Middleware development for reliable services

Gavin McCance

WLCG Collaboration Workshop, CERN

April 2008

Overview

- **This is a review of the techniques and best-practices that you can make use of in your software so that we can run a reliable service using it**
  - Mostly a review of what was discussed at the WLCG Service Reliability Workshop in November
  - And subsequent discussions, papers and presentations since

- **Total cost of ownership to WLCG is the development / test / release cost PLUS the ongoing operations cost**
  - We'll be running a service for a long time…
  - Our operations budgets are not getting larger!
  - For large internet services, industry quotes two orders of magnitude for the ongoing operations cost (for manpower) between a operations-friendly service and one that isn't

Summary

- **“80% of operations problems occur due to problems in the design and development of the software…”[1]**
  - And should be fixed there !
  - Keep it simple
    - Avoid critical dependencies and complexity
  - Design assuming failure
    - Use a decoupled and redundant architecture
  - Automate everything
    - Reduce the operations cost
  - Measure everything
    - To understand and debug things quickly
  - Test everything
    - Release process and stress-testing

[1] “On Designing and Deploying Internet Scale Services, 21st LISA conference, 2007

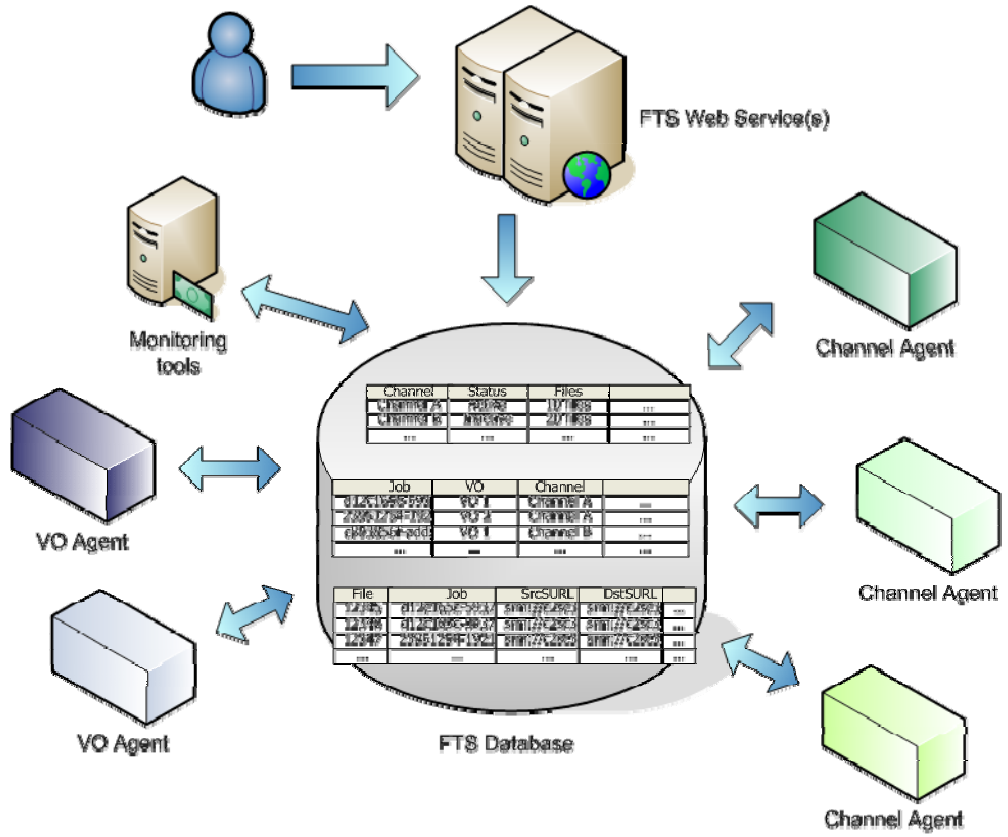
Keep it simple

- **Architecture: components should be decoupled**
  - The service should be designed to minimise cross-component dependency
    - Expect other components to be down
    - Expect latencies – cache if needed
    - Expect glitches in dependent Grid services – retry connections
    - Isolate failures: fail-fast, don't propagate a failure; e.g. partition queues so fast queues don't get clogged by high latency operations
    - Partition: can one part go down without impacting another part of the service, or another service class?
  - Helps make your software "operations friendly"
    - If I can reboot a node and all the rest of the service keeps working, this is really helpful for operations – transparent interventions!
    - If I can reboot a node without needing the 'drain' it, this is even better
    - If I can crash a node without having to worry, this is even [more] better

Expect node failures

- **One of things we are seeing happening now in cloud computing infrastructures like Amazon EC2 is the notion of transient boxes and how to create 99.999+% services that run on these**
  - Moving away from more expensive boxes to cheaper (virtual) ones
  - At one extreme it's suggested (by some) that the standard way of bringing down a box for intervention on a production service should be to pull the power plug
    - If the operations team is 'afraid' to do this, the software is not reliable in face of a hardware problem
    - And the total cost of ownership will be significantly higher
  - Not suggesting we go down 'virtual compute cloud' path ☺
    - but real consumer applications built on things like EC2 show that this level of service reliability in the software is attainable!

Example: decoupled architecture



User-facing web-service is decoupled from the agents that do the work

VO and Channel agents are partitioned

Monitoring and statistics can be collected via the DB and are decoupled from the core service

- **Key: all components are decoupled from each other**
  - A failure of one ~doesn't impact the others

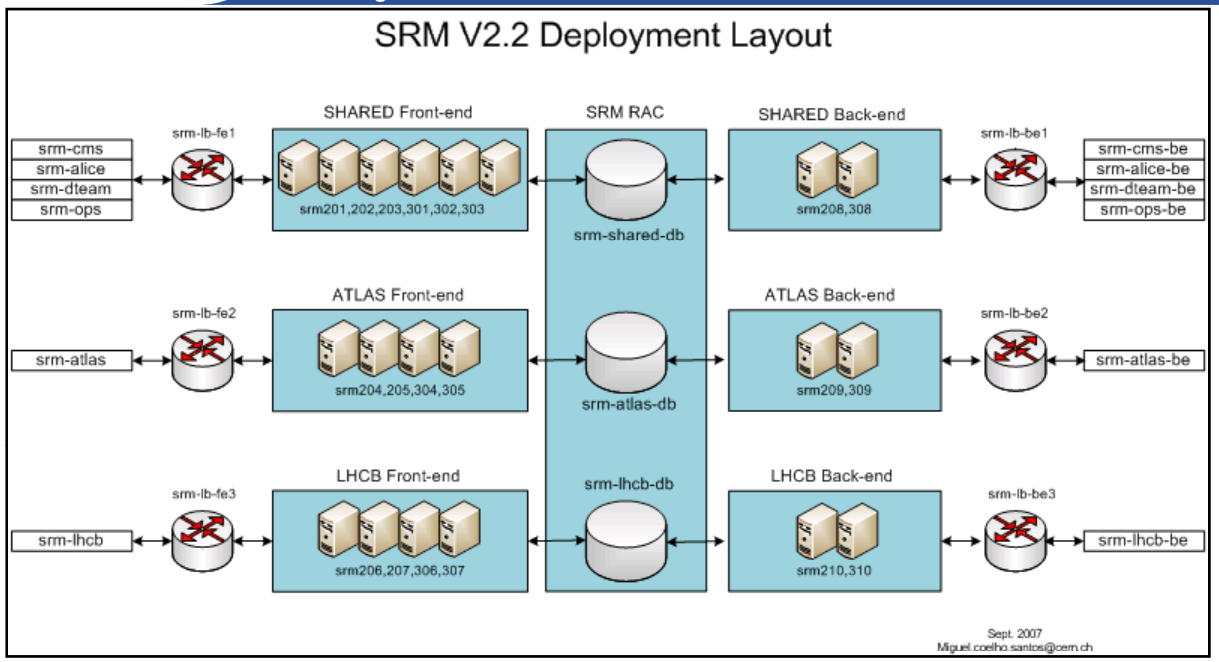
Understand service dependencies

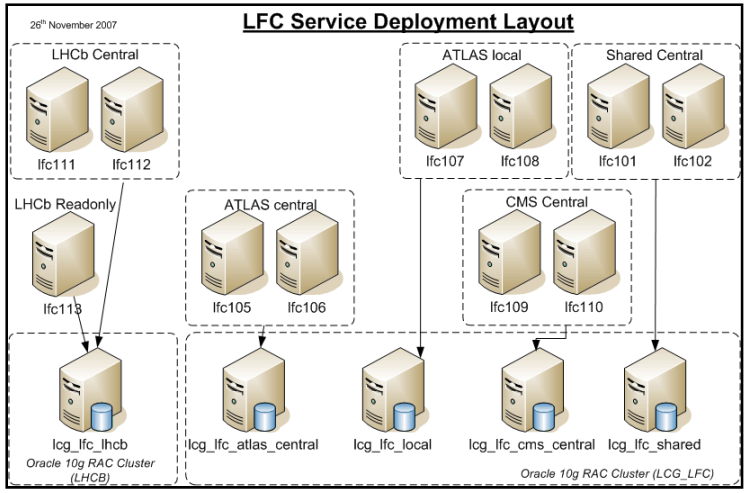
- **Think service!**
- **A grid service may have several components to it**
  - Daemons, databases, storage systems (local filesystem / remote)
  - Hardware running these
  - Operations staff
  - Cross-site interactions (to other services, staff)
- **As a developer working together with the service manager:**
  - It's good to review all the components and ask 'what's the impact to the overall service if this bit fails?'
  - If the answer is 'the whole thing stops', ask what can I do to fix that?
    - Can we use the deployment architecture to help?
    - What is the impact of this on my software?
  - What standard things does the service manager need to do?
    - Kernel upgrades (reboot), vendor calls (broken hardware), etc
    - DOCUMENT ALL THIS! The developers should help write the operations procedures!

Technique: load-balancing

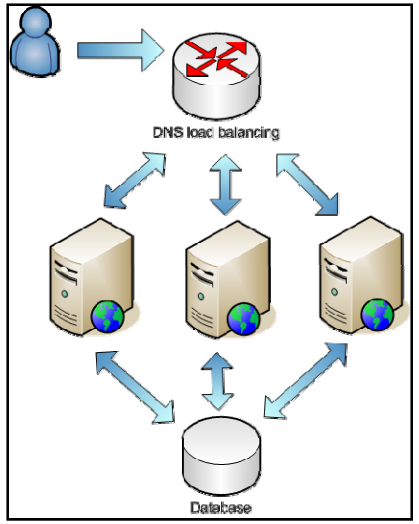
- **Avoid SPOFs! An easy way to isolate the service from failure is by using dynamic load-balancing**
  - Operations friendly: you can intervene on a node without perturbing service since the other nodes are available to handle the requests
  - Reliability: isolate from critical failures
  - Scaling: helps you scale the service cheaply
- **Requirements on software**
  - Decoupled with clear responsibilities
  - It's (much much) easier to do if the component is stateless

Examples: load-balancing



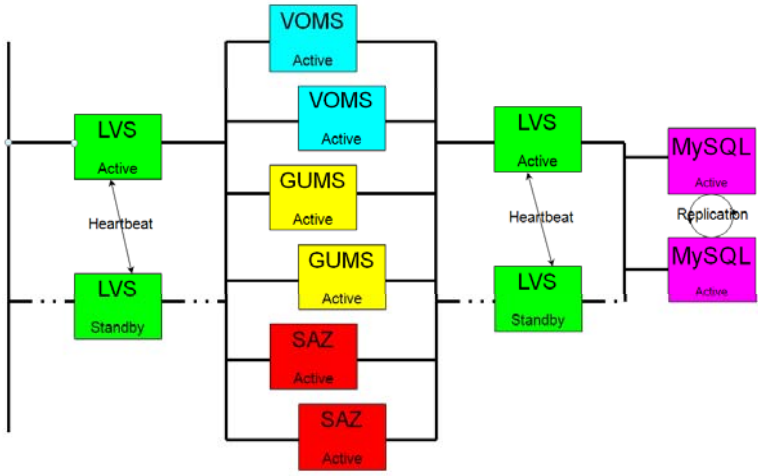


FTS

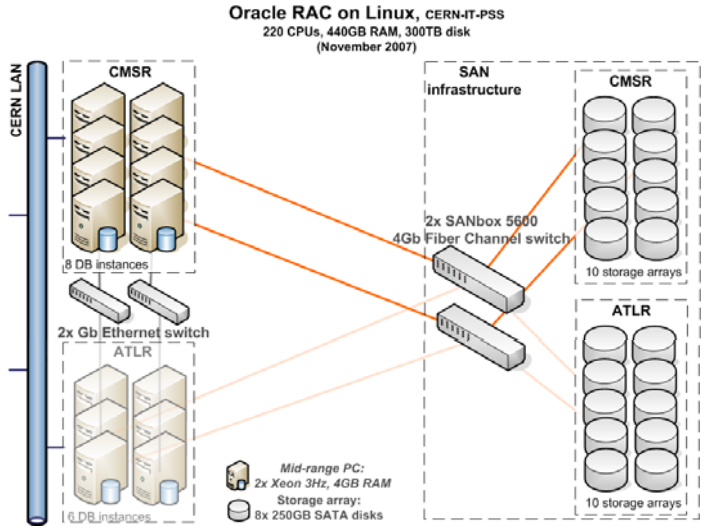


Technique: store it in a reliable DB

- **Store valuable state in a DB or other reliable store**
  - We still run services with vital stuff living on the local filesystem
  - A lot of man hours has gone into making DBs good places to store data you care about
  - There's a whole industry of operations procedures for storing, backing up and recovering the data in case of operational problems



FermiGrid-HA MySQL services



(CERN) Oracle RAC services

Summary of DB techniques from November

- **Use bind variables**
  - Otherwise you kill performance
- **Connection retries**
  - To ride out glitches
- **Use the DB's HA features – e.g. Oracle's Transparent Application Failover**
- **Connection pooling**
  - avoid hammering the DB with new connections
- **Integrity constraints**
  - Particularly important in case of 'problems' or 'special scripts'
    - 'Special' scripts should be tested like all the rest
  - Logical schema corruption is horribly expensive to recover from
- **Test at the appropriate scale**
  - Have a good validation service and test using realistic data and loads
  - Directing a stream of less critical production to the validation service is sometimes useful
- **Buy your DBA a beer ☺**
  - Then they'll tell you about query tuning, partitioning, suggest indexes, etc
  - These things usually become important a couple of weeks into your first production run, when it's rather inconvenient to fix it
  - (Miguel, I owe you a beer…)

Technique: hot-standby

- **Avoid SPOFs! Hot-standby**
- **Apart from all the usual hardware considerations**
  - RAID, dual power supply, etc
- **Requirements on software:**
  - Make it easy to swap in the stand-by node (warm?)
    - Even better, make it automatic! (hot?)
  - Make the configuration simple
  - Transparent take-over: good if the hot stand-by and production daemons can co-exist
- **LFC is a good example here**

Use commodity software

- **Millions of man hours have gone into designing good quality, highly available open source software solutions that you can use as building blocks for your service**
  - J2EE: Tomat, JBoss (open)
  - HTTP: Apache/mod_ssl, squid for caching (open)
  - Directory services: openLDAP (open)
  - Messaging: Apache ActiveMQ, email(?) (open)
- **Operations teams know how to run these**
  - Massive amounts of experience and advice
  - Significantly lowers the total operations cost of a service
- **→ As a service architect, you should have a really good reason not to make use of these building blocks**
  - If you have to significantly change the architecture and protocols you use to be able make use of open-source tools like these, you should critically question whether the architecture and design are good for the longer-term viability of your software

Automation

- **Automate as much as possible**
  - Humans need sleep, machines don't
- **Requirements on software:**
  - Easy and reproducible configuration – it has to be do-able by a script
  - No local storage – a script can't rescue files that are on a system that's come back up in single-user mode with faulty RAID array
  - Commit persistent state to a DB – same point
  - Make it easily re-startable
    - Designing in any recovery actions (understand how to recover from any dirty state that you do have)
- **Work with an operations team to test this**
  - They need to have confidence in your software!

Measurement and monitoring

- **Measure and monitor everything you can think of**
  - The performance hit is marginal compared to the ongoing operations cost of not having it
- **Many subtle problems between components are a result of things timing out**
  - Timing data is good – how long does each operation take?
- **Work with operations to build alarms into the software**
  - Developers should have an idea what an alarm should trigger on
  - What's a good indicator of problems?
- **But watch the quality of alarms**
  - You don't want false alarms
    - Or the operations team will learn not to trust them
  - Always have an action: if you never act on the alarm
    - Rewrite it to something more useful

Testing and release

- **Use a good software development process**
  - See other talks…

- **Test, test, test**
  - On your box, using as much as possible the same environment as the production to avoid the “it works on my box” problem
  - Then on a pilot service, at a good scale as you can afford
    - With fake data
    - Then redirecting a stream of production data (be ready to roll it back)
      - If this scares the operations team, you should ask why ☺
  - Then in limited beta on production
  - Then release to everyone

When to do all this?

- At the architecture and design stage
- Involve the operations team early to get their experience
- Doing it afterwards is expensive
- Doing it forever in the operations is even more expensive and not sustainable

Summary

- **There are techniques available that can significantly improve the service provided by software**
  - But they impact significantly the architecture and design
  - Do it as early as you can – get your operations people involved!
  - Developers should help write the operations procedures!
- **Keep it simple – think service!**
  - Complexity will kill your Service Level
    - Hacks and workarounds to fix ‘mysterious’ problems cost real money
  - Decouple and partition your components
  - Design assuming anything can and will fail
    - and do your best to ensure the service keeps on going even if at a reduced capacity
- **Robustness**
  - Design for load-balancing and hot-standby
  - Build your application using commodity DBs and other commodity software and know the rules for getting the best out of these

Summary

- **Measure everything to aid debugging in production**
- **Work with operations team to think what should be alarmed**
- **Test lots, then test more**
  - Use representative (ideally production-scale) load
- **If we make our software 'operations friendly'[1] then we should be able to keep the ongoing operations cost to WLCG at a sustainable level while meeting our MoU targets**