GONZAGA UNIVERSITY
School of Engineering

STATUS REPORT
for a
Power Simulator Clustered Development
Software Engineering Senior Group Design Project
GU:SESGD 491-02

Prepared By:

__________________________________________  __________________________
Justin Funston                                                                                          Jay Kudo

__________________________________________
Michael Lavrisha
Objective

The **PowerData PowerSimulator** is a program developed for Power Operators. These people need to keep the power systems running at such companies as Avista. To prepare for situations ranging from minor to extreme, they use these simulators. This PowerSimulator has a database at its core and a graphical interface through a web browser in which the Operators interact. Our objective was to take the current program which is confined to a single computer and make it able to handle a multitude of nodes on a cluster.

Requirement Addressed

Finish making the PowerData shared memory database run as a fully distributed process. This means that the PowerData database should be able to run on any node and be able to communicate with each other node for concurrency purposes. The ability for replication of the database is a necessary requirement for reducing the strain on the system, because a large portion of the work is done in the querying process. This query requires much more work to be done than the transfer of the database at the beginning, so to have one node running the database would not produce the maximum effect. For concurrency issues, when the database is being updated, if they are close enough in time for race conditions then it is arbitrary who wins. The most important thing to maintain for this implementation is the assurance that the same process wins on every node.

Scope

This project required the members of the group to first create a requirements document. This document outlined the intended conclusion of our work. After doing revisions and confirming the document with our liason, it was necessary to outline our design ideas. The group then came together to compare and contrast the different ideas. From this, a general design was created. During the early implementation stages, it required the members to figure out how the system was going to be built, including the variety of algorithms and data structures. From there, the coding began and the work was divided so each member worked on a specific component. Upon completion, unit testing took place to confirm that each part was working as planned. Finally, the components were combined beginning integration testing. The final stage is to document all of our work and prepare the deliverables for the handoff to the liason.

Schedule

Our Gantt Chart was as follows:
In terms of our schedule, we ended up being a little behind in our current track, but we should be able to get back on track over the next week and a half. We currently remain in the testing phase of our design, we should have ample time to correct any errors that we may find. Most of the bugs should have been taken care of during unit testing, so our integration testing shouldn't prove to be too troublesome.

**Overall Implementation and Design Issues**

The entire project consisted of the implementation of the parallelization of a real-time column-oriented database on the in-house Linux cluster. Since the database was already in place, as described earlier, our team worked to augment this core database with the capability of running it in parallel. This backend to the database, the so-called Distributed Power Data Server (dpdserver), needed to circumvent a variety of different issues.

The first issue we needed to deal with was the necessity of the database to be replicated to every node. Any user should be able to connect to any of the plethora of nodes and have the same experience as if they were all on the same server. Obviously, this means that users are changing completely different aspects of the databases. Therefore, it was necessary for each of the databases to remain concurrent, otherwise, each database would be holding completely distinct data. To maintain concurrency, it was decided that the message passing paradigm was the best implementation. This fulfilled our original requirement.

The second issue we attempted to cope with was the performance issue. The group tried to increase performance, since the message passing would lead to a degradation of performance time, as compared to just directly manipulating the memory as one might sequentially. As is inherent in all databases, one of the major performance overheads is in the expensiveness of writes versus reads. Since it is common practice to do thousands of updates within this database, we tried to reduce the number of writes necessary. So, after waiting a short period of time, currently set at one second, we collect all the updates collected through message passing and filter them. Therefore, any updates which came at a later time to the same data as an earlier event could supersede these earlier events and therefore require one write to that location instead of many. It might not seem that this would occur, but since the database is a back end to a power line simulator, the data at any point could be constantly changing. These challenges in engineering led to this design:
In this design it is easily apparent that an endless loop is created to handle the concurrency. The team split this loop into three distinct tasks and we shall discuss each, in detail, in the following sections.
Message Passing Implementation

In the decision on ways to maintain concurrency, it was obvious that message passing was the necessary route for our system. This was in part necessitated by the design of the Linux cluster, but it gave us a further advantage. Since it is likely that the PowerData Corporation will not want to be tethered to Gonzaga’s Linux cluster, it gave them the flexibility to implement the same parallelism on basically any system, as is inherent in the message passing paradigm. It was then necessary to make a decision on the form of message passing. As a team, we were better suited to use the Message Passing Interface through experience gained in our curriculum. Nonetheless, MPI gave the overhead of using the Transmission Control Protocol (TCP). This meant that increased amount of time would occur in doing substantial amounts of error checking through acknowledgements. Instead, our liaison requested that we use sockets, specifically in the form of an IP Multicast, which can be run over the User Datagram Protocol (UDP). This decision was made for a variety of reasons, especially due to wanting less library dependencies and easier installation. Further, it meant a tradeoff of reliability for performance, a must in a real-time database. Our reliability was therefore pushed onto the application to attempt to handle this problem. This also led to a requirement to learn the technology of sockets which led to some difficulties for the implementers, especially in handling the hardware to accept this protocol.

Our program called for a few different tasks handled by message passing. The first task is to multicast out a heartbeat to every database. This heartbeat keeps track of the timing of all the databases and is able to maintain that everyone is still up and running. The second task is to multicast the events themselves created by the local users at the database level. This required the implementer to serialize the events and send it out to every database. Finally, for replication purposes, a specific socket was set as the location to send a copy of the data from the master node to a fledgling node.

Event Caching and Merging Implementation

Through the multicast, a plethora of messages are being sent throughout the system. At the endpoint, this part of the dpds server does the handling of those messages. The first part of this is the Event Cache. The receiver calls a function within this Event Cache to store the event. This Event Cache handles a few different tasks, basically gathering the events, filtering the events, and then deallocating the events. After a variety of events are stored, the second rule goes into place, as referenced earlier, and every second the data is filtered into a merged update, replacing the information which is already obsolete. The Event Cache works by taking in events and sorting them instantly. Then, it finds the next available event that has not been cached and uses that as the comparator to find all similar events. If an event arrives out of place, as discovered by being less than the current second, everything must be considered invalidated and each event must be once again searched. This expensive, but is not supposed to occur very often. After all this, the merged events then gets stored into the update queue for application to the database and for echo checking as detailed in the next section.

In this part of the design it was necessary to think about the data structures that would store the events. Using the principles of Object Oriented Programming, we created an Event Item which has content common to all events. Then, each of the specific events inherited from this base class, including the Update Events, the Insert Events, and the Delete Events. There was also the Merged Update which was the container for the filtered Update Events. This was called by
the Event Cache and then stored within the Update Queue. Besides the actual containers for
events, we relied on the Standard Template Library for implementing the storage of the event
data structures. The Event Cache used the Set STL container for its ability to sort on the different
timestamps. This meant that the sorting occurred inherently and also allows for the ability to
incrementally search as opposed to some other STL containers using iterators. The Update Queue
was implemented using a Map STL container, mapped on a unique identifier for a key and a
Merged Update for a value. This allows for extremely fast search with its hash table design,
which is important for the echo check. Using the STL meant good performance as well since it is
optimized.

The final part of this component was the actual application to the database. This was a
rather complex methodology eased by code from our liaison. It required that we know the offsets
of wherever the keys are and the location of the data to be stored within the database. These
offsets lead to speed increases but can be complex. So, each of the key-value elements within the
Merged Update must be applied individually, by passing it through a function to discover the
offset. It also means that the events from the database only hold one key-value pair, an issue
important in the next section.

Event Marshalling and Echo Check Implementation

This final part of the loop receives events from the database. The first challenge came in
actually marshaling the data back from the database. As mentioned earlier, each key-value pair
has its own individual event. Therefore, to fit within our model of Update Events which hold
large quantities of key-value pairs per column, it meant that the implementer had to marshal the
data back into the Update Event. This was eased by the fact that if all the events occurred during
one call, there is a more tag on the event which tells the user that more events are coming.
Therefore, it was possible to use that to figure out whether to continue marshaling or to finish.

The next task required was to take these marshaled events and check them against the
Update Queue. This is necessary because events can come both from the local users and in
response to the update queue updating the database. If the local user is the one who created the
event, we need to multicast that out to the other databases. Otherwise, we obviously don’t want
to send out the same updates over and over again over multicast. This would increase already
relatively heavy network traffic and wreak havoc on the ability for the Event Cache to maintain
the concurrency in relation to time. To prevent this, the Echo Check queue compares the unique
identifier, which the Update Queue is hashed on, and then checks the table and column before
moving onto the data. There is a lot of overhead, but necessary to maintain reliability. Further,
most of the time, the comparison will fail on one of the former comparisons. This will occur
before moving onto the key-value pair, meaning we only do that check if they are the same.

Replication of Database

A side challenge required the members of the group to attempt to replicate the database.
This was not too much of a challenge, the master node was designed to maintain a list of the
available nodes and each node has a specific port in which to send the data over. The master node
bundles all of its data up in a file, using a function inherent in the database. It is then ported using
sockets to the initializing database. The event queue begins collecting events from the database
and applies it once the database is fully functional. To get it to the fully functional stage a script
is started which starts the database and the function is called to load the database with the marshaled information.

Challenges

Most of the challenges were discussed within the earlier sections on the implementations. Nonetheless, a few of the major challenges were not mentioned. The biggest challenge was dealing with the legacy database, which contained hundreds of files to make it work. First created in the mid-90s, this custom database was not meant to be used by anyone external of the company. Therefore, there was little to no documentation on how to query the database, let alone its more inner workings. Fortunately, our liaison was extremely helpful in getting us whatever we needed. Nonetheless, any issue that occurred, we needed to go straight to our liaison to resolve since there is no information on it on the Internet or otherwise, which meant that working was more erratic. Further, the usual time constraints of maintaining a full class load, along with the usual setbacks, required us to reduce the scope of our project in regards to load balancing, which originally was designed if time permits.

Current Status

With a little over a week remaining within the time frame, we are nearing completion of the project. We have already tested each of our components separately using tools like a text-based database, instead of the full scale option, as well as unit testing techniques. We have now joined all of the parts together and are in the process of running tests to make sure that everything blends well together. We also plan on running stress tests with large scales of data. The only other tasks left is to finish writing the documentation, which we have worked on throughout.

Conclusion

This project has been a very interesting and challenging assignment. The original scope was rather large, but we managed to pull it in to a manageable scale and contribute useful design and software to our sponsor. The deliverables will be our code, properly commented and formatted according to PowerData standards, as well as documentation on its usage. Further research or projects relevant to this might encompass attempting to optimize different aspects of our design, automating all of the database creation including an election algorithm if the master goes down, or creating the load balancing that can handle varying user work and performance issues as necessary.