SR&ED Claim

By Greg Nacu, 2015

1) Change Management / DB Structure

What scientific or technological uncertainties did you attempt to overcome uncertainties that could not be removed using standard practice?

1. Over the previous few years we have been developing a commercial multi–location, retail point of sale system which is built on top of a RESTful API that backends on a multi–node, NoSQL database. When we began no one had ever tried to build a point of sale system on top of this kind of data store. As a consequence of this decision we unlocked many advantages but were navigating development waters that were entirely untested. One critical goal of the project was for remote clients to receive realtime updates about the state of products, customers, inventory and sales from other clients, and to also support offline availability.

2. In previous years we were able to utilize a novel feature of the data store designed for a limited number of trusted peers with high network availability to be able to exchange change information so that they can efficiently bring themselves into mutual consistency. We used this ability by trying to extend it to an unlimited number of untrusted clients by routing connections through a custom security layer. Attempting to do this was so new and untested that we were unable to anticipate what would happen when it scaled up to a large number of clients, some of which were without network availability for long periods of time, days or weeks.

3. Due to the fact that each client is only interested in a subset of all possible changes, when a client has been offline for more than a brief period of time, fetching changes required filtering the accumulated changes of all other clients. This is completely

unscalable and led to reduced performance and unavailability of the data layer. The worst problem is that it turned out that clients with the least traffic were the most affected. The structure of the database was simply not designed for what we wanted it to do and we were left without any clear path to a solution.

What work did you perform in the tax year to overcome the scientific or technological uncertainties described above?

1. Initially we were unaware that any problem existed, because as mentioned above the more frequently an organization is tested the less affected it becomes by the problem. However unexpected data layer failures began to filter in as we grew our customer base. In order to understand how the problem manifests we had to uncover how this new data storage technology synchronizes its data between its nodes. Records in a database are stored in a heap each with a globally unique identifier. A primary index is used to be able to access a document by its id without any other information about what the record contains. Every time any record is created, modified or deleted an entry is made into a global changes log for the whole database. Each log entry is assigned a change sequence number. Clients store locally the latest sequence number that they are aware of. When the client is offline it misses realtime change events. When it comes back online it requests changes to its own data starting from its sequence number. It would be very efficient to give it all subsequent changes. However, the way we are trying to use it required us to filter the changed documents to only those involving this client's organization. The less frequently an organization produced changes, the more filtering of other organizations' data would be required. Filtering the data requires a centralized server to parse each data record into memory and apply logical filtering, this process is very inefficient and not scalable. As a result, periodically a client's request for changes would cause the data layer to timeout.

2. We considered and attempted several possible workarounds. One involved creating and updating indexes for each organization that

would offload the work of filtering to the time that a change occurred. This would also have had the advantage of not having to repeat the filtering work with each request, which is how it was working and exacerbated the problem with each additional client an organization connected. Another attempted solution was to allow our data security layer to fetch unfiltered results from the data layer and then do the filtering at the last moment before sending to the client. This did alleviate pressure on the data layer but we found that it was not a workable solution because the burden of what is effectively busy work would be shifted onto our servers. In the short term we would have been able to deal with this by increasing the total number of servers we have, at greater expense, however projecting into the future this solution would eventually hit a failure point as well as we continued to board more customers. We also experimented with radically unrelated solutions involving reproducing the change sequencing system manually. This also proved to not be workable. In the end, none of our attempts to overcome the problem while maintaining the existing data layout was successful. Each solution we came up with had some drawback that introduced its own new problems.

3. Eventually we realized that architecturally the database was not designed to be pressed into service in the way that we were trying to use it. The way we finally moved forward and resolved the problem was to redesign how our data is distributed across databases. Filtered replication allowed us segment the data between more than one database. Migrating each organization's data into its own database allows for each heap to generate its own change sequence number, and reading changes from a pre–segmented sequence does not require filtering the output on the server side. In order to make this new more scalable data arrangement possible, we had to rewrite significant portions of our data security layer to allow an authorization token to dynamically select the appropriate data heap with each request. We continue to have some dependencies on a central data store, however, because some cross organization lookups cannot be done efficiently across data heaps. We were also required to rewrite portions of our system that creates new organizations and validates

activations to programmatically create and permission new databases.

What technological advancements did you achieve as a result of the work described above?

1. The transition in architecture that we were required to undergo has allowed us to continue to scale on top of the NoSQL data storage technology that we had originally taken a risk on. The improved data arrangement has enabled numerous benefits beyond increased performance and stability. The ability to segment the data further into, for example, different geographical regions while continuing to run through the same security layer is now possible. Backups of data on a per organization basis are now very efficient, and can be run on an interleaved schedule. Updating indexes when we introduced view changes are now much faster and can also be introduced on an interleaved schedule. This has led to less downtime required when releasing major new versions of our API.

2. In the unfortunate scenario when a customer leaves our product, which can happen for instance if they go out of business, their data can now much more easily be disabled or archived, and even if parked, left in place, it no longer contributes to the overhead of performing backups, view changes or searching the data of any other organization. All of this contributes to lowering operational costs and improving customer experience.

3. There are some downsides, for instance, view structure has to be maintained across multiple databases. Databases need to be created and assigned permissions programmatically. Some information needs to be stored in a central database in order to efficiently lookup activation codes and times. And there has been a general increase in server side code complexity required to support the new arrangement and also to support the transition parts of which are still in progress.

2) Asynchronous Report introducing TPProcessManager / Overcoming Memory Constraints on iPad

What scientific or technological uncertainties did you attempt to overcome uncertainties that could not be removed using standard practice?

1. [The product] is a clever new retail point of sale system for the iPad. One of our development goals was to have advanced realtime reporting available to the iPad but without the need to query a central server to compute the reporting results. Performing report calculations on our servers is inherently unscalable and would dramatically increase the cost per customer. There are no off–the–shelf components that do what we wanted [the product] to do so we had to design our own realtime data warehousing technique. Data is summarized by revenue, cost and inventory sold or returned per user, item, color and size. The data is stored in a tree and grouped by varying units of time, current day, current month, current year. Taxes and payments are similarly summarized but with less granularity due to how they apply.

2. In order for reporting to occur on the iPad the iPad downloads, caches and continually updates warehoused units. An initial load starts with 60 day units. Without touching the server complex reports with arbitrary filtering, slicing, grouping and sorting can be accomplished all inside the iPad, as long as the date range falls within the locally stored 60 days. If the user wants to go beyond the past 60 days the system will download more chunks of warehoused data in units of days and months.

3. We encountered two major problems. [The product] is architected around web technologies and can be run directly in a web browser. The first problem is that Javascript is single threaded. One

process thread for updating the UI, processing user input and performing business logic as well as calculating report outcomes. As the number of reports grew and the complexity of reports expanded local report calculation began to encroach on usability. The second major problem is that the only standard solution for multi–threading in Javascript required a minimum doubling of our memory footprint. The iPad, especially older iPads, have very limited memory. iOS also has no support for memory paging. When our app tries to use too much memory the task is terminated. The app crashes.

What work did you perform in the tax year to overcome the scientific or technological uncertainties described above?

1. As we began to build out our reporting suite, the length of time that was required to compute the results of a report continued to grow. Because of the single threaded nature of Javascript the iPad's user interface would have to lock up while the report was being computed. We had to engineer a way to allow the iPad to compute report data without locking up the UI. Our initial attempt was to make use of a relatively new web technology that is still uncommonly used, webworkers. Webworkers allow an independent thread to execute concurrently with the main thread, it sounds like just what we need. However, webworkers were designed to prevent a type of memory corruption issue that was common in lower level operating system implementations of multiple threads. As a result, webworkers are granted no access whatsoever to the objects in the memory space of the main thread. The only way for code inside a webworker to compute reporting results is to copy large portions of main memory into the worker thread. To get just one thread we had to nearly double our total memory footprint. We completed this implementation without knowing how close we were coming to surpassing the memory limitations of the iPad.

2. Some of the customers we began boarding had larger datasets than we had originally anticipated. Instead of multiple hundreds of items, and multiple hundreds of customer accounts, they

had between ten and twenty thousand items and between eight and fifteen thousand customers. Javascript is also a garbage collected language, which is a memory reclamation system whose performance depends on having lots of spare memory. As our memory footprint got closer to the limits, the iPad performance started to drop dramatically and the app would crash at unpredictable points. There were further complications to using webworkers. The only way to communicate with a webworker is via an asynchronous message channel. But if a worker is in a tight loop computing report results a message cannot interrupt it. So, when iOS would send low memory warnings there was no way to stop a worker from doing its work, and no responsive way to clean up unnecessary memory usage. Webworkers were simply never intended to be used the way we were trying to use them and were ultimately unsuited which sent us back to square one on solving our original problem. We do still make use of webworkers for some very constrained tasks, but we had to reimagine a solution for our report engine.

3. Drawing on experience with work in operating system theory, we decided it was necessary to design and implement a primitive cooperative multi–tasking process manager. A process manager is a piece of software that is designed to schedule units of work and to interleave those units and allot to them time according to their needs. However, a process manager cannot simply be put in place and expected to work. Every report model generator had to be refactored in such a way that the complete task of producing a model could be represented by hundreds or thousands of small independently executable steps. Nested loops had to be decomposed into anonymous function calls with enclosed variables that keep track of state. Then we wrote a process object that wraps a set of steps that need to be performed and the process can be registered with the process manager. This sort of solution is very low level, and is almost completely unheard of in web applications. Rewriting significant portions of our reporting system came with a lot of risk that the whole thing might not work. Cooperative multitasking was common in older operating systems from the late 80s early 90s. Their advantage is that

they can be implemented in a very light weight manner and they share memory. The disadvantage is that one process can corrupt another process's memory, or fail to cooperate and lock the system up. They are not well suited to systems that are expected to integrate processes that are written by non–cooperative third parties. Because all the processes required to cooperate are parts of the same application it is a very good solution.

What technological advancements did you achieve as a result of the work described above?

1. An intermediate effort along the route to the full process managed solution was to start by breaking the tasks down into independently runnable chunks of code, but they were not scheduled centrally. Moving our report modeling code over to TPProcess gave us a huge, tremendous breakthrough in speed. Report model generation increased in speed somewhere between 10 and 40 times over our previous attempts. The reason is because small units of code that are not centrally organized have no way of knowing whether they are taking up too much time and locking up the UI or not using enough time to be efficient.

2. The TPProcessManager solved numerous problems and unlocked unforeseen advantages. The webworker only gave us one more thread at the cost of twice as much memory. Within that thread we could still only process one report at a time. The TPProcessManager allowed for multiple reports to be generated simultaneously without locking up the UI and without any unnecessary duplication of memory. It also gave us the ability to update the UI with report generation progress, so we were able to add progress bars to present to the user. Additional advantages included the ability to interrupt a report model's generation. If the user doesn't want to wait for the current report, but wants to change filters, the current report generation can now be cancelled mid–flight and started over with new filters. This was impossible with webworkers.

3. However, we saw even further advantages from this new system. When iOS gives us a memory warning the TPProcessManager can immediately stop giving time to inessential processes, and present a low memory warning to the user. In most cases this is enough for other iOS processes to give us some memory they don't need. If no more low memory warnings come in for 30 seconds TPProcessManager automatically resumes processing. The combination of not needing to duplicate any memory and the ability to respond to low memory warnings dramatically increased the total number of data records an iPad can handle. This opens up [the product] to support a range of higher end businesses.