Moritz Bergemann

DC Assignment 2 Report

Practical Design Choices

Practical 1

I chose to use FaultException for the handling of errors in my application. As these can be thrown across the network boundary, they allow for a natural programming style – one must only write code to catch the FaultExceptions an RPC function may throw. No error data (e.g., success boolean, error message) must be contained on the return value of the RPC function – if the function is not successful, it can simply throw an exception with a relevant message to be caught by the network receiver.

Other unique design decisions were related to the transferring of profile images. I chose to use ten pre-set profile images of AI-generated faces as my content. The high resolution of these images (1024 x 1024) was over the default transfer size limit for WCF remote functions, so this had to be manually adjusted (using the MaxReceivedMessageSize property of NetTcpBinding).

The large size of the images created another issue – while the database object could store hundreds of thousands of 'Profile' structs made of integers and strings in memory without issue (each being a few bytes), this was not feasible if each struct contained a 500KB image. Therefore, I developed a system of image IDs – as each profile image was randomly used for a number of users, I assigned an integer ID to each image and stored them in memory only once. Instead of storing the entire image, each profile struct could then simply contain the image ID. Other applications could then query the database with an image ID to return the full bitmap, allowing hundreds of thousands of repeated profile images to be stored.

Rather than sending data as a single large bitmap, I chose to transfer it as a stream to allow large profile data (or potentially other types of data in the future) to be sent from the data tier. As streamed data is not sent in its entirety, it is not possible to have it as an "out" parameter as with other parts of the profile. Therefore, I created a new method on the data tier server specifically for retrieving the profile image for a given index. In the WPF client, this method is called to load the client after other details have been loaded.

Practical 2

The business tier added another level of complexity to the image streaming problem. As the image stream data could not be returned with the rest of the data, having all data returned in the "Search by last name" function was not possible. One approach could have been to have two search functions – one that returns the image stream and one for other data – but this would have required searching the database twice, needlessly inefficient. I initially solved this by having the server statefully store the index of the most recently searched user, and then provide two functions for returning the image/other details for this latest index, though this would not workwith multiple concurrent requests. Finally, I resolved this issue by having the "search by last name function" return the image ID from the database as an integer, and providing another business tier function that retrieves a profile image based on its ID.

Practical 3

Sending the image from the business tier web service to the presentation tier required a design change, as the bitmap image could not be easily sent as a data stream over HTTP. I resolved this by having the business tier base-64 encode the profile image (after it was retrieved from the data tier)

and attach it as a string to the JSON object sent to the client. As the image was now being sent as a string and not as a stream, it also meant all profile data (including the image) could be retrieved with a single request, simplifying the application.

Practical 4

Managing errors with the inaccessible BankDB DLL was a challenge. The DLL throws a variety of custom exceptions, however none of these can be explicitly caught as they are private to the BankDB DLL. I instead was required to catch base exception when calling BankDB functionality, and then throwing a more relevant exception by anticipating what will go wrong in BankDB.

Regarding responsibility for functionality between the data & business tier, I had the data tier contain only the functionality required to make the most basic actions statelessly. This includes adding users/accounts, queueing transactions, and making deposits/withdrawals. This makes the data tier as versatile as possible for any future functionality. The business tier's operations were designed to be convenient and logical for the end user. For example, making a transaction in the business tier calls the data tier to add the transaction to the queue, process all transactions (which should only be this one), and then saves the database. This combination of business and data tier makes the application's functionality easy to implement in the presentation tier, while also remaining versatile for other applications.

In terms of error handling, I chose to use custom .NET exceptions within each application (business tier and data tier) but use HTTP response codes for the ASP.NET boundaries (via the HttpResponseException class). While this reduces the specificity of errors that can be thrown (being limited to common HTTP error codes), it vastly improves platform compatibility. HTTP error codes are understood by any device performing web operations, and error descriptions can still be sent through the response content. This also improves compatibility with AJAX requests – as these expect errors to be in HTTP error code format, the built-in functionality for handling errors can be used.

Practical 5

When using jQuery & JavaScript functionality vulnerable to XSS, exploitation was trivial. Simply setting the user name to <script>alert(0);</script> would cause an alert to be shown if the jQuery .html() property was used to display the name on screen. In cases where html is not immediately executed or if <script> itself was not allowed, using the "onerror" attribute of an <img> tag with a bad image reference would allow arbitrary code to be executed in the same way.

To protect against XSS, I utilised jQuery and JavaScript's built-in secure displaying functionality – the '.val()' method and '.textContent' properties respectively. These properties rely on established escaping functionality to avoid any code being executed, even if one attempts to bypass it specifically – attempts to exit the added '<text>' field with a '</text>' were not successful.

I used the given demo page to display a means of avoiding XSS when not using secure built-in functionality. Before redisplaying user input to the page, I call a JavaScript function that removes any dangerous characters ('<' , '>', ';', etc) from the input. As at least some of these characters are required to perform an XSS attack under normal conditions, an attack from this vector is prevented.

Practical 6

When creating the scoreboard system, I chose to move responsibility for reporting score increases/completed to the server thread, whenever a completed job is posted to the server. This was done primarily as a security consideration, as having clients self-report the completion of a job makes it easier to exploit by falsely reporting jobs. There are still no actual security measures in

place against this, so a client can still falsely report that “another” client (actually themselves) completed a large number of jobs. However, moving this responsibility to the server thread makes protections easier to implement, as self-reporting could be prevented by not allowing clients to update the score for their own endpoint (by comparing the reported endpoint and the endpoint making the report).

A major issue with the basic implementation of this application is that when a job is posted by any client, all clients (unless working on other jobs) will download the job and attempt to work on it, often causing clients to work on the same problems and wasting processing time. Therefore, I implemented a queueing system where, once a job is downloaded, it becomes tagged as “in progress” within the server and becomes invisible to all other clients. This meant each job would only be downloaded and completed by a single client, leaving other clients free to respond to other jobs and massively improving performance.

The above tagging design did however mean that, if a client crashes or loses connection while completing a job, the job would be permanently marked as “in progress” and would never be completed. Therefore, I also implemented a timeout system where each job was also flagged with a timeout time, one minute from when it was originally downloaded. All “in progress” jobs are then periodically checked – if a job’s timeout time had passed, it is removed from the “in progress jobs” and put back into the available jobs queue. If the original client later does return with the job, the result is ignored. This does mean jobs that take longer than the timeout time will perpetually be processed and ignored, but this could be fixed by making the timeout adjustable.

Practical 7

Having a logging system was critical for the cryptocurrency practicals, as the results of an action could not always be displayed back to the user. When a transaction is added to the mining queue and is then later found to be invalid given the current blockchain state (long after the transaction was posted), there is no easy way of informing the original transaction creator. Therefore, all transactions (successful or failed) are logged to a file to act as a record of all transaction attempts.

I also chose to move all hashing-related code onto the `Block` object itself. This way, as long as knowledge of the block object is coming from a shared, trusted source, once can be sure all parts of the program are using the same hashing algorithm. It also makes the algorithm much easier to modify in the future.

Practical 8

The key challenge in this practical was the ensuring of eventual consistency. While I initially only had clients verify the local blockchain immediately after adding a block, this led to situations where the most popular blockchain would change *after* the check is performed, leaving some clients out of sync. I handled this by also running the “most popular blockchain” check at a set interval when no mining was taking place (as to not interfere with other blockchain modifications).

This still leaves the issue of some miners finishing mining earlier than others and thus not having the most popular chain in the subsequent check. This may cause transactions to be lost, though it also seems to be an inherent issue with this distributed blockchain design, without significant deviation from the specification.

XSS Discussion

Cross-site scripting (XSS) is one of the most prevalent and dangerous attacks present on the web today. There are three main types of XSS, but the key principle behind each of them is the same:

Web pages use JavaScript to execute actions in the browser, which can simply be stored in text form in an HTML page using the `<script>` tag (or in various other ways). The danger comes from the fact that there is no inherent difference between the way that the JavaScript/HTML of a page and the page content is stored in the file – it is all in text. This means that if text data added to a page contains HTML code in the right place, the browser will treat it as HTML and render it as such. This can have relatively harmless results, such as allowing a user to embolden their text by surrounding it with a '<b>' tag. However, given the presence of the <script> tag and other indicators of JavaScript, it can also allow an attacker to inject and execute arbitrary code on a victim's browser.

Reflected XSS

Web servers will often use programs (such as ASP.NET) to modify the content of a webpage based on information attached to the user's request (such as query strings or session data). This data can have XSS attacks attached to it. For example, a search engine might expect the query string "?search=*[SEARCH]*" in an URL request, which will be displayed on the returned page as "Search results for *[SEARCH]*". If input is not properly sanitised, sending a request like "?search=<script>evil_code();</script>" will cause the code to be inserted into the web page and run. This is especially dangerous as an URL to a trusted website that contains the attack as a parameter could be sent to an unsuspecting victim, who will have the JavaScript code executed in the returned page when they click the link.

Persistent XSS

Persistent XSS occurs when the JavaScript code is not only returned when a single request is made but is in some way stored in the web application's database and then served to future users of the webpage when data is retrieved. For example, if the body of a post in a web forum is not properly sanitised, an attacker could make a post with the content "=<script>evil_code();</script>". Any visitors to the website that load the attacker's post would have the code executed on their browser. In this way, a single injection can easily affect a very large number of users.

DOM XSS

DOM XSS is like reflected XSS in that it occurs in response to certain data being entered into a page. The main difference is that the attack never has to involve the web server. As JavaScript can be used to modify the content of a page, DOM XSS occurs when legitimate JavaScript on the page retrieved from the web server is used to inject code into the page and execute commands. For example, if a single page search application used JavaScript to dynamically load the text "You searched *[SEARCH]*" in an non-sanitised way onto the page when a search is entered, entering "<script>evil_code();</script>" into the search bar will cause the attack to be executed.

Overcoming XSS

The most straightforward way of overcoming XSS is by simply never writing untrusted input to the page, preventing any malicious code from being injected in the first place. This is not always feasible however, so the next-best option is to sanitise any untrusted input when it is first received and/or when it is displayed to the page. This can be done by removing/disallowing potentially dangerous characters (such as '<', '>' and '"') or by escaping special characters through replacement with a substitute/representational character. Some tools for escaping dangerous characters are already built into many modern web browsers, such as the JavaScript 'textContent' property. All of these methods stop input inserted into the page from being loaded in the same way as legitimate JavaScript code, and therefore can be used to protect against XSS.