Outline

- Introduction
  - Surface MSM overview
- Getting Started
- Surface Microseismic Monitoring (MSM) Architecture
  - Web server and dashboard application
  - RESTful file service
  - IoT for seismic
  - HPC using Rocket
- Additional remarks on AQP preparedness
- Conclusions

Schlumberger

Introduction

- In last decade or so, trend has been away from desktop to devices, proliferation of video/audio/text/image
- Couple this with the explosion of connected devices/internet/bandwidth improvements/human mobility -> back to the mainframe/client model (which was eclipsed by the desktop in the 70s onward), now we call the mainframe 'cloud', consists of remote, *networked computers*
- Vendors have provided cloud offerings for many years, **everything is a service** eg., **I**nfrastructure, **D**atabases, **S**oftware, **P**latforms and more (ala the terminology "x **as a S**ervice")

Schlumberger

Introduction

- For example, in Amazon Web Services (AWS) language:
  - Storage/file service (IaaS) -> S3 (simple storage service)
  - NoSQL Database (DBaaS) -> dynamoDB
  - Compute (Iaas/Saas) -> Elastic Cloud Compute (EC2), Lambda
  - Web App Hosting (PaaS) -> Elastic Beanstalk

Introduction

- **Realtime surface microseismic monitoring (MSM) has been moved to a services/ cloud architecture,** but still need to support desktop applications, legacy software, possibly field/processing truck installations without connection to cloud

- Need several platforms and languages to support all:
  - C/C++ for numerically intensive work, node + mongo DB for web application backend, python and bash scripting for plumbing, angular for web UI, python for desktop UI

Schlumberger

Surface MSM Overview

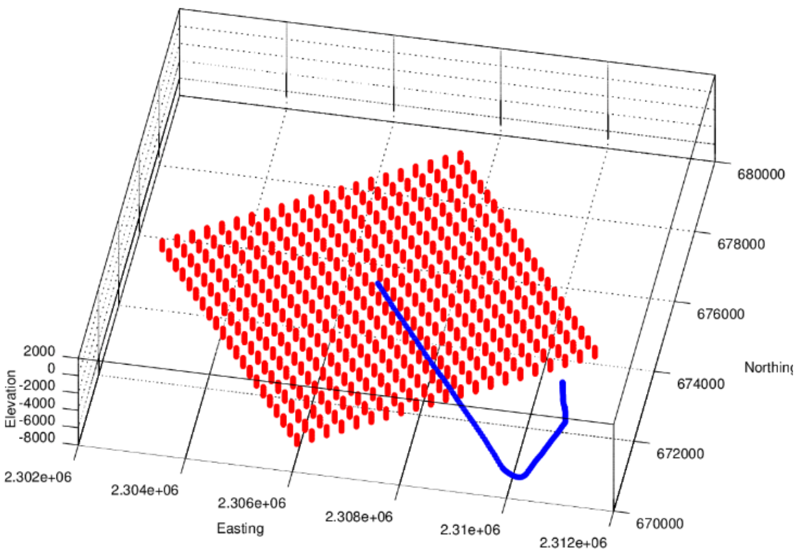
- SurfaceMSM has at least 4 major computational steps:

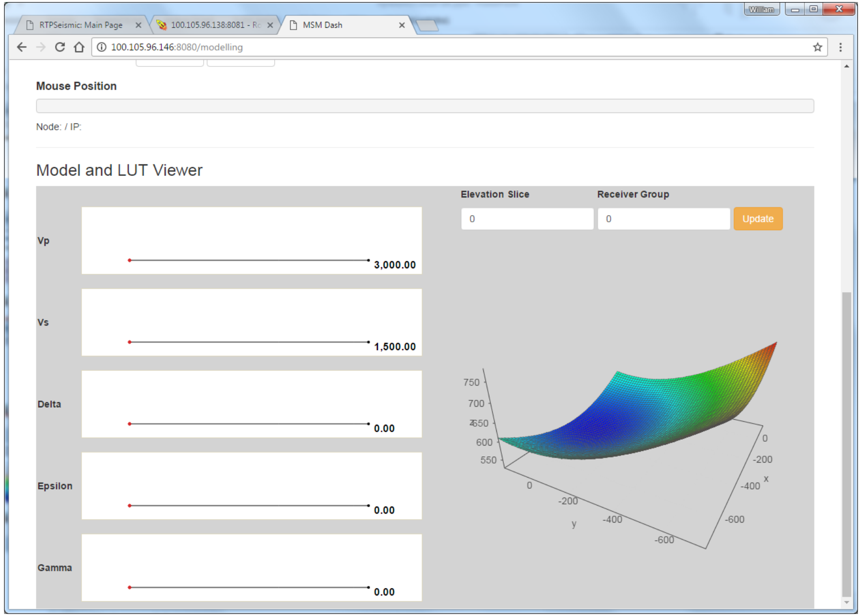
  1. Modeling/LUT generation
    - Coarse grained for initial detection/location
    - (Re) calculated for every re-processed/refined event
    - Velocity model may go through various stages of refinement too eg., using perf shots
    - May also create statics to account for elevation differences between receivers

Schlumberger

Surface MSM Step 1: Modeling / LUT generation

- Model a region around well (left image, blue line) using 1D-layered velocity model
- Perform raytracing to produce a lookup table for subsequent migration





Schlumberger

Surface MSM Overview

2. Signal Acquisition + DSP steps

- Stacking, filtering etc

3. Migration

- Uses LUT

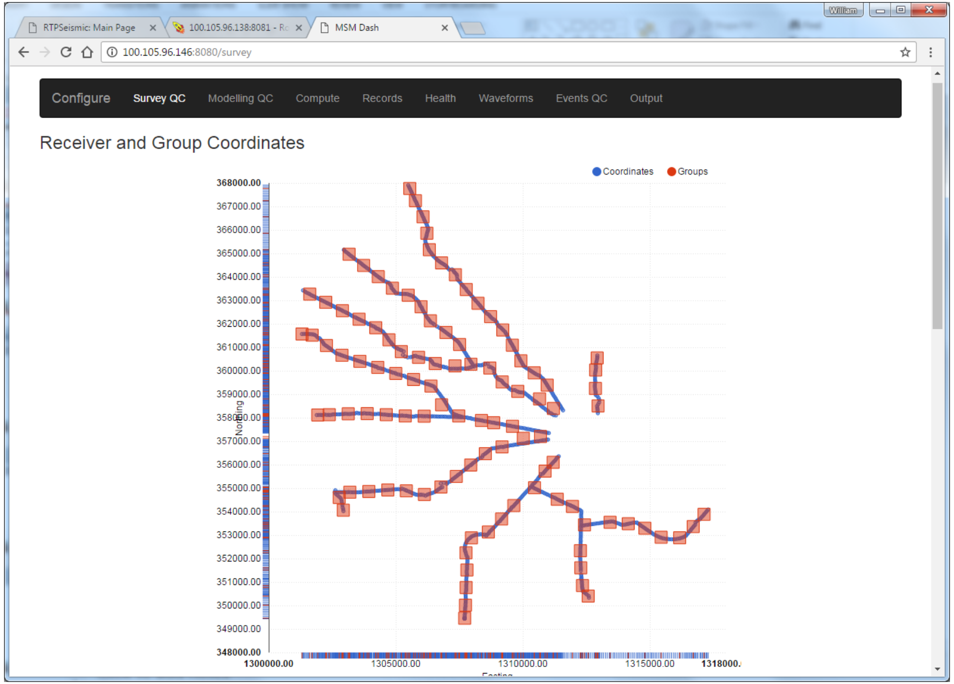
4. Detection + Location

- STA/LTA on migration volume, automatic threshold/pick, smooth and fit maxima

- As implied, these major steps are completed in at least two passes, one coarse grained and one in refinement after QC

Surface MSM Step 2: Acquisition + DSP

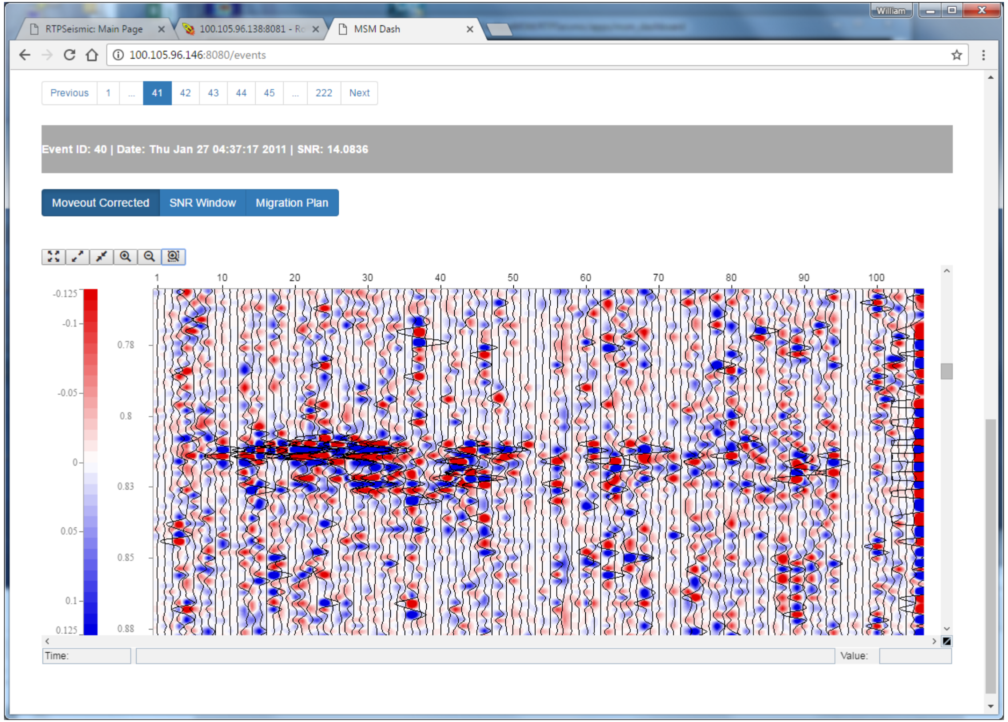
- Passively acquire data from receivers arranged in lines, patches or some combination, typically > 1K channels (blue circles: locations), traces are grouped after moveout correction (red squares)



Schlumberger

Surface MSM Step 2: Acquisition + DSP

- Events are usually very small in magnitude, DSP steps like filtering, stacking etc applied to enhance signal

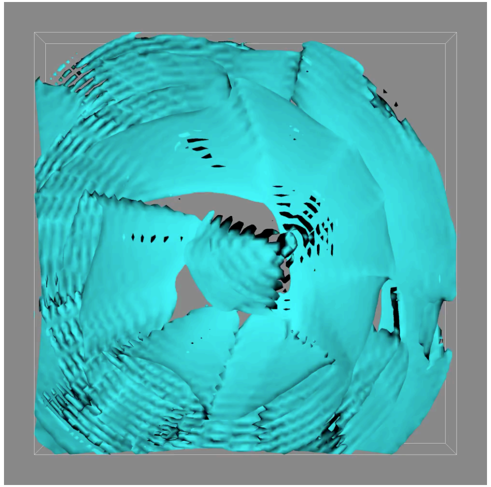


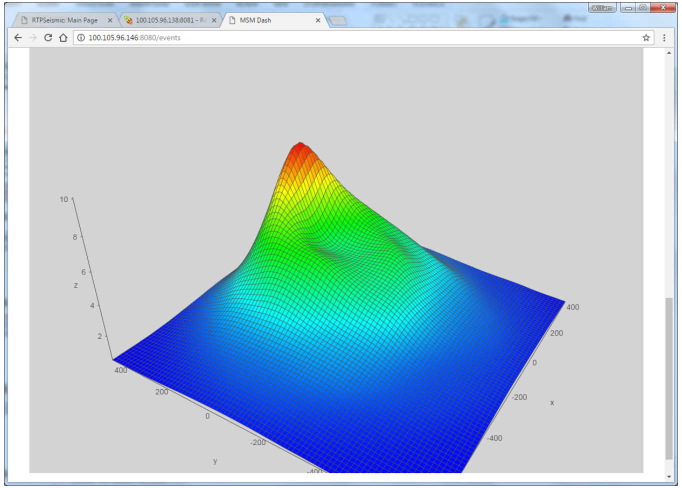
Schlumberger

Surface MSM Step 3: Migration

- Requires at least 10-20GB and 4 threads for 200 trace groups, 10s second of record (1s to process 1s of data for realtime), requiring a powerful laptop or more likely HPC (to be discussed)

- Left: evolving migration isosurface up to t0, right: slice through the migration volume at t0

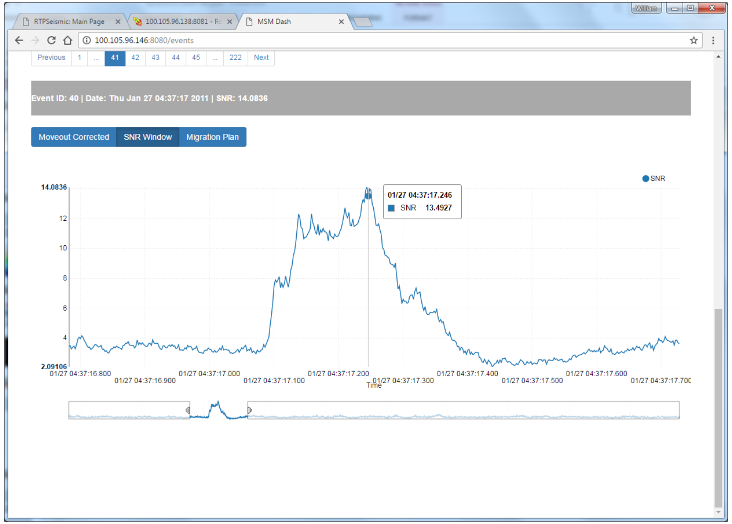


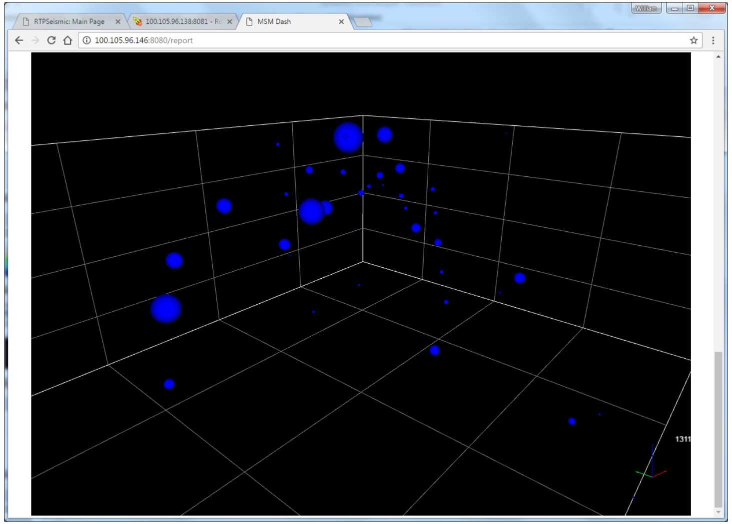


Schlumberger

Surface MSM Step 4: Detect + Locate

- We apply STA/LTA to migration volume at every time step, look for maxima in profile (left image), threshold to find potential events.
- Those that are suitably QC'd (eg., correct moveout profile, migration image) are fit for location *x,y,z* and display/reporting (right)





Schlumberger

Outline

- Introduction
  - Surface MSM overview
- Getting Started
- Surface Microseismic Monitoring (MSM) Architecture
  - Web server and dashboard application
  - RESTful file service
  - IoT for seismic
  - HPC using Rocket
- Additional remarks on AQP preparedness
- Conclusions

Schlumberger

Getting Started

- In moving to cloud architecture, obviously begin by laying out the overall workflow, understand the problem
- Helps to think in terms of basic linux terms/components;
  - What are the processes and files?
    - Where will they run or be stored?
  - How will parameters be captured and stored?
  - How will components communicate?

Getting Started

- Communication
- Processes:
  - Light (clients, small memory, short running ~ 1s), run locally on virtual machine
  - Heavy (large memory, threaded, longer running >> 1s), run using rocket HPC
- Parameters/non-binary data:
  - Input via web application / dashboard
  - Storage in mongo DB
- Binary data:
  - RESTful file service for input seismic
  - Upload/processing also via IoT nodes

Schlumberger

Getting Started

- Generally communication should use websockets or REST
- Representational State Transfer (REST) systems communicate over Hypertext Transfer Protocol using verbs eg., GET, POST, PUT, DELETE
- Systems are addressed using Uniform Resource Identifiers (URIs)
- The data type for information exchanges is frequently JSON or XML

- **Processes will invariably need to run on Linux instances**

Schlumberger

Getting Started

- All C/C++ compiled into single library, librtpSeismic.so, alongside dependencies, suited to CentOS 7

```
bash-4.2$ ls Deliverables/DragonFly/.lib

libboost_atomic.so
libboost_atomic.so.1.61.0
libboost_date_time.so
libboost_date_time.so.1.61.0
libboost_filesystem.so
libboost_filesystem.so.1.61.0
libboost_program_options.so
libboost_program_options.so.1.61.0
libboost_python.so
libboost_python.so.1.61.0
libboost_regex.so
libboost_regex.so.1.61.0
libboost_serialization.so
libboost_serialization.so.1.61.0
libboost_system.so
libboost_system.so.1.61.0
libboost_thread.so
libboost_thread.so.1.61.0
libboost_wserialization.so
libboost_wserialization.so.1.61.0
libfftw3f.so
libfftw3f.so.3
libfftw3f.so.3.4.4
libfftw3.so
libfftw3.so.3
libfftw3.so.3.4.4
libhdf5_cpp.so
libhdf5_cpp.so.12
libhdf5_cpp.so.12.0.0
libhdf5.so
libhdf5.so.10
libhdf5.so.10.2.0
libmseed.so
libmseed.so.2
libmseed.so.2.13
librtpSeismic.so
```

Schlumberger

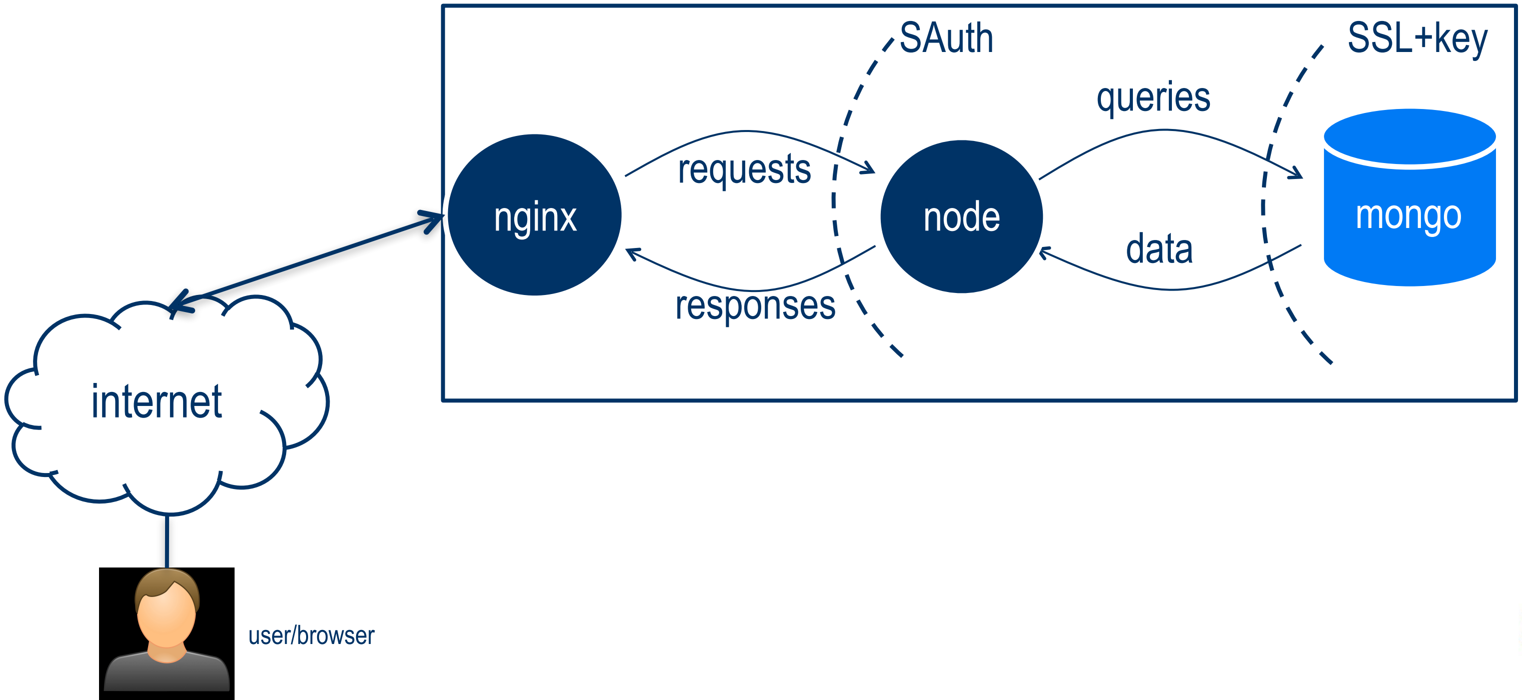
Outline

- Introduction
- Getting Started
- Surface Microseismic Monitoring (MSM) Architecture
    - Web server and dashboard application
    - RESTful file service
    - IoT for seismic
    - HPC using Rocket
- Additional remarks on AQP preparedness
    - DELFI requirements
- Conclusions

Schlumberger

MSM Dashboard

- A single page application (SPA) served using MEAN; nginx provides gateway



Schlumberger

MSM Dashboard

- **MEAN is:**
  - **Mongo DB**
    - C++ implementation of NoSQL, a high-perf, scalable database technology.
    - Wrapped by Mongoose.js in this case
  - **Express.js**
    - Web development framework suited to Node.js
  - **Angular.js**
    - Dynamic frontends using javascript (Angular 1) or typescript (Angular 2)
  - **Node.js**
    - Javascript runtime engine built on Google Chrome V8 engine (backend)

MSM Dashboard

- Design of dashboard UI draws strongly from the Single Page App design and philosophy:
  - Provide experience similar to desktop app
    - Initially load all static content (*html, *css, *js), update views as user interacts with UI
    - Angular JS (client side, runs in browser):
      - communication with web server, retrieving data through API calls
      - bindings between data model(s) and the view(s)
    - Third parties for widgets eg., INT geotoolkit for seismic views, Angular UI

MSM Dashboard

- **Data API based on CRUD -> Create, Retrieve, Update, Delete docs from DB**
- Configuration (*project schema*)
    - survey, processing parameters etc
- Input seismic data (*record schema*)
    - health and record meta data from edge nodes, if applicable
    - start times, size etc
- Event data generated by computation (*event schema*)

Project schema encode how input record schema are transformed into event schema

Schlumberger

MSM Dashboard

- Data API methods represent about 50% of the operations handled by node + express backend

- **Additionally node.js is used to control many (light) asynchronous processes at once**
  - Light processes should eventually take advantage of serverless computing eg., cloud functions, AWS lambda
  - Heavy computations offloaded to Rocket via thin client(s)/light processes

MSM Dashboard

- For example, launching a process with node + express, in response to GET request from dashboard (code extract from server):

```
router.route('/msm_service/stop')

    .get(function(req, res) {

        if (req.query.project != undefined) {

            const pid = uploadPath + "/" + req.query.project + "/msmservice_pid";
            const comm = uploadPath + '/../bash/kill_msmservice.sh ' + pid;

            exec(comm, function(error, stdout, stderr) {

                if (error != null) {
                    console.log('exec error: ' + error);
                }
…
            });
```

Schlumberger

MSM Dashboard

- Example processes attached to node process tree
  - Client for communicating hooke job to/from rocket HPC

```
bill@bill-VirtualBox: ~/Documents/PassiveSeismic/SurfaceMSM/RTPSeismic/apps/msm_dashboard
bill@bill-VirtualBox:~/Documents/PassiveSeismic/SurfaceMSM/RTPSeismic/apps/msm_dashboard$ ls bash
export_hooke_records.sh   hooke_report.sh         msm_service_report.sh        setup_ssh_keys.sh
export_hooke_times.sh     hooke.sh                msm_service.sh               unpack_raw_data.sh
export_log_times.sh       kill_hooke.sh           preprocess_local_report.sh   update_hooke.sh
export_shifts.sh          kill_msmservice.sh      preprocess_remote_report.sh  update_remote.sh
fetch_wrapper.sh          kill_preproc_remote.sh  preprocess_remote.sh         update.sh
hooke_report_remote.sh    kill_preproc.sh         preprocess.sh
bill@bill-VirtualBox:~/Documents/PassiveSeismic/SurfaceMSM/RTPSeismic/apps/msm_dashboard$ ps ef
  PID TTY      STAT   TIME COMMAND
 5309 pts/18   Ss     0:00 bash XDG_SEAT=seat0 XDG_SESSION_ID=c2 LD_LIBRARY_PATH=/home/bill/torch/install/lib:
16798 pts/18   R+     0:00  \_ ps ef XDG_VTNR=7 XDG_SESSION_ID=c2 XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/bi
 5054 pts/17   Ss     0:00 bash XDG_SEAT=seat0 XDG_SESSION_ID=c2 LD_LIBRARY_PATH=/home/bill/torch/install/lib:
11309 pts/17   Sl+    5:41  \_ recordmydesktop XDG_VTNR=7 XDG_SESSION_ID=c2 XDG_GREETER_DATA_DIR=/var/lib/light
 2943 pts/5    Ss     0:00 bash XDG_SEAT=seat0 XDG_SESSION_ID=c2 LD_LIBRARY_PATH=/home/bill/torch/install/lib:
15085 pts/5    Sl+    0:01  \_ nodejs server.js XDG_VTNR=7 XDG_SESSION_ID=c2 XDG_GREETER_DATA_DIR=/var/lib/ligh
15189 pts/5    S+     0:00      \_ /bin/sh -c /home/bill/Documents/PassiveSeismic/SurfaceMSM/RTPSeismic/apps/ms
15190 pts/5    S+     0:00      |   \_ /bin/bash /home/bill/Documents/PassiveSeismic/SurfaceMSM/RTPSeismic/apps
15196 pts/5    S+     0:00      |       \_ tail -20 XDG_VTNR=7 XDG_SESSION_ID=c2 CLUTTER_IM_MODULE=xim XDG_GREE
16646 pts/5    S+     0:00      \_ /bin/sh -c /home/bill/Documents/PassiveSeismic/SurfaceMSM/RTPSeismic/apps/ms
16647 pts/5    S+     0:00          \_ /bin/bash /home/bill/Documents/PassiveSeismic/SurfaceMSM/RTPSeismic/apps
16652 pts/5    Sl+    0:00              \_ hookeRocketEngine.x --ip 137.144.100.88 XDG_VTNR=7 XDG_SESSION_ID=c2
bill@bill-VirtualBox:~/Documents/PassiveSeismic/SurfaceMSM/RTPSeismic/apps/msm_dashboard$
```

Schlumberger

MSM Dashboard

- Example: C++ client controlled by node GETs data from file service

```
std::string request = "GET " + m_route;
        request+="?ID="+m_job_id;
        request+="&project="+m_project_name;
        request+="&time="+time;
        request+=" HTTP/1.1\r\nHost: "+m_file_server_ip+"\r\n";
        …

        boost::asio::io_service io_service;
        boost::asio::ip::tcp::resolver resolver(io_service);
        boost::asio::ip::tcp::resolver::query query(m_file_server_ip.c_str(), "443");
        boost::asio::ip::tcp::resolver::iterator iterator = resolver.resolve(query);

        boost::asio::ssl::context ctx(boost::asio::ssl::context::sslv23);
        ctx.load_verify_file("/etc/ssl/certs/apache.pem");
        rtpSeismic::acquisition::ssl_client c(io_service, ctx, iterator,request,ln);
        io_service.run();
```

Schlumberger

Outline

- Introduction
- Getting Started
- Surface Microseismic Monitoring (MSM) Architecture
  - Web server and dashboard application
  - RESTful file service
  - IoT for seismic
  - HPC using Rocket
- Additional remarks on AQP preparedness
  - DELFI requirements
- Conclusions

Schlumberger

RESTful File Service

- Directly inspired by AWS Simple Storage Service (S3), uses:
  - Linux,Apache,MongoDB, PHP
- Record JSON documents are generated by the file service when:
  - **Legacy data is exported to service**
    - File service will run pre-processing for legacy data, in response to GET requests via dashboard shown previously, reducing bandwidth
  - **Data is POSTed from IoT nodes using cellular modems on SLB network, discussed next**
    - multi-part form data + compression + https

Schlumberger

RESTful File Service

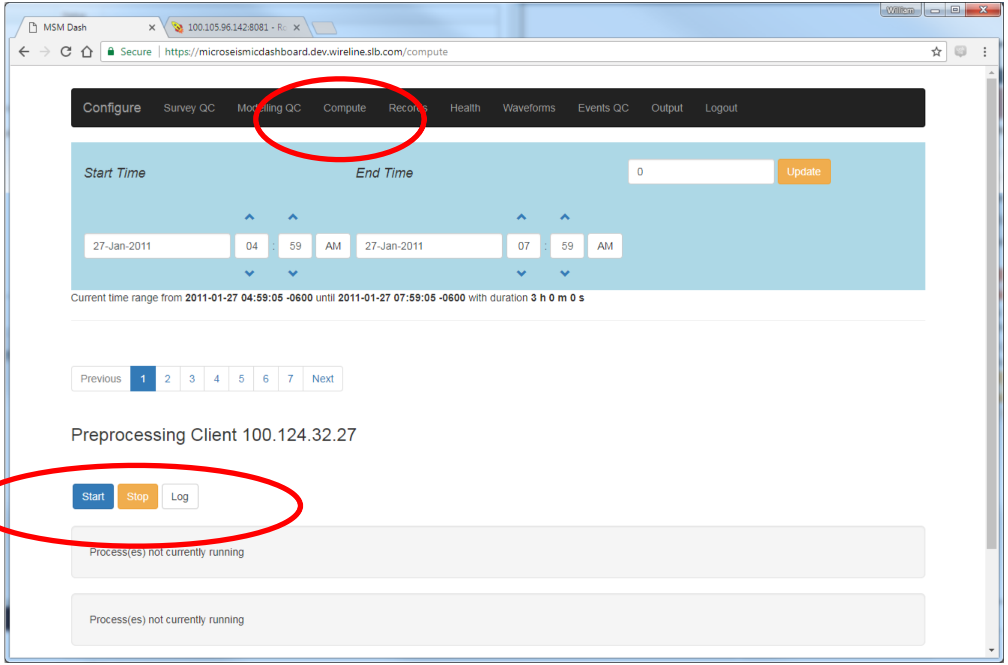
- Example **record** JSON document generated by service on received of form + payload from IoT node:

```
"jobID":"123",
"time":"1470219450",                     // when the data was acquired
"client":"100.124.32.3",                 // who sent it
"modified":"1470219491",                 // when it was received
"diff":"41",                             // latency
….
"file":"1470219450_100.124.32.3.tar.gz",
…
"ch0":"24",                              // number of channels
"h0":"30648",                            // humidity
lon0":"950080351",                       // longitude
"lat0":"293759108",                      // latitude
…
nn1":"12",                               // nearest receiver index
```

Schlumberger

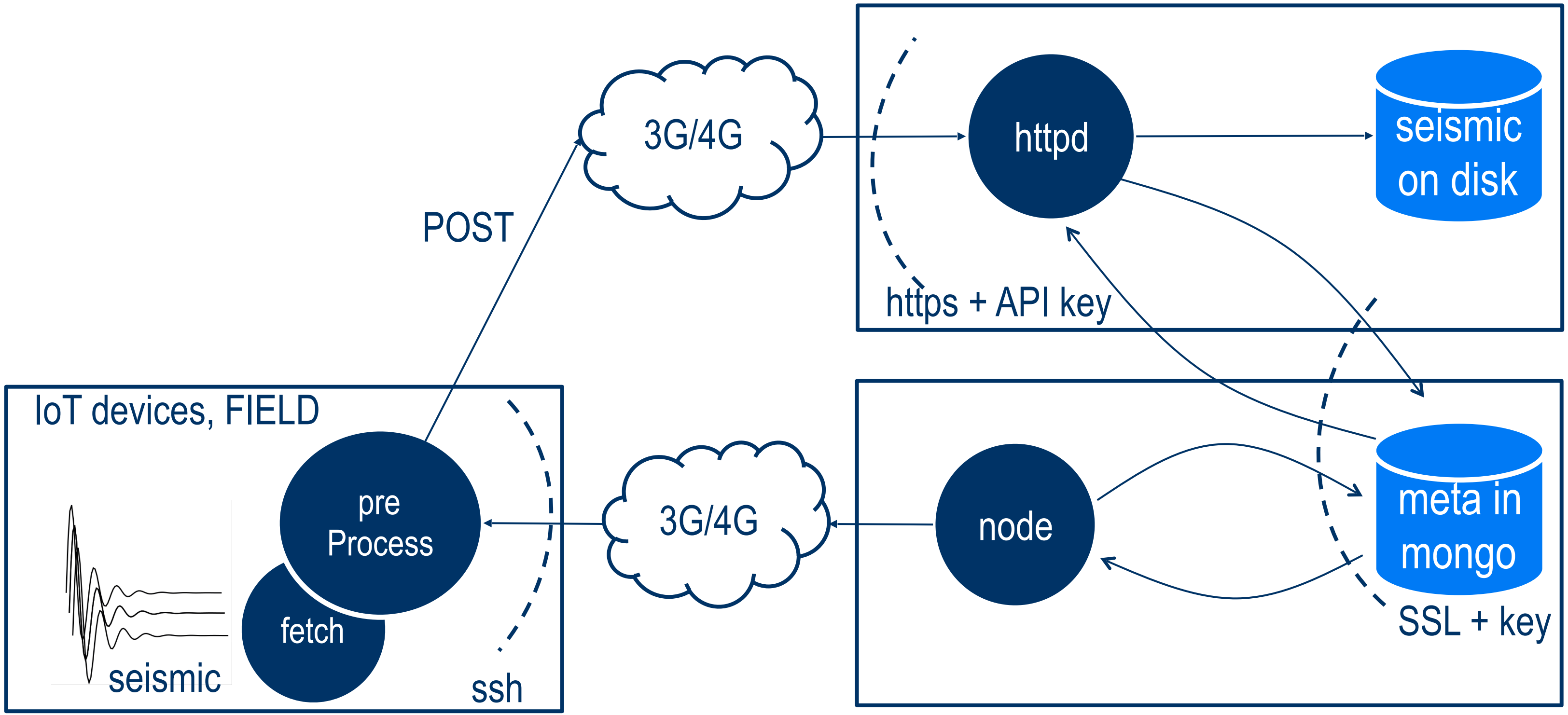
RESTful File Service

- Seismic data acquisition and upload to service from IoT nodes is initiated from dashboard



Schlumberger

IoT for seismic



IoT for seismic

- IoT nodes run in-house software('fetch') for communication with certain seismic acquisition devices.
  - Software performs request segmentation, alleviating network pressure/TCP congestion control on local network in the field.

- IoT are spatially aware ie., knows where in total spread it's situated, can perform moveout + stack using nearest neighbor receivers ('preProcessPatch')
- After stack and compression, same process POSTs to file server
  - Compression is **lossless**, simply uses zlib (gzip)

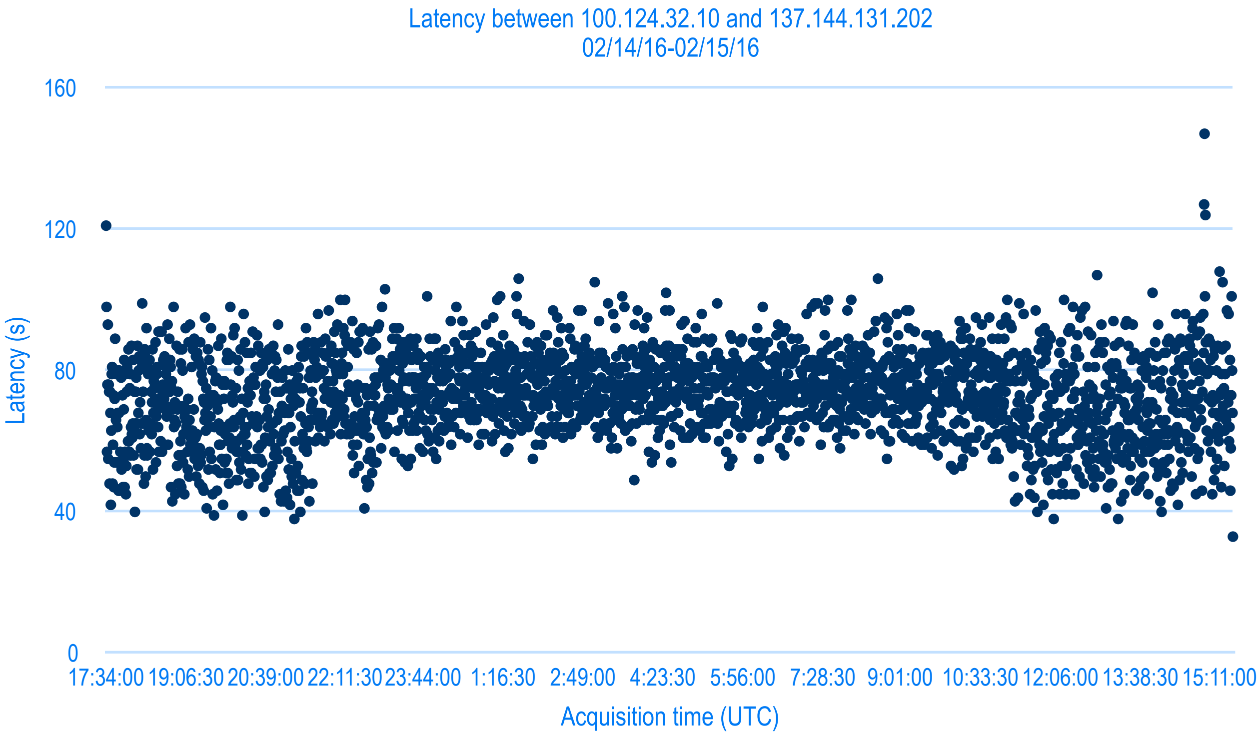
Schlumberger

IoT for seismic

- Successfully measured/optimized key variables in the field:
  - Supply battery discharge rates (~ 48 hours for 80 AmpH batt)
  - Latency between acquisition time + time of reception in Houston from field location, cellular modem bandwidths, etc etc
  - Can operate with as little as ~ 1 KB/s per channel

IoT for seismic

- Using network modestly and well is key to passing Network Application QP



Schlumberger

Outline

- Introduction
- Getting Started
- Surface Microseismic Monitoring (MSM) Architecture
  - Web server and dashboard application
  - RESTful file service
  - IoT for seismic
  - HPC using Rocket
- Additional remarks on AQP preparedness
  - DELFI requirements
- Conclusions

Schlumberger

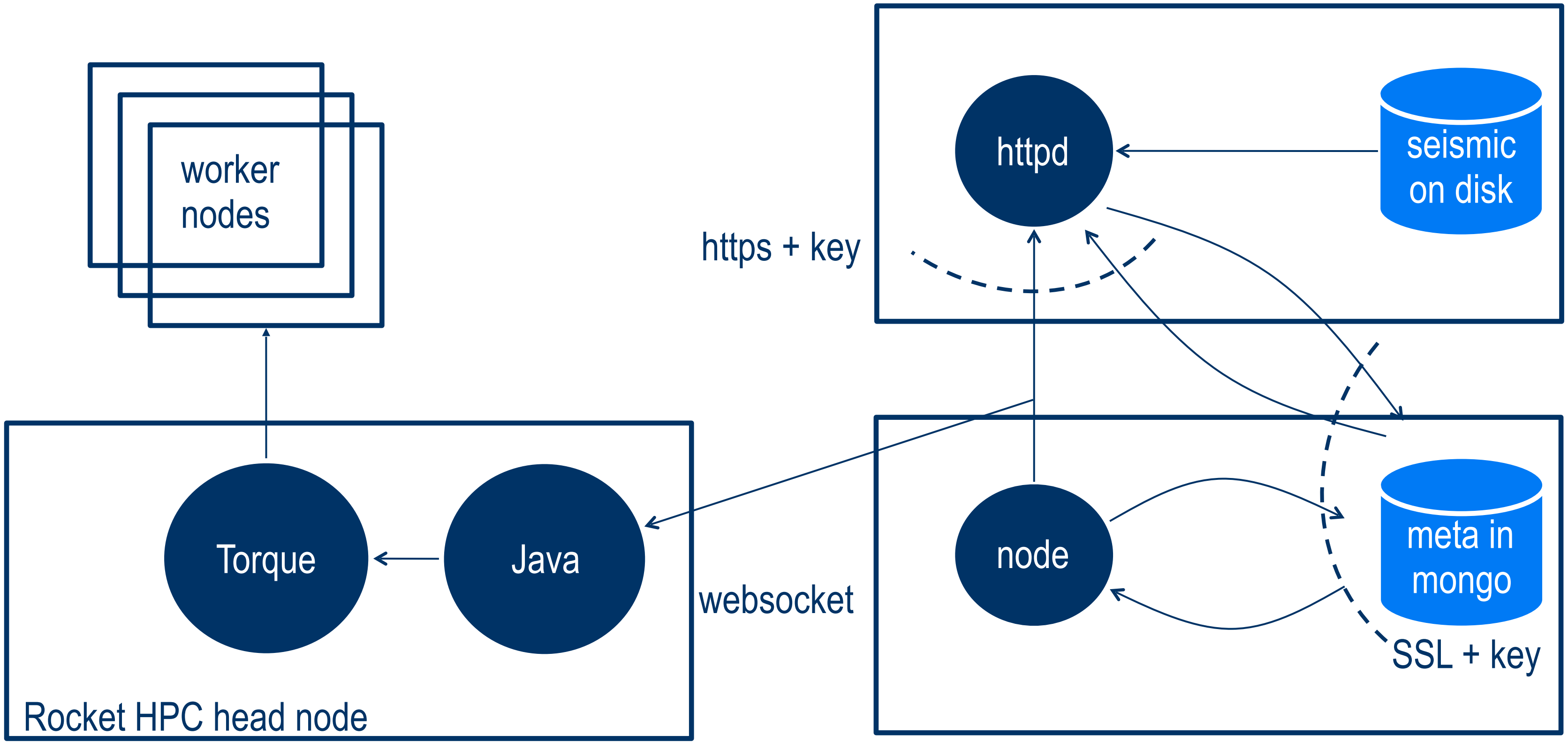
Rocket HPC

- Rocket is largely Java *middleware*, which allows client software (eg., managed by node at web server) to launch and run services, scheduled/managed by Torque.
- Rocket communicates job details to Torque, which runs simple PBS/bash scripts that wrap applications

```
let start=$(date +%s)

#PBS -l nodes=1:ppn=4
#PBS -l mem=14gb

argline="$(<argline)"

$ROCKET_SERVICE/bin/computeVTILookUpTable.x --jp jobParameters.csv &> /dev/null
$ROCKET_SERVICE/bin/msmService.x $argline

let end=$(date +%s)
let tot=$end-$start
echo server execution time $tot
```

Schlumberger

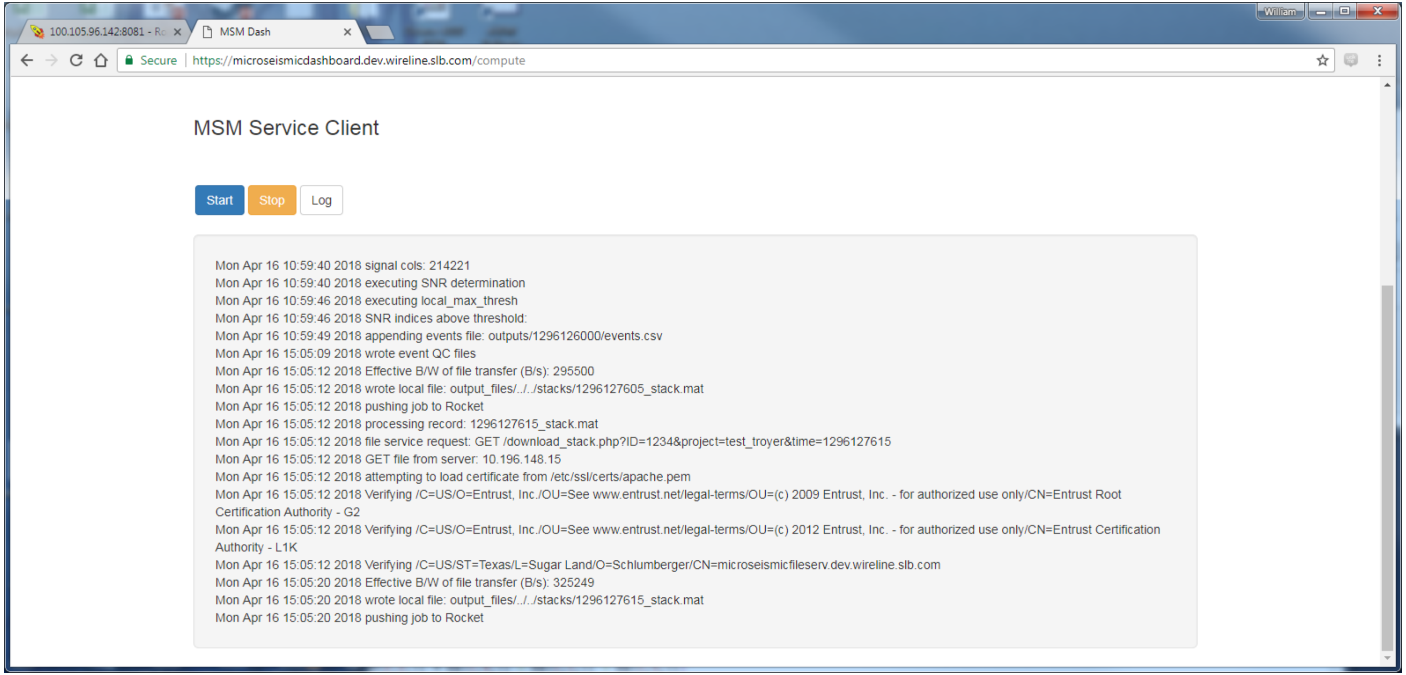
Rocket HPC



Schlumberger

Additional remarks

- **Many components/failure points in service architecture; log verbosely to allow for debugging eg., here client managed by dashboard GETs file from service**



Schlumberger

Additional remarks

- Last but not least:
  - Get to know and love **SELinux**; strange failures are usually related eg., update booleans to allow httpd daemon to network connect
  - You may need to manage firewalls on IoT and other devices yourself ie., get to know **firewall-cmd**
  - **Invest in securing nginx/apache etc, else you will fail penetration testing eg., reject endpoint requests that don't exist on your server:**

```
location ~/*.php{
        return 404;
}
```

Schlumberger

Conclusions

- Moving to cloud in a compliant manner is challenging but not impossible
  - Think in terms of vendor offerings (services) and linux
    - What are your processes/files/data/communication
  - Be conservative with resources eg., bandwidth
  - Be specific with fireflow/firewall requests, know your architecture
  - Testing/scanning is applied throughout CSQP, remove redundant endpoints in your server configurations
  - Obtain official certificates and FQDN early in your development process