP show that someone noticed this, the existing facilities are poorly supported and rarely used in practice. We are left with performing operations one at a time. This inefficiency exhibits itself in various negative ways: operations will take longer to complete and use more computing and network resources. A web service is often if not always part of a much larger software system, so the bottleneck ends up reducing the overall system performance – wasting time, electricity and money that could have been better spent elsewhere.

The REST architectural style does nothing to alleviate this; as we will find out, it can actually make it worse. Yet its other properties make it a good choice for web services – and sites – despite this oversight, so we would like to keep it. Furthermore, the openness and usefulness of the modern web depends on a broad agreement with regards to standards and protocols; we must not break any established practices; e.g. by introducing new protocols and expecting everyone else to work to support them. Instead,
we would like for existing infrastructure, HTTP clients and servers to support our solution to the fullest extent possible without modification.

Once we have found such a solution (or several), the approximate implementation effort must be weighed against the performance gains. In the end, developer hours are more costly to most businesses than a few seconds’ difference on operations that take several minutes to begin with. In reality, this trade-off is likely to be both subjective and scenario-dependent, but we can at least gain an understanding of how much of a speed-up is realistically achievable through different techniques.

**How can we design a RESTful web service to make batch operations significantly more efficient?**

To be relevant and practically applicable in the real world, an answer to this question should not require a significant redesign of standard network protocols or the underlying network architecture, and the performance gains should be significant.

**Purpose and delimitations**

In this paper, I will present, compare and contrast several approaches to how REST-style web services using HTTP can be adapted to handle batches of operations in a much more performant manner. In addition to theoretical aspects, I will cover implementation details – potentially making this paper useful as a “cookbook” for system developers and architects struggling with similar issues.

I have chosen to limit the scope of this paper to approaches that can be practically undertaken on existing web services or those being developed at the time of writing. This naturally excludes any approaches, no matter how potentially effective, that depend on the deployment of new protocols to replace HTTP/1.1 (including SPDY, the current proposed basis for HTTP/2.0 (3)), significant changes to Internet infrastructure or new scientific breakthroughs (e.g. quantum teleportation).

The methods selected for evaluation in this paper are HTTP pipelining, which allows multiple requests and responses to be in transit concurrently, and multipart/mixed, a MIME format that allows composition of multiple requests or responses into one.
Method and conclusions

To compare and evaluate the selected methods, I implemented a pair of clients and a server application, which could be configured to perform and time a set number of requests and responses, each with a defined payload size. Because of the simple nature of the server application, processing time for each request was essentially constant. This allowed for reproducible results and relatively precise measurements.

Running the experiments to determine the relative performance of each method, I found that while requiring a significantly larger development effort, the multipart/mixed method of compositing multiple requests or responses into larger documents yielded very impressive results, because the effects of latency on large numbers of requests were essentially eliminated. On the other hand, significant improvements were made using HTTP pipelining as well, which has a much lower complexity of implementation – being supported, though not enabled by default – by many HTTP server and client implementations in use today (4).
2. Background

To understand where the performance issues lies in HTTP, how services based on the REST architecture (we will simply refer to them as RESTful from now on) inherit these issues, and how batch operations are used and why they are important, it’s first necessary to go into some background about how these things are defined and how they depend on one another. We will not spend too much time going over the particulars of HTTP or REST, as they are covered quite extensively by prior works.

Batch processing in general computing

Batch processing, or batching for short, refers to the practice of combining several logical operations into one, which is then assigned to an execution unit to be run all at once, requiring no intervention from the initiator until all the operations have been completed (5). Similar to a human focusing her mental efforts on quickly finishing a large pile of work, this is a straightforward way to achieve significant performance improvements in essentially any software system: repetetive, similar actions make excellent use of instruction and data caches, as well as JIT (Just-in-time) optimization strategies, improving performance and reducing resource use.

Many computing tasks have significant set-up and teardown penalties, because they depend on the retrieval of resources that are external to the execution unit (e.g. data stored in a database somewhere far away) or because of long and complex routing paths within the software itself (e.g. authentication layers). A system designed to accept batch operations will minimize such penalties by ensuring that they are incurred only once for the entire batch, instead of once per individual operation.

A subset of batches are transactions, which have the property of atomic execution – the state of the system prior to execution of the batch is preserved so that it can be rolled back if any of the individual operations should fail (6). In addition, other users of the system may be able to access the preserved state as the batch is running, effectively making it appear as though the batch is executed instantly at the moment of completion, even though the true runtime may have been very long. Transactions can thus serve to greatly benefit the reliability and availability of a software system.

Network latency – an example

As previously mentioned, set-up and teardown penalties can add significant resource cost to even otherwise simple computing tasks. One example of such a penalty that specifically concerns HTTP is network latency. Being a
request-response protocol, each HTTP operation requires at least one network round-trip, incurring network latency both ways. Even on local area networks, this latency can be orders of magnitude greater than the time required to perform the operation itself. The recent advent of “cloud” services, where the physical location of a server is largely abstracted away, means the risk of high network latencies impacting service performance is greater than before:

One must also not ignore the impact of latency when determining the maximum user throughput. For applications, such as email and Internet surfing, which require some form of TCP/IP packet acknowledgements, the theoretical maximum data rate can be limited by the latency of the end-to-end connection. (7)

Knowing this, we understand that a key benefit of batching is reducing the impact of network latency. Most HTTP clients (e.g. web browsers) will wait for the response from one request before sending another, or only allow a low number (typically 4-8) of “open” requests at the same time. The HTTP 1.1 specification previously suggested (8) a limit of two simultaneous requests per hostname, but this was later revised (9) to not give a specific figure.

Let’s look at a simple example. Consider a client and a server that are separated by a connection with 10 ms of latency and a capacity of 10 Mb/s (a slow LAN, by today’s standards). The client wants to upload a total of 10,000 files, each 8,000 bits in size, to the server. We’ll ignore protocol overhead and latencies stemming from other components than the network for this example. If we define $t_{\text{request}}$ as the time required for sending one complete request and $t_{\text{total}}$ the time required to send all 10,000 requests, we get:

$$t_{\text{request}} = \text{latency} + \frac{\text{bits per request}}{\text{bits per ms}} = 10 + \frac{8000}{10000} = 10.8 \text{ ms}$$

$$t_{\text{total}} = 10000 \cdot t_{\text{request}} = 108 \text{ s}$$

We can see that roughly 90% of the time spent on each request is network latency, even though 10 ms is actually a rather low latency in Internet terms. All 10,000 requests would take a total of 108 seconds to transmit. Consider now the case where all requests are sent in bulk:
\[ t_{\text{total}} = \text{latency} + \frac{10000 \cdot \text{bits per request}}{\text{bits per ms}} = 10 + \frac{10000 \cdot 8000}{10000} = 8010 \text{ ms} \approx 8 \text{ s} \]

The batched version needs only about 7% of the time. The case for the server’s response is analogous. If we consider a network with roughly a quarter to half the capacity but twenty to fifty times worse latency, such as a cellular 3G network (capacity typically 2-6 Mb/s, latency 200-500 ms) we quickly realize that the performance benefits from batching can be very significant.

Developers of applications that routinely deal with tasks like the one our example describes frequently come up with clever workarounds to fix the performance issues. For example, some software packages for publishing photos or other documents on the Web will allow the user to upload or download files in bulk by using an archive file format, such as ZIP or RAR. As most modern file formats already employ some form of compression this does not reduce the total payload size much but still works well because of the effects outlined above. However, such an approach shifts the burden of alleviating the performance bottleneck from the developer over to the user.

**Batching and REST**

The REST architectural style does not prevent batching, nor does it explicitly account for it; REST simply states that components communicate via a standard interface to exchange representations of resources, and a resource is “any information that can be named” (2). Obviously, collections of resources are themselves resources by this definition, so an operation on a collection is an operation on a resource as well.

REST states, however, that the interaction between client and server must be stateless, and that the server must provide a generic, standard interface. In theory, this makes batching straightforward; we can imagine that for every operation \( O \) on a resource \( R \), we define an operation \( O_m \) that operates on collections of \( R \)’s. But what if we want to create a batch out of operations that affect multiple resources?

In the case of REST implemented using HTTP, even the straightforward case is not that straightforward. Should we use the same verbs for batch operations as for single ones? Should we use the same URLs? Should batch operations be implicitly atomic? The answer for many real-world web services is likely to be “it depends”. Ideally, we would like to refer to an
established standard or protocol specification rather than inventing our own. We happen to have such a reference – HTTP.

Since the case of operating on a single resource is, mathematically at least, just the special case of operating on \( n \) resources when \( n = 1 \), an elegant solution would use identical interfaces for both single and batch operations. Unfortunately, existing HTTP methods are heavily geared towards one-at-a-time requests. There are some easy wins, though: consider a web service which delivers books in XML format, by means of the URL books/{title}. We might then define the URL books/all to retrieve all books, simply joining the respective XML representations together beneath a common root element. While not a general solution, this follows the ground rules set up by REST+HTTP and is trivial to implement. We will cover this approach in more detail later on.
3. Methodology

In order to determine how different approaches to batching can be incorporated into RESTful web services, and which approach will be the more suitable under different sets of circumstances, it is important to consider the full spectrum of properties that differentiate the approaches. While the previous chapter focused on performance, (a subject to which we will soon return) software development is a constant balance of priorities, and performance is one of several; as the saying goes (10):

*Fast, good and cheap – pick any two.*

“Cheap” is the main villain here. Of major importance in almost any software development initiative is the relative cost of implementation for a solution. Complexity equals cost – both in initial development and maintenance – so simple solutions are often better in this regard. A more complex implementation also carries an additional risk of defects, the nemesis of “good”. Programmatic complexity and relative ease of adaptation must therefore be a primary focus of the analysis.

When working with Internet-connected services, compatibility with existing clients and intermediaries is not just a priority, but typically a must-have. The Internet as an ecosystem is hostile to services that are not compatible with existing hardware and software in wide deployment. Thus, we need to ensure that each approach does not violate the specifications of the protocols we use, or the (sometimes implicit) expectations of clients and intermediaries.

To cover the aspects just discussed, a theoretical analysis of the approaches will be carried out, using the broad field of existing knowledge about software engineering and Internet protocols as a basis.

Not a field study

We could perhaps perform this analysis by finding examples of each approach in real-world software development projects, reviewing the code and interviewing the developers. However, such examples may be hard to find (given that not all source code is open for outside review) and it can be difficult to know the actual cost of implementation post-facto. Furthermore, real-world software often contains complexities that arise from business requirements – and while such requirements may in fact have an impact on the choice of an approach for batching, they have no
relevance in a comparison of the approaches from a purely software perspective, so they are left outside the scope of this paper.¹

**Measuring performance**

When it comes to performance of software systems, theoretical analysis is fundamentally limited as a means of determining real-world performance. Two good guidelines are Rob Pike’s second and third rules of programming (11), which state:

2. Measure. Don’t tune for speed until you’ve measured, and even then don’t unless one part of the code overwhelms the rest.

3. Fancy algorithms are slow when n is small, and n is usually small. Fancy algorithms have big constants. Until you know that n is frequently going to be big, don’t get fancy. (Even if n does get big, use Rule 2 first.)

In conclusion, the only way to know for sure if a certain piece of code (and input of size n) outpaces another is to test and measure, using various numbers for n. In our case we will have multiple variables to work with.

**Implementation**

As a straightforward means of evaluating each approach, we will simply implement it in a simple web service, evaluating the performance by subjecting each implementation to identical loads and timing the response times. This has the added advantage of yielding easily repeatable and reproducible results – we can run the tests multiple times to reduce the effects of temporary changes in network conditions or other factors.

While the protocols and software packages (e.g. HTTP, web servers) involved are relatively complex and would take some time and skill to implement properly from scratch, modern operating systems and development environments do much of the work for us by including enterprise-grade versions of these tools out of the box.

Microsoft produces a widely used and well-tested software stack (12) comprising the Windows operating system, IIS web server, .NET application framework, C# programming language and Visual Studio integrated development environment.

¹ Perhaps someone pursuing a management degree would care to write a follow-up.
development environment. On this technology platform we can quickly implement client and server applications for each proposed approach to batching (discussed in detail in the following chapters) without having to waste much effort on the basics of web servers or network protocols. While software from other vendors could serve the purpose just as well, this combination was chosen mainly based on the author’s familiarity with it.

Running repeated tests using different values for payload size and request count will allow us to measure the relative impact of each approach. We choose these two properties as variables because they are likely to be highly variable in real-world scenarios, have the greatest impact on performance, and will allow us to learn whether the choice of one or the other approach is dependent on the likely usage pattern of the web service.
4. Approaches to batching

As we've established, the key to attaining high performance in multiple HTTP request lies in reducing overhead; there's not much we can do to the data itself. Broadly speaking, there are two ways to achieve lower overhead: we can try and make each individual HTTP request as efficient as possible, or try to make each request do more – instead of sending one request for each of resources A, B and C, why not simply send one request for ABC?

Obviously, any extension to the HTTP protocol we could imagine would be unsupported by existing HTTP/1.1-compatible clients and would need to "degrade gracefully", or be considered a completely different protocol. On the other hand, there are features already in the HTTP/1.1 specifications that can be of use, even though not all existing clients support them – we will look into this further very shortly.

On the other hand, a “standard” HTTP request with a body that cleverly composites multiple requests/responses into one, will be handled in the standard way by all existing HTTP clients and intermediaries, even if they cannot parse the payload – an eventuality which most will have no objections to. This moves the problem and solution from the protocol level to the application level (although technically still on the same level of the OSI stack).

4.1. HTTP protocol features

Throughout the lifespan of the HTTP protocol, several features have been introduced to help reduce overhead. Some implementations of HTTP/1.0 supported connection keep-alive (reusing the same TCP connection for multiple requests) and 1.1 introduced pipelining (sending multiple requests before waiting for a response). (13)

Connection keep-alive

Of the two, connection keep-alive is the more widely used, having been made the default mode of operation since HTTP/1.1; all HTTP/1.1 connections are persistent unless explicitly declared otherwise using a Connection: close header. A persistent connection will not be closed as the client finishes receiving the response, but is left open for subsequent requests, or until it times out. Keeping this timeout relatively short is in the interest of web server operators, as open connections consume resources.
It should be noted that web clients typically use more than one TCP connection to a given web server, in order fetch multiple resources in parallel. As this also places additional load on the server, the HTTP/1.1 specifications recommends limiting the number of concurrently open connections, but does not suggest a specific maximum number.

**Pipelining**

Pipelining is a feature in which a certain number of requests are sent to the server (over the same connection, implying a persistent connection) before the client receives any response. Such requests are “in the pipe”, metaphorically, and the “size” of the pipeline is the maximum number of pending requests. The server returns the responses in the same manner. This is especially useful on connections with high latency, where the waiting time between sending a request and receiving the response can be significant. The result is a significant increase in performance (14).

Pipelining, while promising in theory, has certain weaknesses (15) which contribute to its infrequent use on the web. The HTTP standard only permits idempotent methods to be pipelined. Since requests and responses are strictly first-in-first-out, there is a risk of a single expensive request blocking the processing of subsequent requests – an issue called head-of-line blocking. Such requests may have been processed more expediently on a parallel connection. Another issue relates to request throttling - given a large enough pipeline size, a malicious client could use a series of expensive requests to exhaust the resources of the server.

**Practical issues with pipelining**

Pipelining must be supported by both clients and servers, and any intermediaries, in order to function. The HTTP/1.1 specification (8) does not mandate conforming servers to support pipelining – only that they must not fail if a client chooses to pipeline requests. However, most modern web servers today support pipelining, but older servers and many proxies do not. As a result, all modern web browsers disable pipelining by default or use heuristics to determine whether a particular server supports it (4).

Unfortunately, there is no way for a server to hint to a client that pipelining is supported, and no way for a client to test that pipelining is available through any intermediaries than to test it. Even so, the previously mentioned issue with head-of-line blocking remains. Web clients therefore typically fall back to the safer approach of using multiple connections in parallel.
Because most modern web servers already support pipelining, enabling it for a web service running on such a server requires little to no effort or code changes on behalf of the implementer. Because many HTTP client frameworks support pipelining, it may be considered safe to use in scenarios where the server is known to support it and the circumstances are such that pipelining can be expected to deliver a significant speedup.

4.2. Composing multiple resources into a single response

Rather than rely on protocol features that merely make it easier to send multiple requests in succession, we can focus our efforts on getting the most out of each request. Instead of asking once for each resource, why not pass a list of all the ones we want? Instead of returning them one by one, why not return all of them? Using a structured data format such as XML or JSON, it’s just as simple to represent a set of entities as it is to represent only one.

Supposing we have an operation $O$ that returns one entity $E$, the XML representation of $E$ being something akin to:

```xml
<E id="1"> <!-- content of E --> </E>
```

We can then define a corresponding operation $O'$ which returns a set of entities $E[n]$:

```xml
<ArrayOfE>
  <E id="1"> <!-- ... --> </E>
  <E id="2"> <!-- ... --> </E>
  ...
  <E id="n"> <!-- ... --> </E>
</ArrayOfE>
```

Apart from the addition of a container element, the representation and XML schema are identical. Given a suitably relaxed schema, the same approach can be used for heterogeneous elements, but doing so may require additional metadata to be present to instruct the parser on the proper deserialization for each element type. For example:

```xml
<Array>
  <Item type="E"><E id="1" /></Item>
  <Item type="F"><F id="2" /></Item>
  <Item type="G"><G id="3" /></Item>
</Array>
```
This approach requires the corresponding XML schema to declare the element Item to have arbitrary content, so we gain flexibility at the expense of type safety. Of course, schemaless formats such as JSON already have this “feature”, and XML can be used without a schema.

Composition with complex operations
The examples in the previous chapter illustrate a simple retrieval operation, a HTTP GET, which is idempotent, and has relatively simple failure modes. Even so, if a client requests a set of resources, some of which exist and some which don’t, the server has to decide whether to fail the entire request (by returning 404 Not Found, for example) or to return only the found resources, leaving the client to check and compare if it got all that it asked for.

In either case, any composition of multiple requests into one introduces the possibility that some of the requests succeed while others fail, so we would like a way to return status information for each request, together with the response body, if any. At this point, one could imagine some kind of HTTP-over-XML tag soup:

```xml
<Array>
  <HttpResponse url="e/1" http-status-code="200">
    <E id="1"><!-- ... --></E>
  </HttpResponse>
  ...
</Array>
```

While technically workable, this breaks the separation between content and metadata, and is fragile; a syntax error in the body of one of the returned entities would make parsing the remaining responses very difficult, so we might not know if the operations succeeded or failed. For non-idempotent operations, this is crucial. While an enterprising developer might find a way around this particular weakness, the approach still does not work well with arbitrary content types. While textual data can be wrapped in a CDATA segment, binary data would have to be encoded (for example using Base64) which is inefficient and adds overhead on both client and server.

To resolve these issues, we would like to find a representation of multiple entities that maintains content-metadata separation, supports binary formats and does not require standard HTTP status codes and headers to be translated. Thankfully, the problem has already been solved in an area
where it is common to have multiple resources composited into the same document: e-mail, and the MIME standard multipart/mixed content type.

**Composite representation using multipart/mixed**
The multipart/mixed content type as defined in the Multipurpose Internet Mail Extensions (MIME) specification (16) is a suitable format for compositing multiple requests or responses into a single document. MIME terminology refers to each contained resource as a “body part”, each of which may have its own headers that override or complement those of the request/response as a whole. Body parts are separated by a divider which is defined in the top-level headers.

MIME, of course, is not really part of HTTP, although the HTTP specification refers to HTTP messages as “MIME-like”. MIME has no concept of “request” or “response”. MIME uses the Content-Id header to allow body parts to reference each other – allowing an image tag in a HTML document to display an image from another part of the same document, for example – so we can simply use this for the URL. To identify HTTP methods and status codes, we’ll have to invent some new headers.

Our HTTP/XML tag soup from before now becomes (the separator value is just `separator`):

```
--separator
Http-Status-Code: 200
Content-Id: <e/1>
Content-Type: text/xml

<E id="1"><!-- ... --></E>

--separator
Http-Status-Code: 200
Content-Id: <e/2>
Content-Type: text/xml

<E id="2"><!-- ... --></E>
```

And so on. Although somewhat wordier, it’s much cleaner. The separator, which is simply any arbitrary-length string that does not occur in any of the response bodies, is declared in the HTTP header of the composite document, for example like so:
The Content-Type headers of the body parts, along with any other standard headers, identify the type and encoding of the part, allowing for arbitrary data types to be mixed in the same response.

A small sub-problem is the choice of algorithm for generating the separator string. To conform to the MIME standards, the boundary should be alphanumeric and no more than 70 characters. In the body, it’s prefaced by a newline and double dashes (--) and also terminates the main body, ending in another set of double dashes. Generating a “sufficiently long” random string is not difficult, but depending on the size and nature of the body parts, actually making sure it does not exist in any body part can be tricky. However, given a string that is only 10-16 characters long, the probability of it occurring in the body is probably so small as to be safely ignored.

To reduce the risk of a separator being accidentally considered part of a body part, the Content-Length HTTP header can be used to explicitly give the length of each body part to the recipient.

**Complex request/response scenarios**

So far, we’ve been mainly concerned with the format of the response to a request, not so much with the request itself. Given that we invent a header to hold the HTTP method, it’s not hard to use the multipart/mixed format to carry multiple requests, each with its own headers. For instance, a request which POSTs a new entity to a list and then retrieves the updated list:

```plaintext
POST /e HTTP/1.1
Content-Type: multipart/mixed; boundary=p7f9Axy2
--p7f9Axy2
Http-Method: POST
Content-Type: text/xml
<E><!-- ... --></E>
--p7f9Axy2
Http-Method: GET
Content-Id: <e/>
```
The response might then be:

HTTP/1.1 200 OK
Content-Type: multipart/mixed; boundary=ff11Xd9b

--ff11Xd9b
Http-Status-Code: 201
Content-Id: <e/15>

--ff11Xd9b
Http-Status-Code: 200
Content-Id: <e/>
Content-Type: text/xml
<!-- list of E... -->

There is no need to identify which response belongs to which request if they are simply returned in the order given (as would be the case had they been sent separately) but one could be added if necessary.

Another possible use is in conjunction with standard HTTP caching methods such as Last-Modified to retrieve only those out of a set of entities that have changed (some headers omitted for brevity):

--abcd1234
Http-Method: GET
Content-Id: <e/1>
If-Modified-Since: Wed, 17 Sep 2014 22:24:00 GMT+1

--abcd1234
Http-Method: GET
Content-Id: <e/2>
If-Modified-Since: Wed, 17 Sep 2014 22:24:00 GMT+1

Body of server response:
Whenever multiple requests can be composited in this manner it naturally follows that we will save a great deal of network latency and overhead by not having to wait for the response for one request before sending another. One should be careful with this, however. In our POST-then-GET example, if the POST fails for whatever reason, the GET is wasted effort because the list might not have changed.

**Headers of the top-level document**

Given that a composite request can potentially contain different HTTP methods, and a response can contain different HTTP status codes (even if all sub-requests succeeded), there is the question of which method and status code should be applied to the top-level request. We can come to a conclusion on this by examining the idempotency and content of a request:

- If the request is **idempotent** and has **no body**, we should use the GET method.
- Otherwise, we should use POST.

The reasoning behind this is fairly straightforward when we consider interoperability. A simple batch retrieval of a collection, where the collection is given by the URL of the request (for example, books/everything), is a prime candidate for caching and needs no body detailing the specific resources to retrieve. GET requests are understood to be cachable by web proxies and other intermediaries, are always idempotent and never have a body.

For all other scenarios, particularly those that involve operations that create or modify data, we should use the POST method, because it has exactly the opposite properties to GET: requests are understood to be non-
idempotent and non-cachable, and are expected to have a body. We then don’t have to care too much about the methods of the sub-requests since it’s always safe to treat an idempotent operation as non-idempotent, but not the other way around.

The downside of POST is that caching proxies and the like will likely refrain from caching such requests (17), so clients may want to implement their own caching as necessary.

When it comes to the response, there are two possible approaches:

- Return a status code which describes the statuses of the sub-responses.
- Return a status code which describes the status of the batch operation.

The first approach is more suitable to simple requests like the one exemplified above. Since the request itself is simple and contains no complex instructions, the focus is on the content of the response; the status for a request to “get all books” could be 200 OK (here are your books), 304 Not Modified (nothing new since the last time you asked) or 404 (nope, no books here). This follows the model outlined earlier where “get all books” is really just a general case of the “get this book” operation, and so the responses should be analogous.

In more complex cases, it is valuable to know the status of the batch operation as distinct from the (possibly various) sub-operations. For example, in the case that the request body is a multipart/mixed format, we need a way to communicate to the client if they got the format wrong! However, even if the request is well-formed, individual sub-requests within may have failed. The client will be made aware of this anyway, upon parsing the response body, so why does it need to be reflected in the status code of the top-level response?

Following this approach, any composite request that is well-formed and understood by the server may well be responded to with 200 OK.

**207 Multi-Status**

There is an existing standard for returning multiple status codes from a request, 207 Multi-Status, defined in WebDAV (18). Somewhat unusually for a status code, it explicitly defines a response body format:
The default 207 (Multi-Status) response body is a text/xml or application/xml HTTP entity that contains a single XML element called multistatus, which contains a set of XML elements called response which contain 200, 300, 400, and 500 series status codes generated during the method invocation.

The 'multistatus' root element holds zero or more 'response' elements in any order, each with information about an individual resource. Each 'response' element MUST have an 'href' element to identify the resource.

This might be a good choice for some services, especially those that use XML as their main data interchange format, or that support WebDAV. Using 207 Multi-Status with a content type such as multipart/mixed would not be advisable.

4.3 An alternative approach: batching via proxy

In cases where technical constraints make implementation of batching operations in an existing web service difficult (for example, backwards compatibility, legacy or unmaintainable code or requirements to interoperate with third-party software) many of the advantages of batching can be realized by using a proxy service, which receives batch requests and splits them up into non-batched requests that are then forwarded to the intended destination. The responses are then collected by the proxy, composited together and relayed back to the client.

Because this approach simply hides the fact that operations are not really executed in batch at the destination server, the performance benefits are greatly dependent on the quality of the network between the proxy service and the actual server, because non-batched requests suffer horribly at the hands of high network latency. If the latency between proxy and server is the same as between client and proxy, batching will not yield any performance benefits over non-batched requests.

As a simple theoretical example where batching via proxy does improve performance significantly, consider a network where the client is connected to the proxy by a very slow connection that has a latency of 300 ms and a capacity of 1 Mb/s (these values are typical of a cellular “3G” link). The proxy and the server hosting the actual web service are using a much faster connection that has a latency of 1 ms and a capacity of 1 Gb/s (typical of a LAN). The client wishes to send 10,000 requests of 8,000 bits each to the
server. If the requests were non-batched, this would take about 3,080 seconds (nearly an hour):

\[ t_{\text{request}} = \text{latency}_{3G} + \frac{\text{bits per request}}{\text{bits per ms}_{3G}} = 300 + \frac{8000}{1000} = 308 \text{ ms} \]

\[ t_{\text{total}} = 10000 \cdot t_{\text{request}} = 3.08 \cdot 10^6 \text{ ms} = 3080 \text{ s} \]

Now consider the case where the client sends a batched request over 3G to the proxy, which then splits it and forwards the individual requests via LAN to the recipient. We will assume that the proxy cannot begin forwarding the individual requests until it has received the composite request in its entirety:

\[ t_{\text{receiveAll}} = \text{latency}_{3G} + \frac{10000 \cdot \text{bits per request}}{\text{bits per ms}_{3G}} = 300 + \frac{1000 \cdot 8000}{1000} = 8300 \text{ ms} \]

\[ t_{\text{request}} = \text{latency}_{\text{LAN}} + \frac{\text{bits per request}}{\text{bits per ms}_{\text{LAN}}} = 1 + \frac{8000}{1000000} = 1.008 \text{ ms} \]

\[ t_{\text{total}} = t_{\text{receiveAll}} + 10000 \cdot t_{\text{request}} = 90380 \text{ ms} \approx 90.4 \text{ s} \]

This example demonstrates that batching via proxy can improve performance significantly, when the connection between proxy and recipient is of sufficiently higher quality than the connection between sender and proxy.
5. Implementation

To ensure full control over the behavior of both server and client, in order to make the test results as accurate as possible, custom client and server applications were written based on the Microsoft .NET Framework. These applications were deliberately made as simple as possible, implementing the concept of a batching client-server application rather than trying to simulate a real-world example, so as to make the results independent of any scenario-specific factors and allowing properties like payload and number of requests to be configured independently.

5.1. Clients

Two separate client applications, though sharing a great deal of code, have been implemented for the purpose of performance testing. One is designed to test the effects of HTTP pipelining when making a large number of requests, the other to test the effects of using multipart/mixed to batch multiple requests and responses together. In both applications, the UI allows the user to specify the destination URL and the number of requests to be made. A report view displays the recorded times for each run.

Methods used for timing each run are documented by the operating system vendor to be accurate to <1 μs, but times when displayed are rounded to the nearest millisecond. For comparison purposes, this is more than enough, and makes the values easier to read.

The client for testing the effects of HTTP pipelining does so by simply saturating the target with requests and waiting for responses. Because the framework is configured with a connection limit of one per host, the client is required to wait for each response before sending the next request when pipelining is disabled.
The flowchart below illustrates the central part of the client.

The requests performed are simple GET requests, meaning the client can also be used to test against public web servers. Obviously this would entail using a lower number of requests so as not to put undue load on the server. Upon receiving the response, the client reads it into a buffer and discards the buffer, simulating what a real-world application might do but avoiding any parsing, computational or local I/O overhead.

Because the responses are processed on a background thread, it is possible for the client to start receiving responses before all the requests have been queued. The behavior of the application is identical regardless of whether pipelining is enabled or not.
The client for testing multipart/mixed requests is less complex, because it only needs to make one single web request, and so does not need to handle requests asynchronously:

Similar to the client for pipelining, each inner request of the multipart/mixed request is a GET for some particular URL on the same server. The boundary string separating the multipart/mixed bodies is generated from 10 random characters out of the U.S. English alphabet, plus digits 0-9.

Upon receiving the response, the client reads it into a buffer and discards the buffer. The client does not make an attempt to parse the response as multipart/mixed or do anything with the individual sub-responses, because doing so would only add a fixed overhead that would not be representative of a real-world application – where the time required by the client to handle each response might vary considerably.

Both clients are implemented as native (x86 instruction set) Microsoft Windows applications in the C# programming language using version 3.5 of the .NET Framework. For testing, they were compiled in release mode (no debugging symbols) using default settings for the Microsoft C# compiler.
These images illustrate the user interfaces of the client applications:

5.2. Server

A web service application was implemented for the purpose of performance testing and evaluation of implementation costs. URLs are used to differentiate between interfaces for testing HTTP pipelining, multipart/mixed requests, or both. In order to provide a realistic but predictable workload for the server, operations used for testing return buffers of random bytes exactly $N$ bytes in size, $N$ being given as a segment of the URL. For example, random/1024 would return 1024 bytes of body. This makes it straightforward to set up test runs simulating varying payload sizes.

The service has a specific URL for testing multipart/mixed requests. Requests made against this URL are parsed according to the multipart/mixed format, each child request is extracted and the server then calls itself, using the URL and headers specified in the child request. For each child request, the server creates a task which is assigned to a thread on the operating system thread pool, thus allowing tasks to be executed in parallel. Because the task queue is FIFO, requests are processed in the order they are received.

However, since requests may execute in parallel this implementation assumes idempotent operations. In the performance testing that was carried out, all requests were GETs, so this assumption holds. An implementation supporting non-idempotent operations would have to eschew parallel execution (at least for those operations), which would have a negative effect on the total performance of the system. With only idempotent operations the tests can be thought of as representing a “best case” scenario with regards to performance.
The flowchart below illustrates the processing of multipart/mixed requests in the server application:

The approach of using “recursive” requests was chosen because it has a relatively low cost of implementation and maintenance; over time, adding operations to the service contract of the server will not necessitate updating the multipart/mixed handling code, which would otherwise become a possible area of contention.

An alternate approach would use a discrete code path for each operation when called through a multipart/mixed interface. Such an approach would likely be faster due to a reduction in protocol overhead (e.g. no need to parse headers twice) but would require each operation to be duplicated in code; once for the regular interface, and one for batching. Incompatibilities between the two could result in hard-to-find bugs and the cost of implementation on an existing web service could potentially be very high.

The web service was deployed on a multi-threading web server (Microsoft IIS), allowing multiple requests to be processed simultaneously. This is a
prerequisite for batching using either HTTP pipeling, where requests/responses overlap in time, or multipart/mixed, where the sub-requests are processed within the scope of the main request.

The web service was implemented as a Windows Communications Foundation (WCF) service using the C# programming language and version 3.5 of the .NET Framework. For testing, it was compiled in release mode (no debugging symbols) using default settings for the Microsoft C# compiler. The service was deployed on several instances of Microsoft IIS, each running on a different physical host, for testing different network configurations.
6. Performance and experimental results

From a pure performance standpoint, batching appears obviously beneficial. If I have to make repeated trips to the store for each thing on the shopping list, obviously it will take longer than if I buy them all at once – a real-life metaphor makes the point almost childishly obvious.

Despite this, to get a rough idea of how much batching is likely to improve the performance of a web service, some degree of experimentation is necessary before starting an implementation effort. Simply implementing whatever sounds like it would improve performance, a sadly widespread process known as premature optimization, yields messy software that may well end up running slower than it otherwise would have.

While these tests were run on a somewhat simple service (see the previous chapter), the general structure of “request, process, response”, where the process takes some relatively consistent amount of time and the response is much bigger than the request – is very common in real-world web services today. Indeed, it is the way most requests public web sites work! Performance tests using this simplified model can therefore be used to give an approximation of how batching would affect performance in a more realistic use case.

These performance tests were carried out over a network connection with roughly 100 Mbit/s capacity and 1-2 ms latency. Both the server and client were reasonably modern PC’s running Microsoft Windows. All network settings were configured as per operating system defaults.

For verification purposes, some tests were carried out over a slower network link (residential DSL) and on different hardware. Because of more variable network conditions, these yielded less reproducible results and greater deviations between runs. However, overall the results were similar, both in terms of relative speed-up compared to non-batched runs and in terms of comparisons between the two methods.

6.1. HTTP Pipelining

The graphs on the following page illustrate the experimental results of timing 1000, 5000 and 10000 requests returning results of 1 kB, 5 kB, 10 kB and 50 kB, respectively. A test for each combination of payload size and number of request was run 5 times, and the values were averaged. The unit of time is ms.
Results in table format below. Values are rounded to two significant digits.

<table>
<thead>
<tr>
<th></th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 kB</td>
<td>1000</td>
<td>5000</td>
<td>10000</td>
</tr>
<tr>
<td>1 kB w PL</td>
<td>460</td>
<td>2300</td>
<td>4600</td>
</tr>
<tr>
<td>%</td>
<td>46%</td>
<td>46%</td>
<td>46%</td>
</tr>
<tr>
<td>5 kB</td>
<td>1400</td>
<td>7500</td>
<td>15000</td>
</tr>
<tr>
<td>5 kB w PL</td>
<td>640</td>
<td>3200</td>
<td>6400</td>
</tr>
<tr>
<td>%</td>
<td>46%</td>
<td>43%</td>
<td>43%</td>
</tr>
</tbody>
</table>
The following graph shows how the total time varies with increasing payload size at 10,000 requests.

<table>
<thead>
<tr>
<th>Payload Size</th>
<th>Average n/PL</th>
<th>Average w/PL</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 kB</td>
<td>1800</td>
<td>9100</td>
</tr>
<tr>
<td>10 kB w PL</td>
<td>1000</td>
<td>5000</td>
</tr>
<tr>
<td>%</td>
<td>56%</td>
<td>55%</td>
</tr>
<tr>
<td>50 kB</td>
<td>6200</td>
<td>30000</td>
</tr>
<tr>
<td>50 kB w PL</td>
<td>4700</td>
<td>24000</td>
</tr>
<tr>
<td>%</td>
<td>76%</td>
<td>80%</td>
</tr>
</tbody>
</table>

As is evident, enabling pipelining while keeping other variables constant yields immediate performance gains, reducing the time between request and response by more than 50% in some cases.

Because each request still carries significant overhead (because it is a full, standard HTTP request, headers and all) the improvement in performance depends highly on the payload size. Large requests gain less from being pipelined because most of the time is spent transmitting the payload, rather than waiting for the network.

Because pipelining is unlikely to ever lead to a reduction of performance, it is generally safe for clients to enable it, given that the server is known to support it. As mentioned earlier, servers “in the wild” generally can’t be
counted on to implement pipelining correctly, so many clients (including most web browsers) disable it by default. With known-good clients and servers however, there should be no issue.

There is an unfortunate tendency of some “RESTful” web services to implement all operations using the POST verb. Because pipelining is only allowed for idempotent operations (normally HEAD and GET, possibly DELETE and PUT depending on the implementation) these services will not gain any performance from enabling pipelining. Such services should be refactored to use the appropriate verb(s).

Generally, use cases for batching that rely heavily on intermixing idempotent and non-idempotent operations – for example, creating a resource, performing some operations on it and retrieving the result – will not benefit much from pipelining. In case of a network failure, it can be hard to know which operations succeeded and which did not. Performance will be reduced because the non-idempotent operations will “stall the pipe”, preventing any further operations from being pipelined until they complete.

6.2. Batching using multipart/mixed

The graphs below illustrate the experimental results of running the same tests as for HTTP pipelining, but using a “naïve” implementation of multipart/mixed.

Note that while the blue lines in these graphs and in those on the preceding page both illustrate the exact same operations being carried out (running the requests in serial over a single connection to the server) the timings are slightly different. This is most likely due to network conditions varying slightly between tests.
Results in table format below. Values are rounded to two significant digits.

<table>
<thead>
<tr>
<th></th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 kB</td>
<td>1000</td>
<td>5000</td>
<td>10000</td>
</tr>
<tr>
<td>1 kB batch</td>
<td>200</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>%</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
</tr>
<tr>
<td>§</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 kB</td>
<td>1500</td>
<td>7300</td>
<td>15000</td>
</tr>
<tr>
<td>5 kB batch</td>
<td>300</td>
<td>1600</td>
<td>3300</td>
</tr>
<tr>
<td>%</td>
<td>20%</td>
<td>22%</td>
<td>22%</td>
</tr>
</tbody>
</table>

Results shown in graphs for 1 kB, 5 kB, 10 kB, and 50 kB batches.
The following graph shows how the total time varies with increasing payload size at 10,000 requests. When illustrated this way, the dramatic difference for large payloads is very obvious.

<table>
<thead>
<tr>
<th>Payload Size</th>
<th>Time 1 kB</th>
<th>Time 5 kB</th>
<th>Time 10 kB</th>
<th>Time 50 kB</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 kB</td>
<td>1900</td>
<td>9200</td>
<td>18000</td>
<td></td>
</tr>
<tr>
<td>10 kB batch</td>
<td>390</td>
<td>1900</td>
<td>3900</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>20%</td>
<td>21%</td>
<td>22%</td>
<td></td>
</tr>
<tr>
<td>50 kB</td>
<td>6300</td>
<td>29000</td>
<td>59000</td>
<td></td>
</tr>
<tr>
<td>50 kB batch</td>
<td>1200</td>
<td>6000</td>
<td>12000</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>19%</td>
<td>21%</td>
<td>20%</td>
<td></td>
</tr>
</tbody>
</table>

Even in a simple implementation, the performance gains improved by using the multipart/mixed method are significant, reducing time from request to response by as much as 80%. The only latency affecting the request is for the initial packet to arrive, and for the initial packet of the response; the rest is mere streaming data with no back-and-forth required at the HTTP level (the transport layer will still have to acknowledge the packets, of course).

This is optimal for networks that have high latency compared to their capacity, which is most of them – latency being limited by the laws of physics (the speed of light) whereas there seems to be some ways to go
until we reach the maximum possible capacity of a point-to-point network link. Mobile networks are especially good examples of this imbalance.

Because this approach requires an adaptation of the service interface to be applied on an otherwise unprepared web service, it is comparatively harder and more expensive to implement than pipelining. Larger requests and responses may also put additional load on client, servers and network equipment – we lose the inherent “load balancing” effect of network latency, instead being forced to process all the data we asked for all at once. In addition, there may be interoperability or compatibility issues hindering the implementation, given that the technology is not widely deployed today.

As such, implementation will require cost-benefit analysis to determine if it is really worth it – and implementing it on the server side will not automatically realize any benefits to clients that do not implement it in kind. This is also in contrast with pipelining, which is already widely supported, although not enabled by default, and not completely free of compatibility issues.

In summary, the choice of pipelining or multipart/mixed methods for batching should not come down to merely a matter of theoretical performance, even though the figures are quite clearly in favor of the latter method. However, any web service that frequently processes large volumes of requests from individual clients would certainly stand to benefit greatly from any variety, given that it was competently implemented.
7. Conclusion

It is intuitively easy to grasp that processing data in bulk can be more efficient – and therefore more performant – than processing transactions, rows or records individually. In this thesis, the aim has been to determine if we can achieve these benefits in a RESTful web service. Chapter 4 provided a detailed overview of the two general approaches and chapter 5 described the specific implementations used for performance testing.

In chapter 6, the test results clearly showed that performance gains of up to 50% using HTTP Pipelining and 80% using multipart/mixed were achievable under controlled conditions. Based on these results, we can conclude that a web service designed in accordance with RESTful principles – the same principles that form the basis for the extreme scalability and growth of the World Wide Web – can reap great performance benefits from batch processing.

Furthermore, the performance benefits of compositing requests and responses (e.g. multipart/mixed) can be significantly greater than those yielded from HTTP pipelining, but bring a considerable higher cost of implementation. While HTTP pipelining can significantly improve performance as well, it is limited in what types of operations can be optimized, and is not without interoperability issues.

Having carried out these tests and the preceding development work to set up the applications, it is my opinion that many real-world web services could reap considerable performance improvements at a reasonable implementation cost by supporting batched operations where usage patterns indicate they would be the most effective. In addition to improving client performance, reducing network overhead and hardware resource utilization brings economical and environmental benefits.

A limitation of this thesis is that it concerned itself only on the technologies widely deployed on the World Wide Web at the time of its writing; there is significant work going on with protocols like SPDY (19) (the basis of the upcoming HTTP/2) that aim to solve these very same issues by replacing HTTP. So far, while SPDY has seen limited adoption (20), HTTP/2 remains in a planning stage. The implication is that once a new protocol to replace HTTP/1.1 gains broad support, adopting this new standard should be thoroughly considered.
8. References and acknowledgements

This thesis has been in the works for a long time, in stops and starts, and would never have been finished were it not for the continued nagging and/or gentle encouragement by my parents, Richard and Monica. I owe great thanks to my examiner, Associate Professor Johan Montelius, for his infinite patience and invaluable guidance through the process; and to my friend Emelie for always being there for me.

References


