Medical Imaging Clinical Workflow System

Team Lambda

Justin Field  Drew Holland  Jeff Linse
Eric Lutley    Vegas Miller

4Qimaging
Edward Schreyer, Karl Baum

Faculty Mentor
James Vallino

Project Overview

4Qimaging develops a desktop based application where users can run various algorithms, or transforms, on medical images. The output of these transforms is either quantifiable data or another image. These results can then be used to aid in diagnostics or in medical studies to determine effectiveness of various drugs.

4Qimaging wants to take this single desktop, single user model and change it to a distributed model. Hospitals and clinics would send medical images over the internet to a set of local servers hosted by 4Qimaging. These servers would hold a distributed framework in which large numbers of images could be quickly processed then returned to the end user.

The scope of this project is to deliver a proof of concept prototype which can take in images and transforms, which are then applied to those images in the form of Jobs, store information about these Jobs, and process the jobs over a distributed framework. Once the job has been processed the result will be returned to the user. The scope of this product also includes creation of a Job Description Language (JDL) that will be used as a way to get a user created Job into our system. We will then parse this JDL and create object oriented Jobs that we will use throughout the system.

After this cycle of senior project has been completed 4Qimaging will have our system deployed on their hardware. The framework will be in place as well as any extra features that we are able to fit in before the end of the second quarter. 4Qimaging will also be provided will full documentation of the product so they can expand on it as they see fit.

Basic Requirements

System Requirements

- Scalable
  - The system must allow for the addition of additional control and processing nodes on any new processing cores or physical hardware. The system should be able to grow almost indefinitely. This system should
also be able to handle an increasing amount of data without losing performance.

- **Fault Tolerant**
  - The system should not have any one single point of failure
  - If one part of the system goes down the other parts should be able to continue functioning
  - One component should not crash other components.

- **Reliable**
  - Jobs must never be lost in the system
  - Patient information must be protected in the system
  - Patient information must always be associated with the correct job. Any mix ups could lead to lawsuits.
  - The correct Jobs must always be returned to the correct sender

- **Scriptable**
  - End users will not be writing XML for the system so the Job submission XML should be able to be generated and saved into templates that the users can select from.
  - 4Qimaging should be able to specify their own templates as well that can be saved into the system.

**Functional Requirements**

- The system will accept incoming Jobs from some sort of submission utility. (For our scope a simple test prototype will be used).
- These Jobs will hold information about images to be analyzed and the transforms to be applied to these images, as well as any tasks associated with the job and any required resources.
- This information will be held in an XML file that will be parsed based on our Job Description Language. Job objects will be created by this parser.
- The system will have a “Job Entry Node” which will hold the following responsibilities:
  - Identifying each job it processes with a unique ID
  - Verify the integrity of Jobs
  - Pass image files along to the Data Store
  - Pass all job information along to a Database
  - Push Jobs onto a priority based first in first out queue in the Database
  - Notify the user of their Jobs progress though the Job Entry Node
  - Notify the Control Node if a new Job has been placed on the queue
- The system will have a “Control Node” which will hold the following responsibilities:
  - Check the queue for new Jobs whenever a current Job finishes processing. If no Job is available the Control Node shall wait until it is notified by the Job Entry Node of a new Job.
  - Take Tasks to be completed out of Job objects and assign them to worker nodes
  - Update the Database when a Job has been completed with all information about the completed Job
The system will have a “Worker Node” which will hold the following responsibilities:
  o Pulling images and transforms, that it does not have locally, from the Data Store
  o Running transforms on images
  o Updating the Database of a completed Task

The system will have a “Data Store” which will hold the following responsibilities:
  o Accept libraries, images and transforms
  o Hold images and transforms in a file system until they are pulled by a Node

Once a Job completes, the results of the completed Job will be returned to the user who sent the Job.

The system will run an internal Reporting API that can be queried for information about Jobs, Tasks, system statistics and information about Nodes.

Constraints

The system (aside from the test submission UI) needs to be deployed in a Linux environment. This is because the distributed framework we are using, SLURM (the Simple Linux Utility for Resource Management), is Linux only. There was some initial confusion on this since the sponsors originally said the entire system had to run in Windows; however after discussing it they said the framework could be purely Linux based as long as it could be accessed from a Windows based submission tool. Once this was established it was much easier to find a framework to use and development also ran easier.

Time was also a fairly major constraint for us during our implementation. Poor first quarter planning led us to start behind in our second quarter implementation. Development could have started in week seven or eight in the first quarter but instead of bumping up the schedule we started week one of the second quarter. We managed to recover quite well but still ran into issues delivering on the full extended feature set we initially documented in the SRS.

One design constraint we had was that the sponsor came to us with a very good idea of what they wanted the architecture of the system to be. They knew they wanted Control and Worker Nodes as well as a Job Entry Node. Therefore, when we were designing the system we were limited on the number of ways we could create our high level architecture since it was already laid out pretty solidly for us. This was not necessarily a bad thing since the architecture they had was solid, but it certainly limited how we were able to design the system.

Using C++ for the entire system (as per the sponsor requirements) also put a large constraint on us, mainly because the number of tools available for distributed C++ are far fewer then the number of tools for other languages, such as Java. On top of this, many of
the built in features in Java, such as reflection, were not available in C++. Luckily our sponsors suggested we use the QT library, which helped us a lot since most of the functionality it gives us makes up for the lost functionality between Java and C++.

One final constraint was the CodeBlocks IDE. The sponsor never said we had to use this IDE but it is what they used and they recommended using it. In retrospect it would have been better to choose a different IDE and develop in that.

**Development Process**

The process we chose for this project was an incremental waterfall. We set aside half the first quarter for requirements and initial documentation, and the other half for architecture design. The second quarter was broken into three overlapping development cycles of detailed design, development, testing, and integration. These cycles were planned to last four weeks, three weeks, and three weeks with the first week of one cycle overlapping with the last week of the previous cycle.

We chose this approach because while the sponsor had a very good high level view of what they wanted for the system, they were unsure on many of the specific requirements; as well as being unsure of exactly what they needed us to do, and what it would be nice for us to do if we had time. This process allowed for a long upfront period of time for requirements gathering with our sponsor. This upfront period was time well spent because we were able to nail down almost all the requirements. 4Qimaging told us that they did not expect any requirements changes either so the unchanging nature of the requirements was also a reason to devote that initial time to figuring them out.

The incremental release cycle also worked really well because we were able to deliver a limited set of functionality first then build on top of that. Building an entire framework at once or building each part in detail and hoping integration went smoothly would have been much more difficult. It also gave us a brief design phase at the beginning of each increment that allowed us to work in any new requirements or requirements tweaks the sponsor thought of.

This process was approved by the sponsor. We presented it to them with our reasoning for using the model and they agreed that it was the best approach for the system. They also thought it was a good process because there is a good chance at some point this framework will have to be approved by the FDA for use. The FDA strongly recommends a waterfall-like process in order to get approved for use quicker and easier, so our sponsors were enthusiastic about our process choice. This process choice was not mandated to us, but we feel that if we had chosen another process our sponsors may have disagreed.

Our process planned for weekly meetings with the sponsor during the requirements and architecture phase; then meetings when necessary (but at least at the end of every increment) for when we were developing. This gave us a very high level of
communication with the sponsor and we were able to work directly with them often, especially in the first two phases of the project.

In terms of team roles we initially assigned roles with the primary tasks of communicating with the sponsor, being the team leader, and the team recorder. These rolls were filled by whoever wanted to fill them. We stuck to the roles pretty strictly at the beginning of the project, but as time went on the rolls began to blend together until we were all playing each role equally or assuming others roles as teammates assumed ours. It would have been a good idea to assign a role to someone that would be the lead test manager on the system. As it was we did not stick very well to code reviews or testing procedures like we had planned. If someone on the team was dedicated to testing the system and running code reviews we probably would have done more testing and created better code.

**Project Schedule: Planned and Actual**

We developed our schedule by looking at what we had to do, then estimating how long each phase would take. We discussed each phase in detail as a team and came to a common agreement as to how long each phase of the project would take. We did this by all throwing in our estimates, getting rid of the high and low estimates, and then discussing which of the middle ones was most reasonable. We based our project schedule on this.

As outlined above we had four major phases: Requirements, architecture, implementation and deployment. Our three implementation cycles were broken down each into: Detailed design, coding, testing, feedback, integration. Our implementation cycles were also set up to overlap each other. The last week of increment one was also the first week of increment two and so on. This allowed us to never have any development downtime. The key milestones from this schedule were completion and approval of the SRS, completion and approval of the architecture, our release one demo, our release two demo, and our final deployment.

We stuck to our schedule almost exactly until the end of release two. We slipped release two by over a week trying to fix bugs and integrate our system. That was the only major schedule slip we encountered in the project. It was primarily caused by an underestimation on our part on the complexity of the Job Description Language (JDL) and the time it would take to implement it correctly. We also ran into minor bugs throughout the rest of the system that were much harder to untangle then we expected. We adjusted to this schedule change by having a team member dedicated to working on our third release while the team finished release two. When it became clear that we would not be able to finish release three fully we scrapped it altogether and moved the features we knew we could get done into a final release along with release two bug fixes. This worked out pretty well for us, but bugs continued to affect our system and our final prototype was not implemented to our satisfaction.
Overall the schedule worked well for us, but it was not planned out as perfectly as it could have been. We overestimated how long requirements and architecture would take us and that hurt us when we slipped release schedules later. If we were to redo the schedule now we would probably remove a week from both requirements and architecture work in order to have two more weeks in development time. This would have probably helped us iron out the bugs that are present in the final system.

System Design

This document outlines our high level design for the system. The biggest aspect of this system is the separation of the nodes. The Job Entry, Database, Data Store, Control and Worker Nodes all are distinct entities with distinct responsibilities. This design was chosen for two reasons. First, the feature sets each node implements are very different and very specialized in their own right. By separating them into Logical Nodes we are able to reduce coupling in the system. Second, by making each node abstract from the others (communicating only through interfaces) the system is designed so any piece can be swapped out for a new piece with little difficulty. On top of this, if any one Node in the system fails, the rest of the system can keep operating normally until that Node comes back online.
To get a bit more specific about each node, the Job Entry node is the exit and entry point of the systems. Jobs are sent to the Job Entry node by our Job Submission utility. The job reception package receives the job, and it is parsed and validated by the validate package. If we have received a valid job, the job queuer package sends a job to the database and informs the control node to expect a job. The database holds information relating to jobs, tasks, functions that can be run, and customer information. The control node is in charge of the worker nodes. It is composed of two parts, a SLURM daemon that runs SLURM jobs and the job coordinator module that sits on top of the SLURM daemon. This package is responsible for converting our system's concept of a job to SLURM jobs, and dispatching jobs from the database. The worker node runs tasks given to it by the control node. It consists of a SLURM daemon that helps it connect to the control node and a task proxy that is run as a job by the SLURM daemon. The final section, the Data Store, is used to store and load larger files such as images and libraries. The Job Entry node places these items onto the data store when valid jobs are received and the worker node accesses or modifies these items as needed.

This design also allowed for the system to be distributed as it was intended. All of these nodes are processes that can be run on the same machine if that is what the user wants to do. However each of the processes can also be distributed on different processing cores or physical hardware. By distributing the processes you greatly increase the fault tolerance of the system. The system is designed so that if one node goes down and the other nodes are distributed they can still continue functioning fully. For example if the Job Entry Node were to go down there would be no way to get Jobs into the system or return completed Jobs to the user. However any Jobs currently in the system would be able to be pulled from the queue, and processed by the other nodes and completed. The completed Jobs would then be stored in the database as they waited for the Job Entry Node to come back online.

No other major alternative architecture designs were considered for this system. 4Qimaging came in with a very strong idea how they wanted the high level architecture of the system to look like. They did not have all the nodes connected but they knew they wanted a Job flow from an Entry node, to a Control node, to a Worker node and back. They also knew we needed to use a database and Data Store. The difference between the two is the database will hold all Job information and system state information and the Data Store will be a large filesystem that will hold all the potentially very large images that will be processed by the system. We ended up using the sponsor’s architecture partly because it was forced on us and partly because it is actually a good design for this type of system.

**Process and Product Metrics**

We used the following major metrics

- Requirements Volatility
- This metric was useful in the first half of our project by letting us know how often our requirements were changing. As we nailed down more and more requirements this number dropped showing that we were having less requirements churn and could move on with the next phase of the project.

- Defect Density
  - This metric shows us if any one part of the system has an abnormally high level of defects over other parts of the system. If one part of the system has a very high amount of the defects in the system it might be a good idea to go back and look at the design for that part. Thankfully we never came across any part of the system with a disproportionally high level of defect density.

- Time and Effort
  - This was a metric we did not track well during the first half of our project, but was tracked much better during the second half of the project. This shows that most of our hours (65% of them) were logged in the second half of the project. It is likely that our second half hours would have been higher anyway but a discrepancy so large shows our poor effort at tracking this metric at the start. Because of this it is hard to tell if we really worked all that much more in the second half or not.

- Job Throughput
  - This is a relative metric. What that means is that the numbers we get from this metric are relative to the environment we are collecting them in. Since all our testing is was conducted in virtual machines we cannot take this as how the system will perform when ran on dedicated hardware. That being said this metric tells us on a high level how fast jobs are being processed by the system. The higher, the better.

- Mean Job Completion Time
  - This is a relative metric. The lower the completion time, the better. This tells us how quickly Control and Worker nodes are managing to execute Jobs.

- Mean Job Waiting Time
  - This is a relative metric. The lower the waiting time, the better. This metric will show us if Jobs are being pulled of the queue in a timely matter. If the mean waiting time is really high it shows there may be a bug in the system dealing with Control Nodes pulling Jobs off the queue.

The metrics we chose to use were useful to us. We were able to use them to see how the system was doing and how we were doing in our process. Overall the metrics were a good learning experience for how to structure and plan future projects. We did not stick to using the metrics as strongly as we could have however. No official metrics document was kept and regularly updated throughout the second half of the quarter. These metrics were gathered at various points in times but they were either never or poorly documented as well as they should have been.

Product State at Time of Delivery
The state of the project at the time of delivery is not where we would like it to be. The entire basic proof of concept prototype is in place, but we are having communication issues between the nodes. In testing, most nodes seem to work pretty well by themselves, but the overall system does not work to the extent that we would like. This problem is most likely caused by discontinuities in data representation throughout the system that has bugged the project all quarter.

Most of the planned features that have yet to be implemented are features that fell into the “if you have time do this” category. The ones that do not fall into that category are at the very least minimally implemented with plenty of hooks for easy expansion. The major missing features are:

- **Extensive Reporting API**
  - This was a feature we planned to have that reports on multiple aspects of the system, as well as system statistics and Node statistics. In the end this feature was only half implemented. Job and Task reporting are implemented as well as basic Job throughput information but all other reporting (such as information about Worker Node hardware and statistic information) is not.

- **Full Standardized System Logging**
  - We planned for a standardized logging format that the entire system would use, as well as having logs that were extensive and could be parsed for useful information. As of now there is system logging, but there is no standard format, and the messages are mostly information messages that may or may not be helpful outside of debugging.

- **Complete User Feedback**
  - We planned to have the user notified at many points throughout the course of their Job being processed. As it is now we have minimal user notification, all of it in the Job Entry Node.

- **E-mail notifications**
  - This was cut due to time constraints and fell into the “extra feature if you have time” category but it was planned to email the user at various points in their Job processing lifetime.

- **Job Pre-Emption**
  - This was cut due to time constraints and fell into the “extra feature if you have time” category but it was planned to have a special category of ‘emergency’ Job that could jump to the front of the queue, grab a Control Node if none were free, and force it to pause its Job and all tasks on all of its Worker Nodes. Once this Job finished, the Control Node would resume normal processing.

- **A full Job Description Language**
  - We have a pretty complete JDL as it is now, but as the second quarter went on the sponsor kept bringing up extra cases that needed to be included in the JDL that we had not even considered. The JDL as it is now works and works for a number of various Job cases, but it is far from all possibilities of Job cases.
All of these discrepancies had to do with time. We simply did not have enough time to fully complete these last few features. We managed to deliver working versions of all but a few features, they are just not ‘complete’, and they only offer partial functionality rather than the complete functionality the sponsor was looking for.

There were not any features that are currently implemented that were unplanned at any point in the system. The requirements were nailed down pretty solidly at the beginning of the project (aside from the full JDL specification).

**Project Reflection**

Looking back on this project we are proud of what we did. We fell behind a bit at the end of the first quarter but the entire team pulled together and put in a ton of effort the second quarter to deliver. Going through the SRS now, all the requirements the sponsor outlined as “must have” or “should have” are either fully functional or mostly functional (working just not feature rich). We have come a long way in a short period of time. However the final deliverable to the sponsor is still having minor integration issues and is not at a point where we are happy with it.

Examining the project in detail there are a few things we did really well. The first thing we did well was pick a process that suited the system well and stuck to it for the duration of the project. This helped us throughout the entire project and was a good choice of process. We also had a well documented SRS. While there were a few use case scenarios that were not brought to our attention until later in the project we managed to get down all the core requirements and document them well. Another thing we did well was our detailed design. These designs for each part of the system were in depth and clearly represented how to implement the system. It made coding much easier. A final thing we did well was pick the right tools for the job. SLURM ended up being just what we needed (even if it was not as simple as its name implies). QT was a life saver as many of our nodes needed the additional functionality it offered in order to be completed.

We also did a few things poorly. We had two areas of fault that were the root cause of all our other problems. First by not documenting all our time and effort during the first quarter we caused a potential useful metric to be much harder to analyze. The second area was something that has been mentioned a few times already. We overestimated how long our requirements and architecture phases would take. This turned out to be a major error. It caused us to be behind by almost two weeks from where we could have been if we had tightened our schedule. If we had managed to get those two weeks of development time we probably would have been able to deliver a complete feature set to the sponsor, instead of a half-finished one. One other item that went wrong, not on the scale of the previous two examples, was our failure to fully outline the JDL before we began implementing it. This led to the situations when the sponsor brought up items that needed to be included in the JDL; not only was it difficult to add them, we did not have enough time to implement them all.
There were also quite a few issues in the actual implementation of the system. The team on a whole seemed to degenerate back into a code and fix model during implementation. Furthermore, some of the code does not utilize good coding principles and can be a bit more complicated than desired. This has led to some “hydra” like activities were fixing a bug causes more to change. Furthermore, because the JDL, job database, and the information objects are all very tightly coupled, changing the way one of these modules worked often required changes in the other two modules. This may have been a design issue, but there’s not really a very good way to work around it.

On top of this code reviews were never held and no official testing was ever done. All team members individually tested their own code but this usually involved just writing testing classes rather than using a unit testing framework. Integration testing was spotty at best as well. We feel team members cared about testing, but either did not have enough time or motivation to actually go through the process of selecting an official testing framework and designing and writing unit, integration, acceptance and regression tests.

In the future, for software projects we would all more carefully estimate how long items will take and plan accordingly. We also will not be scared to bump up our schedule if we are finishing items early. It is also a good idea to have one team member dedicated to the testing of the system and to schedule and enforce the team to regular code review meetings. Finally, we would make sure we fully understand what a customer wants from a product before we begin implementing it so we do not run into problems down the line.

Like it was said earlier we believe the project was an overall success, if somewhat of a technical failure. Despite the setbacks and the non-ideal final product we were still able to build a very complex system from scratch and deliver an almost full feature set on top of a prototype framework. It was a lot of work and as the second quarter progressed, team members looked more and more tired and burned out but we never gave up and kept working to the end unwilling to throw in the towel or hack together features.

This project was also a huge learning experience for us all. We all learned a great deal about project management and developing a product start to finish. For all of us this was the first time we were directly in charge of a brand new project of this scale. Taking it through all the phases of the Software Development process we learned throughout our time at RIT. Setting our own deadlines and being in charge of a product was at first difficult to grasp since we were all used to being told “you have to do this by then” instead of “you decide what to do and by when”. Now at the end of the project we all have a much better understand of project flow and control but it was still very difficult to say “no” to the sponsor. We theorize that because this project took place in an academic setting we viewed it more of a class then as a project to be completed (more like a co-op). Because of this mentality it was a subconscious decision to say “yes” to the sponsor and potentially overscope some. We are not sure if there is a clear way to fix this problem but it is something to think about when starting the project.
Besides the process aspect we all learned a great deal on a technology level. Learning about QT alone taught us a valuable C++ library to use in the future. Besides this learning how to test and deploy a distributed system (even if they were just on VMs) was something that was new to a few of us and was another valuable learning experience to all.

In the end we are glad we picked this senior project. We worked with great people to deliver an interesting and complex project. Along the way we learned many project, process and technical techniques that we would not have learned otherwise. On top of that we gained invaluable experience that could not have been learned any other way. It was a fun ride, we are glad we were able to have the experience, and we are proud of our product as a whole.