Jacek's notes from the Fault Tolerance Workshop

Fault Tolerance Workshop

For sure we can't afford high-end hardware

Design system for alerts such that no scheduled outages are required during data taking

Availability requirements: need to define acceptable frequency and extent of outage

- e.g. ok to have 1 msec outage every 1 sec?,
- ok to have 3 day outage every 300 days?

(both examples give 99% availability)

High reliability is not only good for science, but saves on budget

- bad reliability requires experts available on call 24x7 to fix problems, that is expensive

we should provide "service level agreement" to infrastructure people, see example "sample requirements" on FaultToleranceInterfaces page

need to pick something to start costing (will we use redundancy? retry?, ...)

need to provide FT within existing budget (challenge)

sandboxing is useful, but adds lots of complexity

will users be bringing and executing their own code inside pipeline?

- unclear
- will probably have clusters of machines for user code that is not part of production

certain thing "have to run", some "would be nice to run", system should allow such division

reruning has two modes:

- rerun to fix problems. Maybe we want to overwrite bad data in this case. Might keep multiple versions, flag bad versions (or trash)
- rerun to reprocess data with better code/calibration etc. It is a new version, can't rewrite

fault tolerance will take advantage of sophisticated provenance, ft is customer of provenance

requirement: don't release data until you have provenance recorded

real time processing: is it better to stick to deadline and if we miss it, skip it, or is it better to deliver alert about interesting event later (eg 3 min after deadline)?

- look at time distribution, look at system capacity, see how much we can afford we really have 3 different deadlines:

- 1 min (real time alerts)
- 24 h (nightly redone at main archive)
- 6 months (deep detection)

it is much easier to diagnose problem if we do not fail over

it is probably fine if things fail for a moment from time to time if that results in building much simpler system

need to try to build firewall against mistakes that will make large portions of data non-usable (eg entire night)

- does this task belong to middleware or app/sdqa?
- app people will give us rules to apply, middleware needs to provide code, we need to think what middleware should support, eg in babar users could manually load calibration constants, which were used during data processing. Mistake made: there was no way to check what constants were used --> there was a missing piece in the middleware code

human mistakes will dominate, especially at the beginning

failure types

many of things of FaultToleranceUseCases are really characteristics of failure, not types

aggregation of failures: every step takes 0.5 sec longer than average (these are not failures), but overall it looks like a failure because total time will be above threshold

correlations of failures, eg. due to batches of same hardware, environment, (eg high temperature in the room)

security plan will cover some of the issues

don't have to catch every failure at fine grain level

who is watching the watchers? (catching failures of the fault tolerance system)

role of sdqa and impact on ft design?

tradeoff:

- pushing ft lower: easier (eg we can just buy fault tolerant network),
- pushing ft higher: complex and hard to implement but it is the only way to implement a complete fault tolerance, that means going towards application code

requirements (Gregory)

- srd.pdf (science requirements document)
- dm functional requirements.pdf, docushare document-5438

any information lost that prevents us from processing image should be treated as loosing the image

0.1% alert publication failure

98% availability

- need to define requirements better, eg what about lots of very short failures vs one long failure...

"less than 24h downtime" requirement

- doesn't make sense
- eg it does not tell us how much diesel fuel we might want to buy to keep generators running, power outage can be very long...
- this requirement is derived from 60 sec real time alerts (based on one of the diagrams) - this is wrong
- what about 23 h down, 5 min up, 23 h down, 5 min up, is this acceptable? --> raise this with Jeff

we think the plans are that DR1 will be always kept on disk

> --> ask Jeff if it is captured somewhere in the official requirement document?

DM-APP-DP-CA-7 does it imply sources in real time?!?

> --> Ask Jeff

DM-APP-DP-CA-11 is confusing too

architecture

for nightly, we expect core per amp, so 3000 cores, expecting to have 16 core per box, so ~200 boxes

if we are going to meet deadline, if something fails we can:

- redo entire image if failed, or
- checkpoint, redo since last checkpoint, or
- reprocess twice in parallel (full redundancy)

we should not checkpoint to local disk, so we need SAN

- a reasonable option: many small SAN clusters, say 2 disks per 8 machines

can we drop pieces of image?

> --> ask application people
>
> if so, what granularity? amp? ccd? raft?
>
> > --> ask application people
>
> if we can drop parts, maybe we could just ignore failures of single machines?

are there any dependencies between different images? (ghost images) it might be useful to have a node dedicates to redoing processing of the very same image every day to uncover problems with new code

full single copy of templates is 225 TB

--> check is we are planning to have enough storage at base camp?

0.1 arc sec = 1.5 pixels = point source

maybe we should keep two copies of catalog at base camp, and update one copy with a day-delay, in case we mess up the latest copy, we can revert to the day-old copy

--> find where are documents describing hardware at base camp?

practices

- monitor (watchdogs)
- redundant execution / auto fail over / rescheduling
- limit updates
- keep immutable and mutable data separate

--> check xproof / xrootd

mapReduce worry: must do checkpointing between map and reduce stages (lots of io), but some data is read only (eg calibration), in non-MR world we can keep such data in memory and avoid IO

there is 16 amplifiers per ccd, we can collocate all amps from a ccd on a single 16 core box

we need 80 MB of IO per image (80 bits per pixel in image x 1 Mpixel image per amp)

- 80 because: 32 + variance (32) plus masked (16)
- x 16 cores per machine
- x 2-3 images we need per stage (raw, calibration, template)
- have ~10 stages, some need less io

- need to decide how much time from the 30 sec/visit we want to devote to io

- also, if we have 1 min requirement to deliver alert, and 30 sec per visit, io is x2

- bottleneck: getting data off the chip and out of the box
    - checkpointing would require: 11.5 GB
    - need ~1-2 sec per stage to checkpoint

- RAM seems ok, 8 GB / box sufficient to do image processing

quad resolution in template images

--> this is not captures in storage estimages, follow up!

postage stamps coming out of calibrated images or difference?

- calibrated

at main archive we will save calibrated images, at base camp we only need to save postage stamps

options we have:

- double the capacity (full redundancy)
- checkpoint, it requires tens % more hardware to handle extra io, plus need high speed SAN
- maybe it is ok to design system that continues if one amp fails?

    --> need to talk to app people

how long it'll take to reconfigure system after node failure

- if fault tolerant MPI does not allow us to do it quickly, might consider other approaches
- xrootd looks like a good candidate to consider

we should capture requirements and representative usecases in the doc for pdr

need to consider catastrophic failures separately

- example: database
- in case of database, we probably want to maintain 2 synchronized databases (redundant spare)
    - hardware for redundant db servers already in baseline for base camp

how to make sure components are fault tolerant?

- we can define classes of fault tolerance and decide which class each component belongs to
- or developers must deliver ft components
- or we can have ft experts that consult/help

provenance worries that Gregory has:

- if we are serious about provenance, we need to capture all dependencies, eg all shared libraries, we might also want to keep contents of external libraries
    - example, innocent library changed runtime flag which switch the way 64-bit int is treated (80 bit representation instead of 64), affected floating point calculations for the executable
- that is hard, pushing us towards sandboxing
- also, how are we going to configure machine and install software on it based on provenance info?
- also, how do we know if code does not open some random files in random places

--> put in requirements: all I/O to disk goes via middleware.

- this simplifies provenance
- and catching failures
- but we need to be careful, because don't want to introduce extra copying of data

application code may decide to change algorithm during execution, if that happens, it needs to have a way to report it for provenance

remote disk similar speed as local disk because network improves?

- yes, but local disks derandomize io

checkpointing: development cost to implement checkpointing is modest, we will implement it, it will be configurable: we can turn it on or not

new usecases:

- marked piece of data as good, later after it went to community we discover it is bad for some kinds of analysis
- we rerun program twice and each time we get different result
- someone find something interesting in data with bad calibration in nightly catalog
- user retrieves image
- network problems to ncsa
- we were told to prepare for position x, the real observed position is y
- system log filled out (eg /var/log/messages)
- aggregation of failures: every step takes 0.5 sec longer than average (these are not failures), but overall it looks like a failure because total time will be above threshold