LCI workshop
on high performance clustered computing

LCI Advanced Workshop 2025: **Slurm API Exploration**

Alan Chapman
HPC Systems Analyst
Arizona State University Research Computing
alan.chapman@asu.edu

ASU Research Computing

Agenda Overview

LCI workshop
on high performance clustered computing

Content:

- Slurm Architecture Review
- Slurm API Fundamentals
- Slurm REST Architecture
- Practical Implementation
- Live Code Demonstrations
- Comparison & Use Cases
- Q&A

Slurm Architecture Review

LCI workshop
on high performance clustered computing

Controllers & Daemons:
- slurmctld: Central controller
- slurmd: Compute node daemons
- slurmdbd: Database daemon

Communication Methods:
- Socket-based RPC (traditional)
- REST API (modern approach)

Data Flow:

User Tools → Controller → Nodes
↓
Database

Traditional Slurm Communication

LCI workshop
on high performance clustered computing

slurmctld Process:
- Listens on port 6817 (configurable)
- Handles all scheduling decisions
- Manages job state

Direct RPC Communication:
- Protocol buffers
- Direct socket connections
- C library-based (libslurm)

Limitations:
- Language-specific bindings required
- Complex state management
- Tight coupling

Slurm REST API Introduction

- **What is slurmrestd?**
  - Standalone REST daemon
  - HTTP/HTTPS interface
  - JSON request/response format

- **Key Advantages:**
  - Language agnostic
  - Firewall-friendly (HTTP)
  - Easy integration
  - Scalable

- **Architecture:**
  HTTP Client → slurmrestd → slurmctld

REST API Components

- **OpenAPI Specification:**
  - Version 0.0.38 (Slurm 22.05+)
  - Fully documented endpoints
  - Schema validation

- **Endpoint Categories:**
  - Jobs (query, submit, cancel)
  - Nodes (status, configuration)
  - Partitions (properties, limits)
  - Accounts/Users (accounting)
  - Controllers (health, status)

- **Authentication:**
  - Token-based (JWT)
  - Munge authentication
  - Optional TLS/SSL

REST API Security

Network Security:
- TLS/SSL encryption support
- Port configuration (default 6820)
- IP whitelisting possible

Authentication Methods:
- Munge tokens (default)
- JWT tokens
- User/token pairs

Authorization:
- Inherits Slurm ACLs
- Per-user access control
- Role-based restrictions

Python Slurm API – Direct Binding

LCI workshop
on high performance clustered computing

Method 1: Using ctypes & libslurm

- Direct C library access
- Fast performance
- Requires libslurm installed

Key Functions:

- `slurm_load_jobs()`
- `slurm_submit_batch_job()`
- `slurm_kill_job()`

Advantages/Disadvantages:

- ✓ Direct, no daemon required
- ✓ Full feature access
- ✗ Complex state management
- ✗ Lower-level code

Python Slurm REST - HTTP Method

LCI workshop
on high performance clustered computing

Method 2: Using requests library
- HTTP-based interaction
- Requires slurmrestd daemon
- Simpler implementation

Key Endpoints:
- GET `/slurm/v0.0.38/jobs`
- POST `/slurm/v0.0.38/jobs`
- DELETE `/slurm/v0.0.38/job/{job_id}`

Advantages/Disadvantages:
- ✓ Simpler, standard HTTP
- ✓ Cross-platform
- ✗ Requires daemon running
- ✗ Slightly higher latency

Slurm Command-Line Tools

- **Traditional Approach:**
  - `squeue` - List jobs
  - `sbatch` - Submit jobs
  - `scancel` - Cancel jobs
  - `sinfo` - Node information
  - `sacctmgr` - Account management

- **Query Results:**
  - Human-readable output
  - Parseable formats (JSON, CSV)
  - Regular polling required

- **Use Cases:**
  - Quick manual operations
  - Script-based automation
  - Monitoring integration

Comparative Analysis

Feature	CLI	libslurm	slurmrestd
Speed	Medium	Fast	Medium
Language Support	Any	C/Python	Any (HTTP)
Setup Complexity	Simple	Medium	Medium
Network-based	No	No	Yes
Real-time Updates	Polling	Yes	Polling
Scalability	Limited	Limited	Excellent
Firewall-friendly	No	No	Yes

Practical Example 1 - Query Jobs

LCI workshop
on high performance clustered computing

```python
import requests
import json

# Query all jobs
response = requests.get(
    'http://localhost:6820/slurm/v0.0.38/jobs',
    headers={'X-SLURM-USER-NAME': 'username'}
)

jobs = response.json()['jobs']
for job in jobs:
    print(f"Job {job['job_id']}: {job['name']} - {job['job_state']}")

Output:
Job 1001: analysis.sh - RUNNING
Job 1002: simulation.py - PENDING
Job 1003: report.R - COMPLETING
```

Practical Example 2 - Submit Job

LCI workshop
on high performance clustered computing

```python
import requests

# Submit a batch job
job_script = """#!/bin/bash
#SBATCH --job-name=my_job
#SBATCH --time=01:00:00
module load python
python script.py
"""

data = {
    'script': job_script,
    'job_name': 'my_job'
}

response = requests.post(
    'http://localhost:6820/slurm/v0.0.38/jobs',
    json=data,
    headers={'X-SLURM-USER-NAME': 'username'}
)

job_id = response.json()['job_id']
print(f"Job submitted: {job_id}")
```

Practical Example 3 - Monitor Job

LCI workshop
on high performance clustered computing

```
import time
import requests

job_id = 1001

while True:
    response = requests.get(
        f'http://localhost:6820/slurm/v0.0.38/job/{job_id}',
        headers={'X-SLURM-USER-NAME': 'username'}
    )

    job = response.json()['job'][0]
    print(f"State: {job['job_state']}")

    if job['job_state'] in ['COMPLETED', 'FAILED', 'CANCELLED']:
        break

    time.sleep(5)
```

Practical Example 4 - Node Status

LCI workshop
on high performance clustered computing

```python
import requests

# Get all node information
response = requests.get(
    'http://localhost:6820/slurm/v0.0.38/nodes',
    headers={'X-SLURM-USER-NAME': 'username'}
)

nodes = response.json()['nodes']
for node in nodes:
    print(f"{node['name']}: CPUs={node['cpus']}, "
          f"State={node['state']}")

# Filter for problematic nodes
problem_nodes = [n for n in nodes
                 if n['state'] not in ['idle', 'allocated']]
print(f"\nProblematic nodes: {len(problem_nodes)}")
```

Advanced: Bulk Job Operations

LCI workshop
on high performance clustered computing

```
# Cancel multiple jobs
job_ids = [1001, 1002, 1003]

for job_id in job_ids:
    response = requests.delete(
        f'http://localhost:6820/slurm/v0.0.38/job/{job_id}',
        headers={'X-SLURM-USER-NAME': 'username'}
    )
    print(f"Cancelled {job_id}: {response.status_code}")

# Or cancel all jobs for a user
response = requests.get(
    'http://localhost:6820/slurm/v0.0.38/jobs',
    headers={'X-SLURM-USER-NAME': 'target_user'}
)
```

Performance Considerations

Query Performance:
- REST API: 50-200ms per request
- libslurm: 10-50ms per request
- CLI: 100-500ms per call

Polling Strategy:
- Avoid polling every second
- Use 5-10 second intervals for monitoring
- Subscribe to event notifications when available

Connection Pooling:
- Reuse HTTP connections
- Session management critical
- Reduce latency significantly

Batch Operations:
- Submit multiple jobs together
- Use efficient filtering

Production Deployment

slurmrestd Configuration:
- Run as dedicated service
- Use systemd integration
- Multiple instances for HA

Monitoring:
- Health check endpoints
- Response time metrics
- Error rate tracking

Best Practices:
- Implement retry logic
- Use exponential backoff
- Cache non-critical data
- Log all API interactions

Common Pitfalls & Solutions

Issue 1: High Latency
- Solution: Connection pooling, batch operations

Issue 2: Stale Data
- Solution: Polling frequency tuning

Issue 3: Authentication Failures
- Solution: Verify token, check ACLs

Issue 4: Overwhelming Controller
- Solution: Implement rate limiting, caching

Integration Frameworks

Monitoring Systems:
- Prometheus exporters
- Grafana integration
- Custom collectors

Job Submission:
- Workflow engines (Nextflow, Snakemake)
- Container orchestration (K8s)
- Cloud integration

Analytics:
- Job history analysis
- Resource utilization reports
- Forecasting models

When to Use Each Method

Use CLI when:
- Quick manual operations
- Simple scripts
- One-off queries

Use libslurm when:
- Maximum performance needed
- Complex, real-time monitoring
- Embedded applications

Use REST API when:
- Remote access required
- Cross-language integration
- Scalable systems
- Network-based solutions

Case Study: Monitoring Dashboard

Scenario: Real-time HPC cluster monitoring
Implementation: Python + REST API + Web UI
Benefits:
- Remote access
- Multi-cluster support
- Scalable to thousands of nodes
- Performance: < 1 second query response

Case Study: Workload Orchestration

Scenario: Multi-stage scientific workflow
Implementation: Python orchestrator + REST API
Benefits:
- Dynamic job submission
- Stage-dependent parameters
- Automatic error handling

Result: 40% reduction in manual intervention

Advanced Topics Overview

- Job dependency tracking
- Event stream APIs
- Custom accounting integration
- Multi-cluster federation
- GPU/Advanced resource scheduling

Resources & Documentation

- Official Resources:
  - Slurm documentation: https://slurm.schedmd.com
  - REST API docs: https://slurm.schedmd.com/rest_api.html
  - GitHub: https://github.com/SchedMD/slurm

- Community:
  - Slurm mailing lists
  - HPC centers
  - User groups

Key Takeaways

1. Multiple interfaces exist for different use cases
2. REST API is ideal for modern, scalable solutions
3. Python provides accessible integration
4. Plan architecture based on performance needs
5. Proper monitoring and error handling essential

Q&A / Discussion

- Questions?
- Interactive demonstrations available
- Code examples provided
- Follow-up resources available