Overview of Current Performance and Performance Plans

Bill Allcock
ANL
20 Jan 2004
Java Core
Core Performance

- We’ve been working hard to increase basic messaging performance
  - Factor 4 improvement so far
- We’re testing reliability
- We’ve shown that core can scale to a very large number of resources (>>10000)
Core Messaging Performance

Message Size

- Axis Update Branch (1/10/05)
- CVS Head (1/10/05)
- CVS Head (11/05/04)
- CVS Head (11/01/04)

Time (ms)

(Message Size (number of GRAM subjob messages))
Security Performance

- We’ve measured performance for both WS and transport security mechanisms
  - See next slide for graph
- Transport security is significantly faster than WS security
  - We made transport security (i.e. https) our default
  - We’re working on making it even faster by using connection caching
C WS Core
C WS Core Clients: Java vs. C

- Java VM startup: large initial overhead
  - Simple Java client Request/Response: ~5 seconds
  - Simple C client Request/Response: ~0.5 seconds
C WS Core Performance: Service Container

- **Without Security**
  - Java Container
    - 0.36s avg. Request/Response
  - C Container
    - 0.015s avg. Request/Response

- **With Security**
  - Java Container
    - 0.66s avg. Request/Response
  - C Container
    - 0.12s avg. Request/Response
C Performance Improvements

- HTTP Persistence
  - No Security, No Caching
    - 0.25s avg. Request/Response
  - No Security, With Caching
    - 0.17s avg. Request/Response
  - With Security, No Caching
    - 2.6s avg. Request/Response
  - With Security, With Caching
    - 0.52s avg. Request/Response
C Performance Improvements (Planned)

- Improved Deserialization performance of optional schema elements
- WS-Security performance:
  - Inlined Canonicalization
C globusrun-ws Performance

- Query Delegation Factories: 0.046s
- Query Certificate Chain: 0.058s
- CreateManagedJob: 0.12s
- Active Notification: 5.11s
- Cleanup Notification: 0.73s
- Done Notification: 2.29s
- C client total processing time: 1.12s
Usage Scenarios: the Ideal

“GRAM should add little to no overhead compared to an underlying batch system”

- Submit as many jobs to GRAM as is possible to the underlying scheduler
  - Goal - 10,000 jobs to a batch scheduler
  - Goal – efficiently fill the process table for fork scheduler

- Submit/process jobs as fast to GRAM as is possible to the underlying scheduler
  - Goal - 1 per second

- We are not there yet...
  - A range of limiting factors at play
Usage Scenarios: the Attempt

- **Efforts and features towards the goal**
  - Allow job brokers the freedom to optimize
    - E.g. Condor-G is smarter than globusrun
    - Protocol steps made optional and shareable
  - Reduced cost for GRAM service on host
    - Single WSRF host environment
    - Better job status monitoring mechanisms
  - More scalable/reliable file handling
    - GridFTP and RFT instead of globus-url-copy
    - Removal of non-scalable GASS caching
  - **GT4 tests performing better than GT3 did**
    - But more work to do
GRAM 3.9.4 performance

- Service performance & stability
  - Throughput
    - GRAM can process ~70 /bin/date jobs per minute
    - ~60 jobs/minute that require delegation
  - Job burst
    - Many simultaneous job submissions
    - Are the error conditions acceptable?
  - Max concurrency
    - Total jobs a GRAM service can manage at one time without failure?
  - Service uptime
    - Under a moderate load, how long can the GRAM service process jobs without failure / reboot?
Plans for Future Testing

• More throughput testing with different job types (staging, schedulers, etc)
• Max concurrency testing (how many jobs can be queued up)
• Long running test (Moderate load, service running for weeks or months)
• Also work being done on usability
GridFTP
Current Development Status

- GT3.9.4 has a very solid alpha. This code base has been in use for over a year.
- The data channel code, which was the code we added to wuftp, was re-used and so has been running for several years.
- Initial bandwidth testing is outstanding.
- Stability testing shows non-striped is rock solid (leaks maybe 30K / day)
- HOT OFF THE PRESS: Striped had a memory leak, which we think we have fixed.
- [http://dc-master.isi.edu/mrtg/ned.html](http://dc-master.isi.edu/mrtg/ned.html)
Status continued

- Stability tests to date have been for a single long running transfer
- We are working on sustained load and “job storm” tests
- A usable response in the face of overload is a key goal.
- Completed an external security architecture review
  - Likely to make changes to the “recommended configuration”
  - This is a deployment issue, not a code issue.
- Planning an external code review.
TeraGrid Striping results

- Ran varying number of stripes
- Ran both memory to memory and disk to disk.
- Memory to Memory gave extremely high linear scalability (slope near 1).
- We achieved 27 Gbs on a 30 Gbs link (90% utilization) with 32 nodes.
- Disk to disk we were limited by the storage system, but still achieved 17.5 Gbs
Memory to Memory Striping Performance

BANDWIDTH Vs STRIPING

Bandwidth (Mbps)

Degree of Striping

# Stream = 1  # Stream = 2  # Stream = 4  # Stream = 8  # Stream = 16  # Stream = 32
Disk to Disk Striping Performance

Bandwidth (Mbps) vs Degree of Striping

- # Stream = 1
- # Stream = 2
- # Stream = 4
- # Stream = 8
- # Stream = 16
- # Stream = 32
RFT Testing

• Pre 3.9.4
  ◆ Out Of Memory Exception when generating notifications @ ~3/sec. Fixed in core.
  ◆ Out of Memory When the transfer request size was > 5000 transfers. We altered the schema to allow minOccurs=0 for optional fields. We currently scale to 21020 transfers for a single request.
  ◆ Out Of Memory When transferring a directory of more than 15000 files. We had a data structure in memory that scaled with the size of the request, this is no longer true. Now we can scale to ~60,000 files (that was what has been tested)

• Current Testing :
  ◆ Infinite transfer - LAN - killed the container after ~120,000 transfers. Servers were killed by mistake.
    ● Was a good test. Found a corner case where postgres was not able to perform ~ 3 update queries / sec and was using up CPU.
  ◆ Infinite transfer - WAN - ~67000 killed because of the same reason as above
  ◆ Infinite transfer - 3 scripts creating transfer resources of one file with life time of 5 mins. Found a synchronization bug and fixed it. -- Active
  ◆ We got an error at 28 transfers before with one script. Now each pf the three scripts have successfully finished 200 resources each.
  ◆ directory transfer with 16304 directories, 92578 files -- Currently active
MDS
MDS Query results

- Only one set of data so far. No data yet for Trigger Service. Ran at this load for 10 minutes without failure.

- **DefaultIndexService**
  - Message size 7.5 KB
  - Requests processed: 11262
  - Elapsed Time: 181 seconds
  - Average round-trip time in milliseconds: 16

- **ContainerRegistryService**
  - Message Size 32KB
  - Queries processed: 6232
  - Elapsed Time: 181 seconds
  - Average round-trip time in milliseconds: 29