SSH-Backed API Performance Case Study

Anagha Jamthe[1(✉)], Mike Packard[1], Joe Stubbs[1], Gilbert Curbelo III[2], Roseline Shapi[3], and Elias Chalhoub[1]

[1] Texas Advanced Computing Center, Austin, TX, USA
{ajamthe,mpackard,jstubbs,echalhoub}@tacc.utexas.edu
[2] California State University of Monterey Bay, Marina, CA, USA
gcurbelo@csumb.edu
[3] Mississippi Valley State University, Itta Bena, MS, USA
rshapi@usapglobal.org

Abstract. We establish that SSH is a viable transport mechanism for API access to HPC resources. In this paper, we study the performance and scalability properties of SSH using various SSH libraries (Python, Java, Linux command line client). We consider SSH daemon configuration changes that improve the API scalability significantly. We observe that, for the memory and CPU resources available on the test machines, our SSH-based API performs sufficiently well until a certain threshold of requests per second (RPS). At 90 RPS, 99% of the requests finish in less than two seconds. At 50 RPS, almost 90% of the requests finish in one second, which shows that the API is responsive enough under these loads. However, as the number of concurrent requests increases past 100, we see a gradual increase in time to complete requests. We perform load tests for the SSH API by sending bursts of concurrent connections and continued sustained connections over time and observe an acceptable responsiveness from the remote systems in both cases. With this study we conclude that SSH performance is sufficient for API access to computational HPC resources.

Keywords: Application Programming Interface (API) · High Performance Computing (HPC) · High Throughput Computing (HTC) · J2SSH Maverick · Paramiko · ssh2-python · Locust · Jetstream

1 Introduction

HPC computing and storage resources are increasingly being accessed via web interfaces and HTTP APIs as opposed to direct command-line interface. All cloud providers, including: Amazon AWS [1], Google Cloud Platform [2], and Microsoft Azure [3], provide such services. At the Texas Advanced Computing Center (TACC), Tapis Cloud APIs [4,5] currently enable 14 different official

W. Gao et al. (Eds.): Bench 2019, LNCS 12093, pp. 295–305, 2020.
https://doi.org/10.1007/978-3-030-49556-5_27

projects (a total of nearly 20,000 total registered client applications) to manage data, run jobs on the HPC and HTC systems, and track provenance and metadata for computational experiments. When jobs are run on HPC machines, hundreds of files are needed to be transferred between storage and execution systems for staging input data and archiving output data. The underlying APIs that perform these asynchronous file transfers through SFTP are expected to securely transfer files without significant delays. In order to understand and potentially improve performance and scalability of the Tapis Files management APIs, we study performance of SSH as a protocol.

SSH (also referred to as Secure Shell) is well-known as the most secure method of authenticating and encrypting access to remote Linux systems via command line. It is a secure alternative to insecure file transfers with FTP. In this paper, we investigate whether SSH is also viable as a transport layer for simultaneous API requests to similar systems. We discuss two methods used to evaluate the API performance. First, we identify the bottlenecks with respect to memory, CPU, and I/O when a burst of simultaneous SSH connection requests are initiated by the clients. We study how tuning the SSH daemon configuration parameters can improve successful concurrent connections to the remote system. Secondly, we seek to understand the performance of our existing files management APIs and ways to optimize it for remote access to HPC resources.

In this study, we also compare the performance of various available SSHv2 implementations in Python and Java, such as Paramiko [6] and ssh2-python [7], J2SSH Maverick [8] and the Linux command line client. We then select the most suitable implementation for our SSH API design. To evaluate the performance, we calculate the total time to connect to the HPC system and execute different commands (I/O and non-I/O based), for example, "ls" command, which is primarily used for listing files with the Tapis files management API. We perform load test for our APIs by simulating realistic work loads using Locust [9], a load testing tool. Locust can simulate a multi-user API access scenario with thousands of active users, which is similar to existing Tapis files management API usage.

The rest of this paper is organized as follows. In Sect. 2 we discuss the related work and motivation behind this case study. In Sect. 3 we provide background details on the general API performance expectations, introduction to Tapis Files Management API, and survey on available SSHv2 libraries. In Sect. 4, we describe the SSH backed APIs case study design. We discuss the proposed design and experimental setup such as the VM (Virtual Machine) configurations, comparison of different SSH libraries, SSH API framework, and load test setup. Finally, we conclude this paper with our research findings and discuss the scope of extending this study.

2 Related Work

In the realm of high speed bulk data transfers and file management, solutions such as GridFTP from Globus [10], glogin [11], BBCP [12], LFTP [13],

Cyberduck [14], scp [15], rsync [16] and Kerberos kFTP [18], exist. The choice of data transfer tools highly depends on the frequency of transfers and transfer time. For example, manual scp and rsync are more suitable for 1-time transfers [17], whereas with tools like GridFTP and BBCP, faster data transfers are achievable by doing multi-stream transfers [17]. Most of these transfer tools however have some limitations in terms of their cost and configuration complexity. Our study, on the other hand, leverages SSH directly, to securely login to remote host and leverages SFTP to transfer files, which involves minimal installation, and is easy to maintain. To the best of our knowledge, none of the prior studies evaluate the performance and scalability of SSH for multiple concurrent connections to the HPC resources, which makes this study one of a kind and important. With this case study we intend to build our next generation File Management APIs, which can provide high performance and scalability for accessing the HPC resources.

3 Background

3.1 General API Performance Expectations

Domain scientists and researchers work with distributed HPC systems to run their high performance computing jobs. They need to access data, which might be distributed across several systems present at different geographical locations. These users mostly use command-line utilities and APIs, and expect them to be responsive enough to view the job output and transfer files without significant delays. Interactive command line users are accustomed to system responsiveness fluctuating due to load on a shared system. It is not uncommon to have hundreds of individual users interactively logged in to a login node of a HPC system for accessing their resources.

With API access, it can be less obvious that one is using a shared system, so users may have an elevated expectation of responsiveness. In general, average responsive times for APIs vary substantially, but our anecdotal experience suggests that average response times exceeding one or two seconds can lead to a perception that the API is "slow". Leading cloud providers, such as Amazon and Google have described a similar phenomenon for web page load times, where above one or two seconds, the user experience is significantly impacted [19].

API quality can be determined from a combination of critical factors such as performance/responsiveness, availability/uptime and correctness. An API contract explicitly covers all the related implementation details and what to expect when the caller calls a function. However, the performance and correctness contract are always implicit and success of any software that uses the API largely depends on whether these expectations are met. Since remote calls over SSH are combined with other usage on the system, responsiveness is also affected by system load. SSH overhead usually represents a fractional amount of this delay. Often, API users benefit from being behind some sort of asynchronous queuing system (e.g. RabbitMQ [20]) that returns a response to the user before actually finishing the command. This can mitigate the responsiveness issue for end users.

3.2 Tapis Files Management APIs

Tapis [4] is an open source, NSF funded Application Programming Interface for hybrid cloud computing, data management, and reproducible science. Tapis leverages standards-compliant, open source technologies and community promoted best practices to enable users to manage data, execute research software, and share results with collaborators and colleagues. Tapis has been in production as the middleware that currently supports a number of community science gateways. It is a multi-tenant, cloud-native distributed system. All services within the platform run as Docker containers, orchestrated as a set of microservices. The Tapis files management APIs, which is one of the core services, allows management of data across multiple types of storage systems such as Linux, cloud (a bucket on S3), and iRODS. It supports traditional file operations such as directory listing, renaming, copying, deleting, and upload/download that are traditional to most file services. It also supports importing files from arbitrary locations, metadata assignment, and a full access control layer allowing to keep the data private, shared, or made publicly available. To fulfill the above operations, the current Tapis Files management API uses the J2SSH Maverick library's SSHv2 implementation.

3.3 SSH Libraries

Th choice of SSH library during API design can have a significant impact on the overall API performance, specifically for handling burst of concurrent requests. For these reasons, we evaluate different SSH implementations in this study and choose the most suitable library for SSH API development. Some of the available SSHv2 implementations in Java and Python are listed below:

- J2SSH Maverick is a complete Java implementation of the SSH2 client. We conduct performance benchmark studies using this library as it is an integral part of existing Tapis files management service.
- Paramiko is a Python implementation of SSHv2 protocol. It has been widely used in automation applications such as Ansible [23].
- ssh2-python is a new SSH library written in Python which is based on the libssh2 C library. Based on prior research, ssh2-python shows improved performance in session authentication and initialization. It is almost 17 times faster than Paramiko in performing heavy SFTP reads [24].

4 SSH API Case Study

With the distributed nature of HPC computing, there is a pressing demand for developing highly responsive file management APIs, with performance expectations that can efficiently support several concurrent users. The aim of this case study is to investigate how to develop such APIs by answering research questions below:

4.1 Research Questions

- RQ1: Is SSH a viable transport mechanism for API access to HPC resources?
- RQ2: Can we improve the performance and scalability of APIs to support multiple concurrent users by studying SSH as a protocol?

4.2 Research Design

It is not uncommon to have several concurrent users accessing the HPC resources with the Tapis files management APIs. Several web portals and CLI users access the shared HPC resources concurrently and expect the APIs to be responsive. In order to determine whether we can design a SSH backed API that meets the performance and responsiveness expectations, we need to demonstrate the feasibility of using SSH as a transport mechanism. In this study, we propose to evaluate the performance of parallel SSH connections to remote systems using bursts of simultaneous connections and continuous sustained connections over time. Benchmarking the SSH API performance by simulating multi-user request loads is a critical part of this case study. In order to demonstrate the improvements in handling concurrent SSH requests at the server, we conduct tests by modifying the default values of MaxStartUps and MaxSessions in the sshd_config file on the server. We measure the number of successful SSH connections established when a burst of concurrent requests are made by the clients during each test run. This data best describes the number of concurrent user requests that can be handled successfully at a given time for a given load. Similarly, by measuring the performance metrics "time to connect" and "time to execute commands", for commands such as "ls" and "uptime", on the remote system during a burst of simultaneous connections and continuous sustained connections over time, we can determine if SSH APIs are responsive. In the following sections, we describe the experimental setup and proof of concept SSH API framework and summarize our findings for the research questions above.

4.3 Experimental Setup

For the proposed experiments we set up three virtual machines and evaluate SSH API performance under different loads.

VM Configurations. Tests are launched from a single client VMWare virtual machine–referred to as *SSHClient*–with 2 CPU cores and 8GB of memory running CentOS 7.6 Linux. Each test then connects to one of two different server virtual machines; one of them is a VMWare virtual machine–referred to as *Taco* here–which has 2 CPU cores and 2 GB of memory, running CentOS 7.6 Linux. The other one is a Jetstream [21] Openstack virtual machine–referred to as *Jetstream* here–which has 2 CPU cores and 4 GB memory, running CentOS 7.5 Linux operating system. We selected these VMs because they are relatively small in size and represent what a developer might readily have access to. We used VMWare because it is TACC's standard VM deployment system, and Jetstream because it has a different network, IO, and hardware configuration.

Load Test Setup. In order to conduct load tests on our API, Locust, an open source load testing tool, is used to "swarm" the API and simulate concurrent multi-user requests. To set up Locust, we create a configuration file that defines the task of a simulated user, and what information to POST to the API. Other configurations includes setting wait times and sending information. Along with this, Locust provides a graphical interface where we could launch and see different request/response information such as minimum/maximum/average/median response times to connect to the server and run the commands.

Selecting SSH Library Implementation. As discussed, the choice of SSH library implementation for the API design affects API performance. We run benchmark tests to evaluate the API performance using two SSHv2 implementations: J2SSH and ssh2-python. We measure the total time to connect and run commands on both the VMs–"Taco" and "Jetstream"–from "SSHClient". On a successful connection, either "uptime" or "ls" (directory listing containing 10,000 files) is run and total response time is measured. The total time measured for 10, 100, and 500 concurrent requests provides a baseline for selecting SSH library implementation. From multiple test runs, almost seven times faster response times are seen with ssh2-python library on both Taco and Jetstream, executing the "uptime" command as compared to J2SSH implementation. Similarly, a ten to twelve times faster response is seen on both VMs, executing "ls" command on a successful connection with ssh2-python. Based on these evaluations, ssh2-python seems to be an appropriate choice for our prototype SSH API design.

4.4 SSH API Framework

We developed an SSH API using Python's Flask library. This study serves as a proof of concept to evaluate if SSH can be used as a viable transport mechanism for file management APIs to access HPC resources. With this API, users can securely connect to remote HPC resources and execute commands on the server. To make use of the API, a user first makes a one-time API call to save their sever connection credentials, including credential name, host name, user name, and an encrypted private key. These data get stored in a MySQL database for later use. Once credentials have been saved, the user can use the other API endpoints to execute different commands on the server. Table 1 describes the various SSH API endpoints and methods allowed. We note that the API in its current form is unathenticated; as a part of future development, we are working on adding authentication via JSON Web Tokens (JWT) [25]. The API would use a JWT included in the request to verify that the API call is coming from an authorized user.

This API provides an abstraction for accessing the remote HPC resources without having to use the command line interface. Most importantly, the SSH API is vital in testing the reliability of the SSH daemon server's ability to handle multiple requests at once. Using the load testing tool Locust [9], we simulate realistic multi-user requests.

Table 1. SSH API endpoints

Name	GET	POST	Endpoint	Description
Home Page	X	-	sshapi/v2/	App Welcome Page
Credentials	X	X	sshapi/v2	Manage Credentials
Commands	-	X	sshapi/v2/[cred_name]	Run command via credential
Load test	-	X	sshapi/v2/load	Load test

4.5 Findings

In this section we present our findings for RQ1 and RQ2 and discuss how answering these questions helps us in developing a prototype for SSH-backed files management APIs, using ssh2-python library.

RQ1: With Locust, which is a distributed load testing tool, we test how many concurrent users, the SSH API is capable of handling. We simulate a multi-user API access scenario, where burst of SSH connection requests are made to each of the remote servers: "Taco" and "Jetstream" with the SSH API. The user behavior and task sets are defined in the locustfile.py. Locust spawns one instance of the Locust class for each simulated user. The user task calls the commands API enpoint, which connects to the remote host with the credential name defined in the POST request. Once a successful connection is established the command specified in the same POST request is executed. The min and max wait attribute values defined in the locustfile.py determine how much time the user will wait between each API call. In our test setup we have defined a single user task of calling the SSH API. Figure 1 shows the load test results obtained. The X axis shows the percentile of successful connections, whereas the Y axis shows the response time measured in milliseconds.

We observe that, for the memory and CPU resources available on the test machines, our SSH-based API performs sufficiently well until a certain threshold of requests per second (RPS). In fact, we expect that available server memory, not SSH, is the first limiting factor up to a certain threshold of requests per second (RPS). At 90 RPS, 99% of the requests finish in less than two seconds. At 50 RPS, almost 90% of the requests finish in one second, which shows that the API is responsive enough under these loads. For the most part, as the number of requests per second increased from 10 to 90, we saw a gradual increase in response time. The 60 RPS trial was the outlier, where performance was in fact worse than in the 90 RPS trial. Understanding this outlier will be part of a future study. Considering the existing loads that our current file management system API handles, we believe that being able to handle 90 RPS in less than two seconds is more than acceptable.

Figure 2 shows the average response times in seconds for both VMs, Taco and Jetstream, using the ssh2-python implementation. For each trial, total time to connect and run one of the commands, "uptime" or "ls", for directory size

of 10,000 files, is computed. Performing a directory listing is one the most common use cases of the Tapis files management API and is therefore necessary to benchmark its performance. For these tests, we have created a nested directory structure, which includes 10,000 files to simulate the files listing call with heavy load. The average response time is computed for a set of 10 trials for each 10, 100 and 500 RPS. Similar average response times are observed on both Taco and Jetstream, when "uptime' and "ls" commands are executed at 10 RPS. At 100 and 500 RPS, a gradual increase in the average response time is seen for both the VMs, running either of the commands. However, the average response time does not vary much, when compared on both systems for 100RPS or less. We propose to study the variability of measurements (as defined, for example, in [26]), which can further explain the overall API stability as a part of our future study. With this study, we can conclude that SSH is viable transport mechanism for API access to HPC resources and can be integral part of our next generation Files management API design.

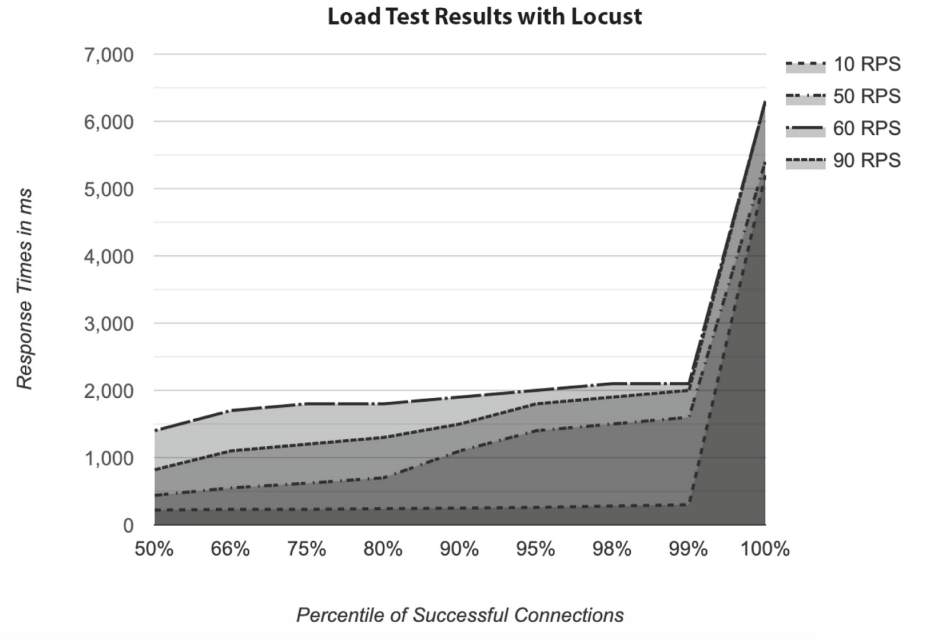

Fig. 1. Load test results for SSH API

RQ2: In order to answer our second research question, we study whether modifications to SSH daemon configuration at the server improves the scalability of the API, thereby allowing larger numbers of simultaneous connections. We made the following settings changes in the sshd_config file at both the servers:

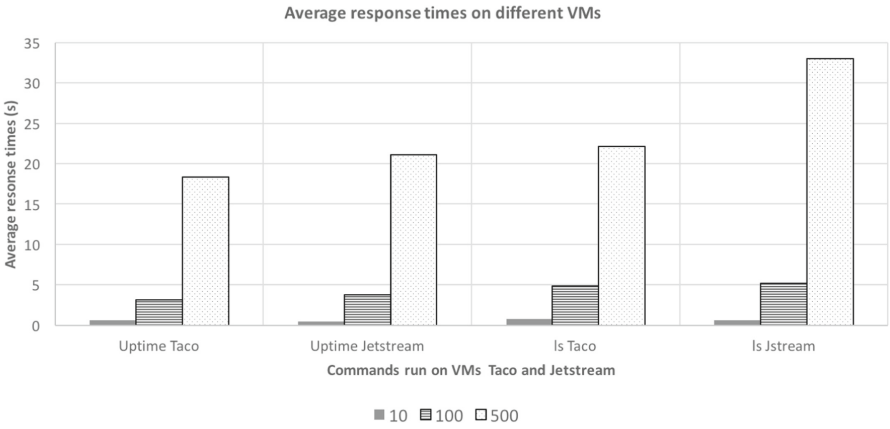

Fig. 2. SSH average response times with SSH2-Python

- MaxStartups: Specifies the maximum number of concurrent unauthenticated connections to the SSH daemon. Default is `10:30:100`, we used `3000:30:3000`, where:
  - 3000 is the number of unauthenticated connections before we start dropping
  - 30 is the percentage chance of dropping once we reach 3000 (increases linearly for more than 3000)
  - 3000 is the maximum number of connections at which we start dropping everything

  and
- MaxSessions: Specifies the maximum number of open shell, login, or subsystem (e.g. SFTP) sessions permitted per network connection. Default is `10`; we used `3000`.

With these settings, we were able to successfully connect to the server with even higher concurrent request rates. Therefore, by making these changes we were able to improve the overall scalability of the SSH APIs.

5 Conclusions

In this case study, we proposed to design a SSH-backed API towards answering two research questions: can SSH be used as a viable transport mechanism for API access to HPC resources, and can SSH performance and scalability be improved tweaking the SSH daemon parameters at the server. We tested SSH load performance in two ways: using bursts of simultaneous connections, and continuous sustained connections over time. In both cases, we observed an acceptable responsiveness from different Linux systems. This demonstrates that, in addition to its other advantages, SSH performance is sufficient for API access to HPC resources. With this study, we conclude that ssh2-python can potentially be used for our next generation Files Management API implementation.

6 Future Work

In the near future, we plan to expand the number of destination hosts to test against more diverse system configurations. We also plan to evaluate the possibility of modifying client behavior so that the server does not require sshd_config modifications. This could be done by pooling connections or taking advantage of other optimizations. We also plan to study the variability of measurements which will determine the overall performance of the SSH API for various HPC systems.

Acknowledgments. This work was made possible by grant funding from National Science Foundation award numbers ACI-1547611 and OAC-1931439. We thank the staff of TACC and Jetstream for providing resources and support.

References

1. Amazon AWS. https://aws.amazon.com
2. Google Cloud. https://cloud.google.com
3. Microsoft Azure. https://azure.microsoft.com/en-us/
4. Tapis Cloud API. https://tacc-cloud.readthedocs.io/projects/agave/en/latest/
5. Stubbs, J., et al.: Tapis: an API platform for distributed computational research. Futur. Gener. Comput. Syst. (2020)
6. Forcier, J: Paramiko: A Python Implementation of SSHv2 (2019). http://www.paramiko.org/
7. Pkittenis, ssh2-python (2019). https://github.com/ParallelSSH/ssh2-python
8. Pernavas, R.: J2SSH API. http://freshmeat.net/projects/sshtools-j2ssh
9. Locust. https://locust.io
10. Allcock, W., Bester, J., et al.: Secure, efficient data transport and replica management for high-performance data- intensive computing. In: Proceedings of the IEEE Mass Storage Conference, pp. 13–28, April 2001
11. Rosmanith, H., Kranzlmuller, D.: glogin - a multifunctional, interactive tunnel into the grid. In: Fifth IEEE/ACM International Workshop on Grid Computing (GRID 2004), pp. 266–272 (2004)
12. BBCP. https://github.com/slaclab/bbcp
13. LFTP. https://lftp.yar.ru
14. Cyberduck. https://cyberduck.io
15. SCP. https://linux.die.net/man/1/scp
16. Rsync. https://linux.die.net/man/1/rsync
17. Data transfer basics and best practises. https://princetonuniversity.github.io/PUbootcamp/sessions/data-transfer-basics/PUBootCamp_20181031_DataTransfer.pdf
18. Kohl, J., Neuman, C.: The kerberos network authentication service (V5). Request for Comments (Proposed Standard) RFC 1510, Internet Engineering Task Force, (Web site: www.ietf.org)
19. Einav Y., Amazon Found Every 100ms of Latency Cost them 1% in sales https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/
20. RabbitMQ. https://www.rabbitmq.com

21. Stewart, C.A., et al.: Jetstream: a self-provisioned, scalable science and engineering cloud environment. In: Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure, 2792774, pp. 1–8. ACM, St. Louis (2015). https://doi.org/10.1145/2792745.2792774
22. Towns, J., et al.: XSEDE: accelerating scientific discovery. Comput. Sci. Eng. **16**(5), 62–74 (2014). https://doi.org/10.1109/MCSE.2014.80
23. Ansible. https://github.com/ansible
24. SSH2 Python Comparison with Paramiko. https://parallel-ssh.org/post/ssh2-python/
25. Json Web Tokens. https://jwt.io
26. Jain, R.: The Art of Computer Systems Performance Analysis: Techniques for Experimental Design Design, Measurement, Simulation and Modeling. Wiley, New York (1991)