SWEN90007 Assignment Part 4

Performance discussion

Introduction

In this report we aim to discuss the performance implications of the patterns and principles we chose to apply in our application. We also discuss patterns that weren't implemented, and the reasons for not doing so.

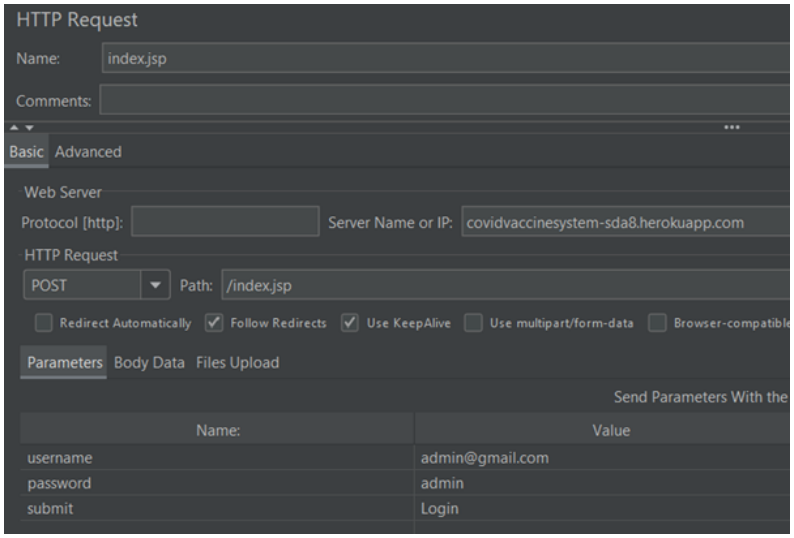
Possible improvements

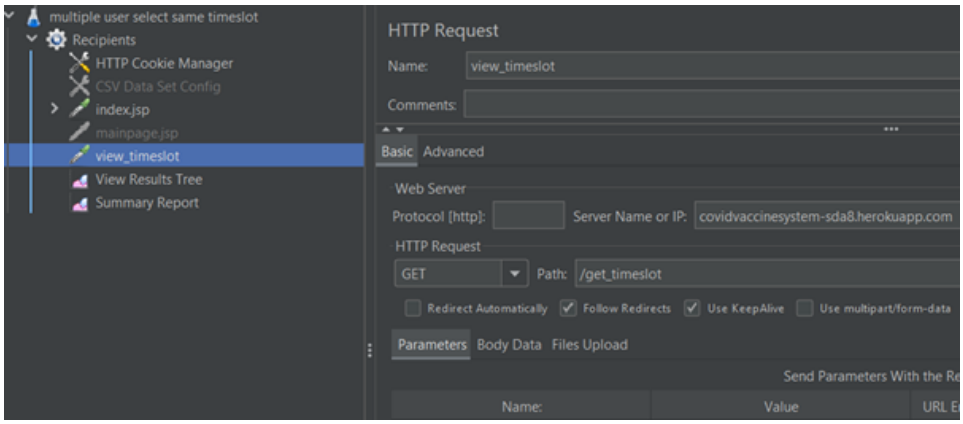
The following section describes the design pattern implemented in our application, and the impacts of each on performance.

Sample JMeter test

The performance of our system can be reflected through various perspectives including latency, responsiveness, and throughput. In part 3 our group used JMeter to simulate concurrent requests and added locks to resolve conflicts. For performance testing, JMeter is also a handy tool to test response time and throughput of a java application.

Since the core of our application is for users to interact with the database, we implemented different design patterns to optimise this process to satisfy the potential business needs of the application. We ran a simple test of simulating a large number of admin users logging in at the same time and viewing all the time slots in the system, configured as below, where the users first log into the system and then request access to the timeslot table. This can directly demonstrate the overall performance of our architecture and more or less reflect on the performance of other operations to the database such as adding timeslots.





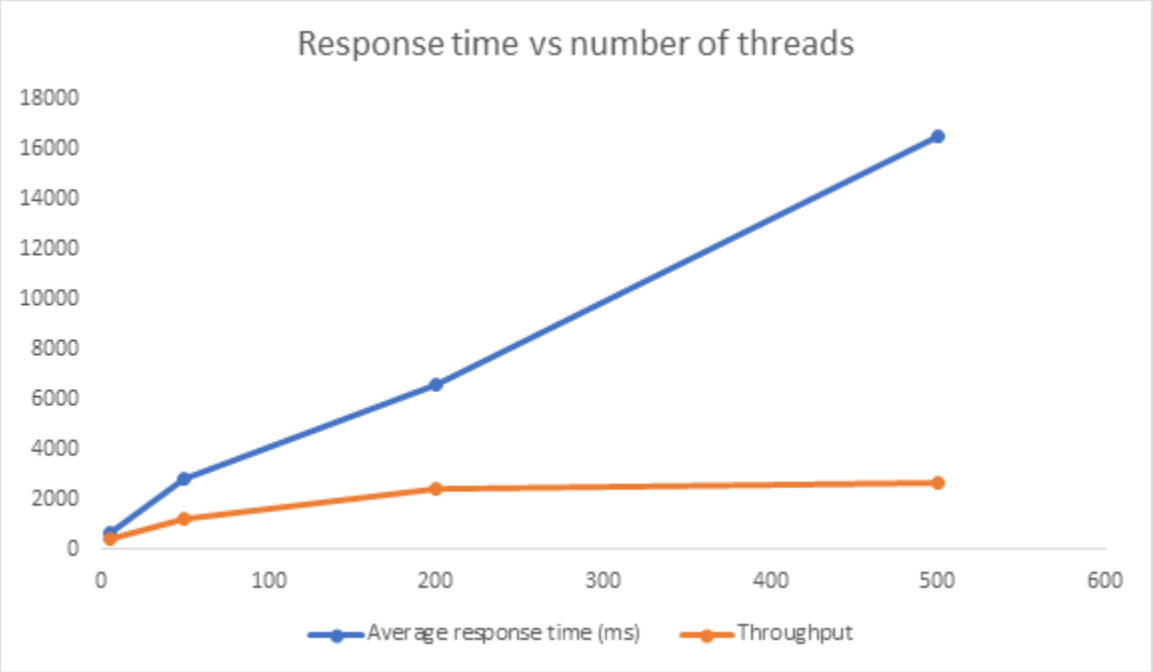
We ran the test with 5, 50, 200 and 500 threads respectively and gathered the following results:

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput
index.jsp	5	1027	989	1069	33.75	100.00%	2.8/sec
view_timeslot	5	320	303	343	14.92	0.00%	4.8/sec
TOTAL	10	673	303	1069	354.16	50.00%	4.7/sec

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput
index.jsp	50	4782	1620	7581	2115.39	100.00%	6.4/sec
view_timeslot	50	930	303	3475	1160.42	0.00%	7.6/sec
TOTAL	100	2856	303	7581	2573.00	50.00%	12.2/sec

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput
index.jsp	200	11897	2199	15509	2582.81	100.00%	12.5/sec
view_timeslot	200	1370	312	6135	991.71	0.00%	14.2/sec
TOTAL	400	6634	312	15509	5615.42	50.00%	24.4/sec

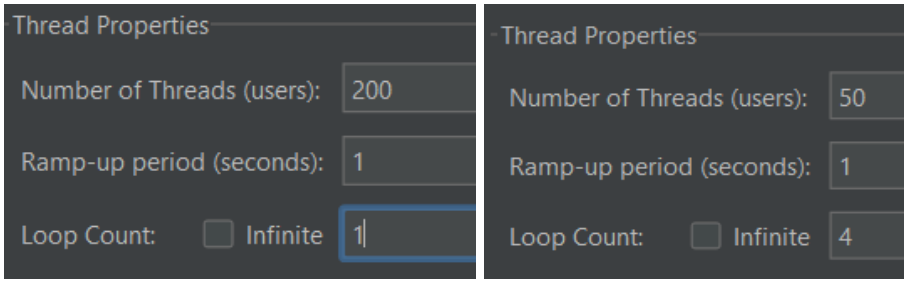
Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput
index.jsp	500	28683	25770	33341	2173.81	100.00%	14.7/sec
view_timeslot	500	4349	0	8177	2328.07	8.60%	45.2/sec
TOTAL	1000	16516	0	33341	12373.40	54.30%	27.1/sec



As plotted in the graph the system demonstrates a linear growth of response time with increasing number of requests, which is acceptable in terms of scalability.

Identity Map

The identity map can improve performance by storing an object list in the cache to avoid fetching the same object multiple times from the database. To test this feature of our system, we ran the same JMeter test in the previous section with two different configurations: 200 unique threads versus 50 threads each looping 4 times.



200 threads:

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput
index.jsp	200	11897	2199	15509	2582.81	100.00%	12.5/sec
view_timeslot	200	1370	312	6135	991.71	0.00%	14.2/sec
TOTAL	400	6634	312	15509	5615.42	50.00%	24.4/sec

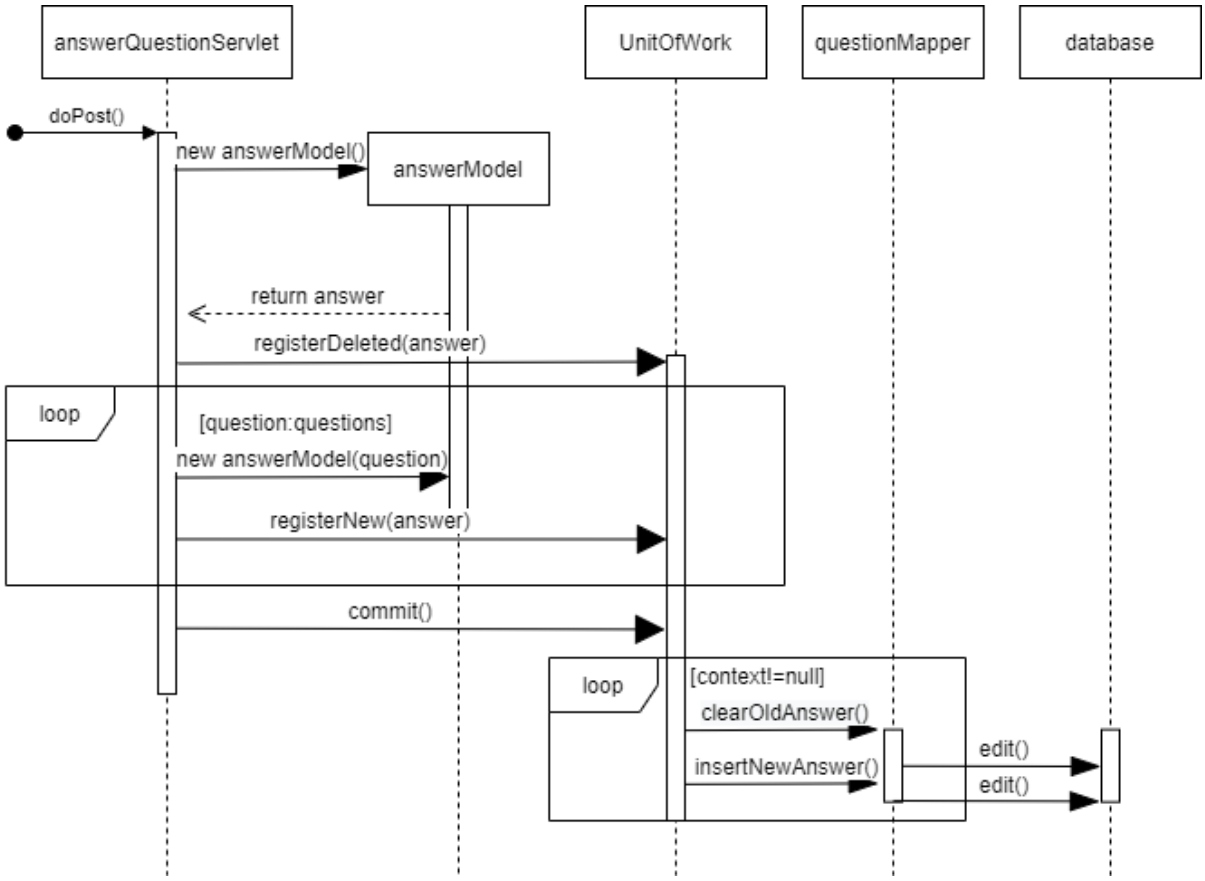
50 threads with 4 loops:

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received K...	Sent KB/sec
index.jsp	200	927	257	3926	1192.69	25.00%	31.5/sec	39.19	13.34
view_timesl...	200	419	292	580	62.42	0.00%	37.3/sec	86.40	8.21
TOTAL	400	673	257	3926	881.93	12.50%	60.1/sec	106.97	19.35

The total number of samples are both 400 however when looping the same threads multiple times has an average response time of 673 milliseconds compared to 6634 milliseconds when running 200 separate threads, as well as much less throughput, indicating that the same user will not fetch for the same data when it is already fetched.

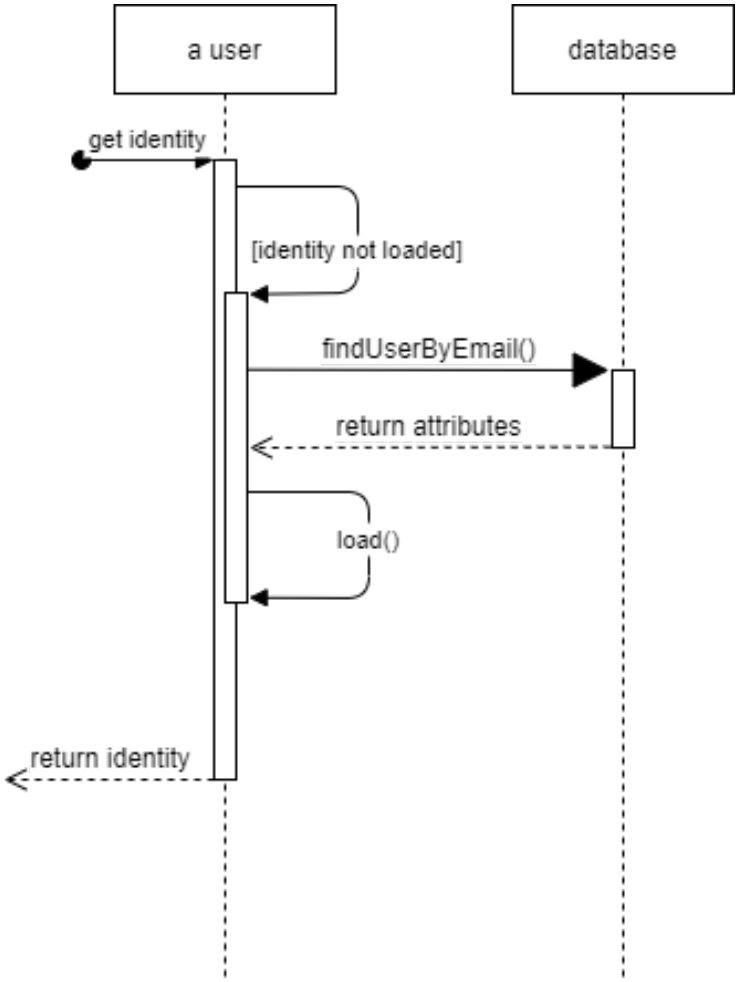
Unit of Work

We used the unit of work pattern to encapsulate multiple queries issued by the user into one unit and submit them together as a whole. It keeps track of the list of objects that are modified or updated, and flush all when an in-memory transaction is finished. The unit of work pattern improved the performance of our system when the user wants to modify multiple attributes in the database. For example when the user answers a questionnaire for the eligibility of a vaccine type, instead of modifying each answer question at a time, the system interact with the unit of work pattern and registers all question answers as a new object, reducing the number of connections between the presentation layer and the data source layer, therefore increase the performance.



Lazy Load

Lazy load is another pattern we used to increase performance. It is used when loading data from the database. Instead of fetching the full version of a desired piece of data and then displaying it in the web, the client only requests partial information to proceed display. For example when a HCP wants to fetch a list of users from the database, a minimal version of the user table which only contains the email attribute will be loaded. Then later when the user requires more detailed information the system will then fetch them according to the email as the unique identifier.



DTO and Remote Façade

Data transfer objects are data objects that do not contain any business logic and since we use JSP and controller as part of the architecture DTOs will not be relevant as the data are collected and handled by servlets.

Optimistic/Pessimistic Offline Lock

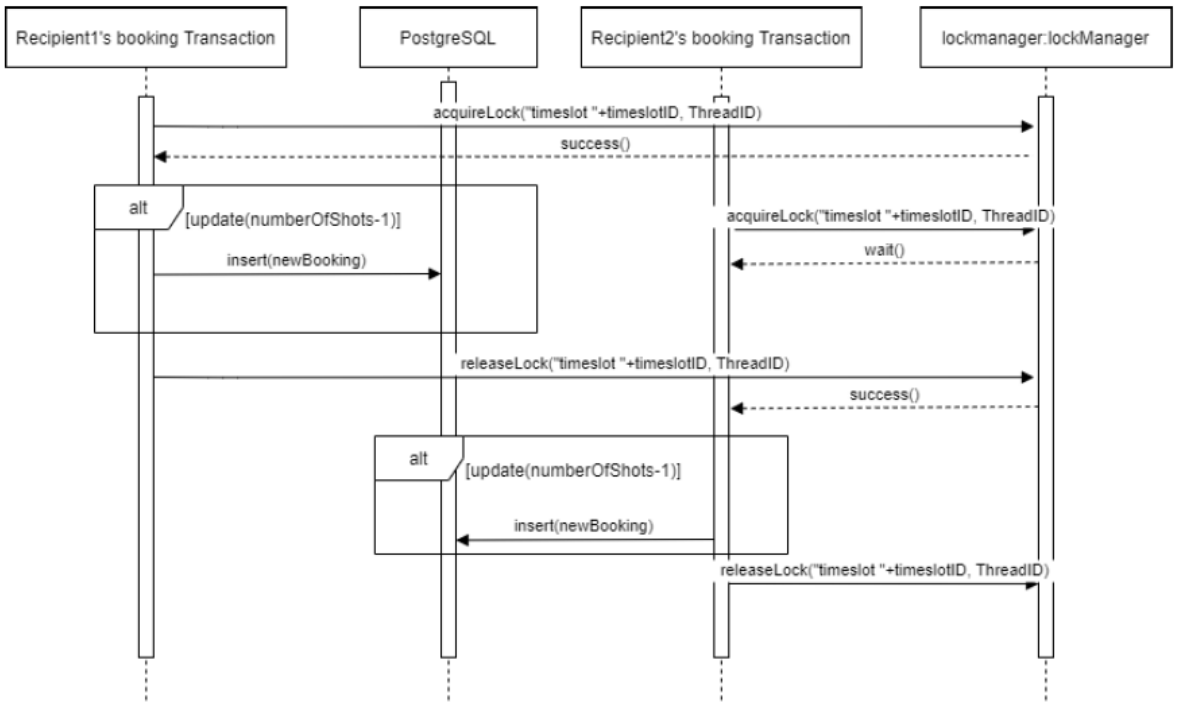
In part 3 we applied several pessimistic locks to handle specific concurrency issues of the system. Locks in general will increase the response time since a user can be requesting a data resource that is being locked, resulting in a longer wait time for the user to view the data or complete a transaction. Adding locks sacrifices some performance in exchange for more secure and reliable business functionality.

Principles

A discussion of the design principles we did and did not use in our application, and their effects on performance.

Pessimistic Locking

To manage concurrency in our application we implemented Pessimistic Locking. The specifics of this implementation were detailed in the Part 3 report, and this section will only touch on the performance implications. This locking is used to handle situations where multiple users will attempt to change the same database row, for example two users try to book the same vaccine timeslot or two instances of a healthcare provider modify a timeslot. The first of these cases is shown in the diagram below:



When choosing a locking approach, we had to decide between pessimistic and optimistic locking. We chose pessimistic, as we felt the chances for conflict were low and we liked the stronger guarantees it gave us. When a user attempts to modify a database row, they first request a lock. This adds a small amount of application latency, but it is something we're comfortable with given the necessity. The real performance hit is when two users request the same lock - only one user will be granted access, and the other thread will need to wait until the first is completed. This will lead to a more significant application delay for the second user, as they will need to wait for the first database transaction to complete - with the added latency that comes with database requests. Again, while this is a more significant delay we chose to accept it because it out of necessity and because it only happens infrequently.

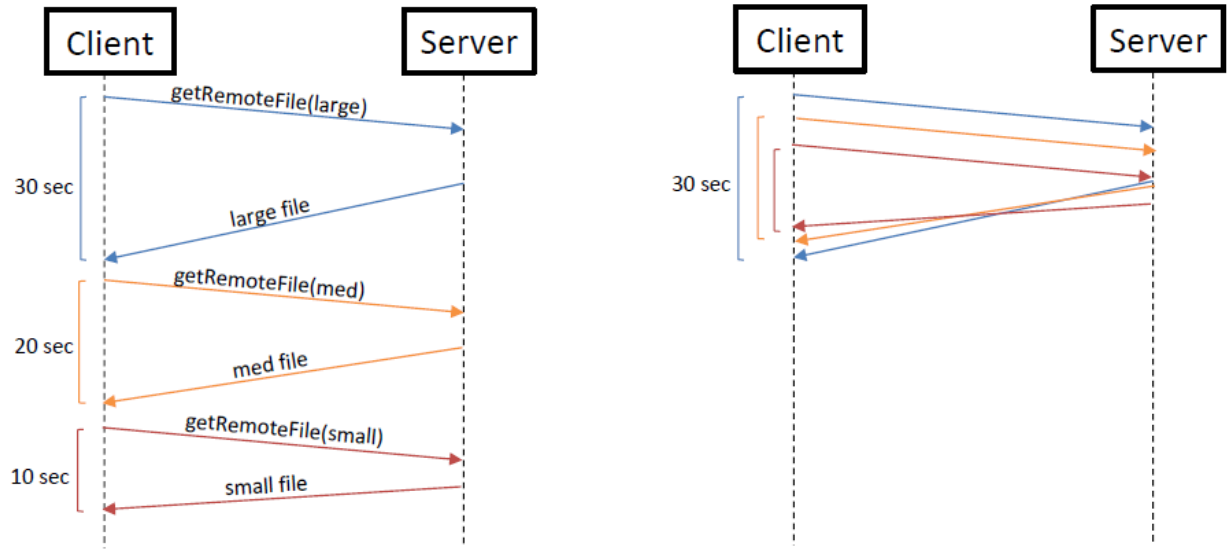
The alternative, optimistic locking, is not without its own downsides. Every single tread that tries to modify the database will have to keep track of version numbers when performing a transaction. This in itself will lead to a small amount of application delay. The real delay occurs though when version numbers do not match after performing a transaction - this means that another user has already updated that table at the same time. In this case, the transaction will need to be aborted and the database reset - a costly process that will force all other threads to wait.

Bell's Principle

Bell's Principle is that "the cheapest, fastest and most reliable components of a system are the ones that aren't there". This is a loose rule that says simple designs tend to perform better than more complex ones. We feel that our system has broadly followed this rule across its development, partly because of the staged development required by this assignment. Our system started as a very simple collection of Java Server Pages and Servlets. Each JSP had a respective Servlet, and there was a single class that controlled access to our database. In each part of the assignment the requirements grew, and in response we added new components and functionality to meet them. With this type of development we started from a place of simplicity, and only added complexity when it was absolutely necessary. We feel this has kept our system as lean as possible, which according to Bell's principle should give us better performance on average.

Pipelining

Pipelining is a design principle that aims to increase throughput without improving response time. It does this by fulfilling multiple requests from the client to the server asynchronously rather than changing requests in series. The follow diagram from the lecture slides displays this:



On the left is the traditional method, and on the right is a pipelined method. The benefit of pipelining is that users spend less time waiting for resources. The downside is that to achieve

this multiple threads in the server need to be dedicated to a single user. There is also increased complexity - both the client and server need to keep track of request ordering, and additional components need to be added to the server to facilitate it.

We chose not to implement any pipelining in our application. The reason for this is that users in our application never experienced a significant delay waiting for resources - most pages were already filled by a single HTTP request, rarely multiple. The downsides are also quite significant in the context of our application. Each thread used for pipelining is one that can't be used to fulfil the request of a new user. With a Vaccine booking system we anticipate a large number of unique users, the entire population in fact, that want to book vaccines. Our priority is to serve as many users as possible, with the trade off that each user could experience slightly longer loading time. The addition of extra complexity to our application also goes against Bell's principle. The more components in our server, the more chances there are for problems and bottlenecks. We've only added components as absolutely necessary, and we don't think there is a case for pipelining.

References

- Week 9 Lecture Slides - Performance