Falcon: Fair and Efficient Online File Transfer Optimization

Md Arifuzzaman, Brian Bockelman, James Basney, and Engin Arslan

Abstract—Research networks provide high-speed wide-area network connectivity between research and education institutions to facilitate large-scale data transfers. However, scalability issues of legacy transfer applications such as scp and FTP hinder the effective utilization of these networks. Although researchers extended the legacy transfer applications to increase their performance by exploiting I/O and network parallelism, these solutions necessitate users to fine-tune parallelism level, a task that is challenging even for experienced users due to the dynamic nature of networks. In this paper, we propose an online optimization algorithm, `Falcon`, to tune the degree of parallelism for file transfers to maximize transfer throughput while keeping system overhead at a minimum. As research networks are shared infrastructures, we introduce a game theory-inspired novel utility function to evaluate the performance of various parallelism levels such that competing transfers are guaranteed to converge to a fair and stable solution. We assessed the performance of `Falcon` in isolated and production high-speed networks and found that it can discover optimal transfer parallelism in as little as 20 seconds and outperform the state-of-the-art solutions by more than 2×. Moreover, `Falcon` is guaranteed to converge to Nash Equilibrium when multiple transfers compete for the same resources with the help of its game theory-inspired utility function. Finally, we demonstrate that `Falcon` can also be used as a central transfer scheduler to speed up convergence time, increase stability, and enforce system/user-level resource limitations in shared networks.

Index Terms—Online transfer optimization; throughput optimization in research networks; file transfer tuning; high-speed networks.

1 Introduction

The rapid evolution of instrument technologies along with growing storage and compute capacities have led to an unprecedented increase in the amount of data generated by scientific applications. For example, advancements in high-throughput genome sequencing technology increased output size per single run from around 5 MB in 2006 to more than 5 GB in 2018, a three-order magnitude increase in 12 years. This increase in data sizes when combined with the fact that science projects are increasingly *distributed* and *collaborative* paved the way for the formation of high speed research networks such as Internet-2 and ESnet. These networks provide dedicated network connectivity between research and education institutions to ensure that science workflows have sufficient bandwidth to move large-scale data in a timely manner.

Researchers also proposed the ScienceDMZ architecture to separate research and internet traffic at the campus level to minimize the friction for science workflows [1]. Despite these efforts, most transfers in research networks fall short of reaching beyond a few gigabit-per-second throughputs mainly due to a lack of scalable data transfer applications. Legacy file transfer applications (e.g., FTP, scp) are designed for low-speed internet transfers (i.e., in the order of megabytes per second), thus, they fail to perform well at high speeds. Specifically, they create a single transfer thread and network connection to transfer one file at-a-time. This in turn leads to suboptimal performance as parallelism is essential to reach full utilization in today's HPC clusters and research networks. For example, while research networks offer up to 100 Gbps in network bandwidth, a single TCP connection is typically limited to 30 Gbps due to memory and CPU limitations. Similarly, the throughput of individual storage servers in parallel file systems is less than 30 Gbps even with disk striping. Therefore, it is necessary to transfer multiple files simultaneously to improve resource utilization and achieve high transfer throughput.

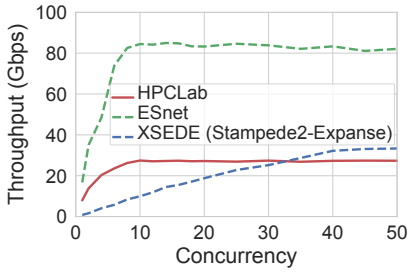


Fig. 1. Although transferring multiple files concurrently helps to improve transfer throughput significantly, the optimal value of concurrency depends on many static and dynamic factors.

To demonstrate the impact of parallelism on file transfer performance, we transferred 500 × 1 GiB files in one local area (HPCLab) and two wide-area (XSEDE and ESnet) networks with different levels of concurrency and measured the transfer throughput. HPCLab nodes have a RAID storage system (using 4 NVMe SSDs) and are connected with a 40 Gbps local area network. ESnet nodes also have a RAID storage system (using 8 NVMe SSDs) and are connected

- *Md Arifuzzaman and Engin Arslan are with the University of Nevada, Reno*
  E-mail: marifuzzaman@unr.edu, earslan@unr.edu
- *Brian Bockelman is with Morridge Institute for Research*
  E-mail: bbockelman@morgridge.org
- *Jim Basney is with National Center for Supercomputing Applications*
  E-mail: jbasney@ncsa.illinois.edu

with a 100 Gbps wide-are area network with 89ms round trip time. Finally, Stampede2 [2] and Expanse [3] supercomputers use a Lustre parallel file system to store files and are connected with a 40 Gbps network and 46ms round trip time. Figure 1 shows that transferring one file at-a-time (i.e., $concurrency = 1$) approach obtains around 8 Gbps, 18 Gbps, and 2 Gbps throughput in HPCLab, ESnet, and XSEDE networks, respectively, mainly due to read or write I/O limitations. However, transferring multiple files simultaneously increases the throughput to around 30 Gbps in HPCLab and XSEDE networks and to more than 80 Gbps in ESnet, corresponding to a $3 - 15x$ increase compared to the baseline configuration. Despite its significant impact, finding the optimal concurrency level is challenging due to large search space and the prohibitive cost of exhaustive profiling. Deriving accurate analytical models for production systems is also nearly impossible due to administrative and operational challenges associated with collecting real-time performance metrics from all components of end-to-end transfers across multiple clusters and network domains. Specifically, HPC clusters and research network providers typically do not share real-time performance metrics for system resources such as utilization of storage nodes and network packet loss rates.

Although using a predefined large concurrency value such as 30 may mostly solve performance issues, it will unnecessarily overwhelm I/O and network resources in addition to causing fairness issues between competing transfers [4]. As a result, *while concurrency is essential to increase resource utilization to achieve high performance for file transfers, it is challenging to find its optimal level due to depending on many static (e.g., network and file systems settings) and dynamic (e.g., network congestion) factors that are hard to capture.* Previous work proposed heuristic [5], [6], supervised learning [7], [8], and real-time optimization [9], [10] models to find a solution to this problem. Despite yielding higher throughput than the baseline configuration (i.e., one file at a time), heuristic models fail to offer robust performance in all networks as they cannot incorporate dynamic transfer conditions, such as background traffic into their prediction models. Supervised learning models when combined with real-time probing can make precise predictions; however, deriving an accurate model requires a large amount of historical data to be collected in a wide range of transfer conditions (e.g., dataset, background traffic, etc.), which could take weeks or months. Real-time optimization algorithms can discover the optimal settings in the runtime; however, existing solutions in this area suffer from long convergence times and fail to provide fairness and stability guarantees in shared environments.

In this paper, we introduce `Falcon` which combines a game theory-inspired utility function with state-of-the-art online search algorithms to swiftly discover the optimal concurrency level. As opposed to previous solutions which solely focus on increasing the throughput of transfers, `Falcon` innovates a novel utility function to discover "just-enough" concurrency that can obtain near-optimal transfer performance while lowering system overhead and improving fairness. Our extensive evaluations in various network settings with up to 100 Gbps bandwidth show that `Falcon` achieves $2 - 6\times$ higher throughput compared to state-of-the-art solutions. *Furthermore, `Falcon` is the first file transfer optimization algorithm that guarantees fairness among competing transfers by incorporating regret into its utility function.* Finally, `Falcon` lends itself to being used as a web-hosted central transfer scheduler in shared networks to facilitate the adoption of `Falcon` by novice users, allow resource usage policies to be imposed, and improve the converge time and stability of competing transfers. In summary, our major contributions are as follows:

1) We innovate a game theory-inspired utility function to evaluate the performance of different concurrency levels. The proposed utility function rewards high throughput and penalizes increased system overhead to strike a balance between performance and cost.
2) We implement Online Gradient Descent (OGD) algorithm to scan the solution space for optimal concurrency level swiftly. OGD allows adjusting step size based on the gradient of previous search steps to identify the optimal solution quickly.
3) We show that when `Falcon` is deployed as a central transfer scheduler to manage transfer tasks, it offers (i) a convenient way to use `Falcon`, (ii) improved stability in the presence of multiple competing transfers, and (iii) an ability to define and enforce resource usage policies.
4) We evaluate the performance of `Falcon` (both for decentralized and centralized versions) in multiple isolated and production high speed networks with up to 100 Gbps bandwidth and show that it is able to discover optimal transfer settings in as little as 20 seconds thereby attaining close-to-maximum transfer throughput in all network settings. We also show that incorporating a penalty term in the utility function of `Falcon` ensures that competing transfers converge to a fair and stable state.

Compared to our previous work [11], this paper explores a new and important direction that involves using `Falcon` as a central scheduler (i.e., contribution #3). We demonstrate that `Falcon` central scheduler performs as efficiently as its decentralized counterpart in addition to offering several key benefits that would not be possible with the decentralized approach. As an example, the central scheduler improves the stability of data transfers when multiple transfers share the same bottleneck resource. Moreover, the central scheduler lets system administrators prioritize some transfers over others so that time-sensitive streaming flows can run along with batch transfers without worrying about degrading the quality of service metrics.

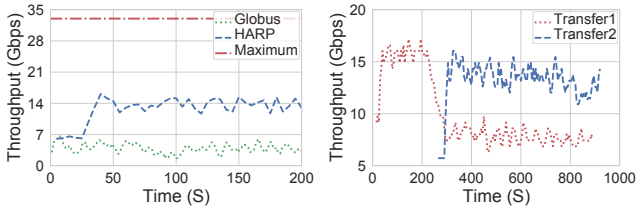
2 Related Work

As the trend toward more data-intensive applications continues, developers and users must invest significant effort into efficiently moving large datasets between distributed sites. For example, it is estimated that cosmology simulations will create 50 PiB data monthly, part of which needs to be moved between collaborating institutions for processing and archival purposes, which requires roughly 1TiB/hour transfer rates [12]. Even though high-speed networks with

hundred-gigabit-per-second bandwidth are available, many users still experience difficulty in reaching high speeds in these networks mainly due to failure to utilize available end systems and network resources efficiently. Most of the existing work on transfer optimization has been at the transport layer, including designing new transport protocols [13], [14], [15]. For example, Google released BBR [15] to address performance problems in the TCP congestion control algorithm. However, since file transfers in high-speed networks often face I/O performance limitations, improving the performance of congestion control algorithms is itself not sufficient to overcome the performance issues in today's high performance networks.

A common way to address performance problems for file transfers is tuning application-layer transfer settings such as pipelining [16], parallelism [17], concurrency [10], buffer size [9], and striping [18]. These parameters when tuned carefully can significantly improve the end-to-end data transfer performance by addressing the most commonly faced performance bottlenecks such as lack of I/O parallelism, TCP buffer size limitations, and lots of small files problem. Among them, pipelining, parallelism, and concurrency are the most effective in addressing the majority of underlying performance issues [5], [10]. There have been several attempts to tune some of these application-layer parameters to maximize transfer throughput using heuristic models [6], historical data-based models [8], [7], and stochastic approximation [9]. However, they either fail to achieve good performance or require significant up-front work in order to perform well.

Globus [19] is a widely-adopted data transfer service used by many HPC clusters to schedule large data transfers in high-speed networks, with an OAuth-based security model similar to our security model for the `Falcon` central scheduler. Globus uses fixed settings to application-layer transfer parameters such as pipelining, parallelism, and concurrency. To avoid overwhelming end system and network resources, it sets the concurrency for each transfer to a small value (between 2 and 8 in most cases), thus failing to achieve high performance in most networks as presented in our evaluations. Moreover, it does not change the transfer settings once they are set, so it is also not responsive to changes in network conditions. Yun et al. proposed ProbData [9] to tune the number of parallel streams and buffer size for memory-to-memory TCP transfers using stochastic approximation. ProbData is able to explore the near-optimal configurations, but it takes several hours to find a solution, which makes it impractical to use as most transfers in high-speed networks only run for a few minutes [20]. Yildirim et al. proposed PCP [10] to tune the values of application layer parameters in the runtime. It uses a simple hill-climbing method to scan a subset of search space; thus, it is neither fast nor precise. In previous work, we proposed a heuristic [6] and a historical data-based [8] models to determine the transfer settings for file transfers that can maximize the throughput. As detailed in our evaluations, HARP requires large-scale, up-to-date historical data collected under various background loads, datasets, and transfer settings to derive predictive models that can guarantee high performance. However, collecting such large-scale data from production networks is infeasible.



(a) Single Transfer Performance (b) Competing Transfers (HARP)

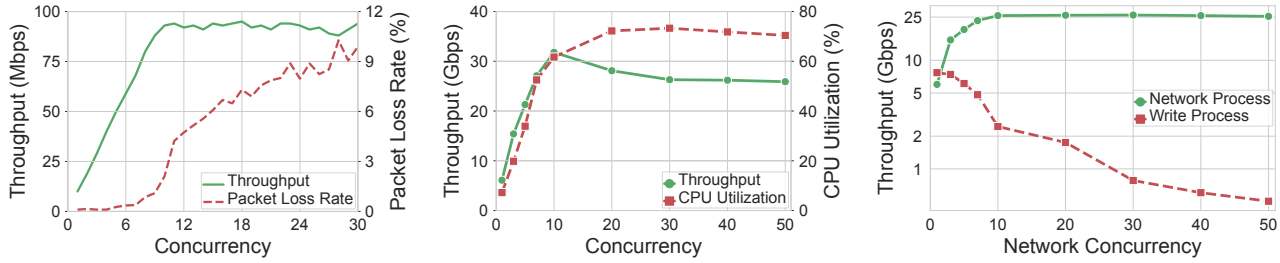
Fig. 2. State-of-the-art transfer optimization solutions, Globus [5] and HARP [8] are unable to achieve high performance due to lack of adaptive parallelism (a). They also fail to guarantee fairness between competing transfers (b).

3 Motivation

Although file transfer optimization has been studied extensively in the past, we identify two major issues with existing solutions as *failure to guarantee high performance* and *unfair resource sharing* in the presence of competing transfers. We argue that these two issues cannot be addressed through simple extensions of existing solutions due to potential side-effects; *increased overhead on system resources* in particular.

Poor Transfer Performance: Despite the availability of high-speed networks and high-capacity parallel file systems, existing file transfer applications and services fail to take full benefit of these resources due to a lack of adaptive resource parallelism. Figure 2(a) shows the performance of two state-of-the-art file transfer solutions when transferring 1024×1 GiB dataset between Stampede2 and Expanse supercomputers. Globus [19] uses fixed and mostly suboptimal transfer settings, hence achieving less than 6 Gbps throughput. On the other hand, despite attaining higher throughput than Globus, HARP [8] also underperforms by yielding less than 50% of maximum throughput. This is mainly because HARP lacks historical data in this network, so it makes predictions based on transfer logs gathered in other networks with different characteristics. While collecting new data to re-train HARP for this specific environment will improve its performance, it can take weeks to months to collect a sufficient amount of training data, which is not a feasible option for most production networks.

Unfair Resource Sharing: In the presence of multiple independent file transfers competing for limited available resources, convergence to a fair and stable state is desired. This well-known concept in game theory requires competing agents to either use the same fixed strategy or periodically update their strategy using a symmetric, strictly concave utility function [21]. Thus, transfer optimization algorithms that employ fixed strategies (e.g., Globus) guarantee that transfers will converge to a stable state; however, they fail to adapt to changing conditions since the optimal strategy is heavily dependent on current network conditions such as the number of competing transfers. On the other hand, solutions that tune the transfer settings only once at the beginning of the transfer (e.g., HARP) fail to provide fair resource sharing between the competing transfers as "*late-comers*" will have an unfair advantage by choosing a transfer setting that favors them. Yet, extending existing transfer solutions to run the optimization process periodically to adapt to dynamic conditions will not work either since their throughput-oriented utility functions (i.e., higher transfer



(a) Packet Loss (b) High CPU Utilization (c) Write Performance degradation

Fig. 3. While concurrency is needed to increase transfer throughput, choosing an arbitrarily high value increases network congestion (a) and end host CPU utilization (b). It also adversely affects other processes running on the same end hosts due to increasing I/O contention (c).

throughput means higher utility) do not meet the strict concavity requirement of convergence to fair state. As an example, the utility, u, of a transfer task that creates n concurrent transfers (i.e., $concurrency = 4$) each obtaining t throughput[1] can be calculated by

$$u(n,t) = \sum_{i=1}^{n} t = n \times t \quad (1)$$

when the utility function is set to be linearly proportional to transfer throughput. Since fair resource sharing requires the second derivation of the utility function to be negative, u cannot guarantee fairness between competing transfers since its second derivative is 0. Figure 2(b) illustrates this behavior for HARP which utilizes historical analysis to find a transfer setting that maximizes transfer throughput [8]. It is clear that when the second HARP transfer starts, it chooses a setting that favors itself and yields nearly 50% higher throughput than the first HARP transfer.

Overburdened Network and End Hosts: A naive solution to maximize transfer throughput in high-speed networks while ensuring fairness can be implemented by using a fixed transfer setting that involves high values for the concurrency parameter, such as 30. However, high levels of concurrency can overwhelm end system and network resources by creating too many processes and network connections. To demonstrate this, we evaluate the performance of a file transfer when concurrency is set to values between 1 and 32 in a simple network where sender and receiver nodes are connected via two switches. While sender and receiver nodes are connected to switches with $1Gbps$ links, the two switches are connected with a 100 Mbps link, thus end-to-end network bandwidth is limited to 100 Mbps. We throttle disk read throughput to 10 Mbps per process to emulate the behavior of parallel file systems in which concurrent I/O access (using multiple threads) is necessary to achieve high I/O performance. Since the network bandwidth is limited to 100 Mbps, ten concurrent transfers are needed to achieve 100 Mbps aggregate I/O throughput, thereby reaching to maximum possible transfer speed. Although creating more than ten concurrent transfers does not degrade the transfer throughput considerably, it results in a significant increase in packet loss due to network congestion at the bottleneck link as presented in Figure 3(a). Packet loss is below 2% when concurrency is smaller than 10, but it increases drastically and reaches 10% for a concurrency value of 32.

In addition to increased packet loss, high concurrency values also overburden end hosts and storage systems due to creating too many processes and threads [4]. Figure 3(b) shows the relationship between transfer throughput and sender host CPU utilization. When the concurrency is set to the optimal value of 10, the transfer yields 32 Gbps throughput, and CPU utilization of transfer threads is around 60%. On the other hand, setting the concurrency to larger values not only decreases throughput but also increases CPU utilization. As an example, a concurrency value of 30 returns 26 Gbps throughput in exchange for 70% CPU utilization. Finally, we tested the impact of using an unnecessarily high concurrency value on the performance of other applications on the same node. To do so, we run a process on the transfer receiver host that writes 100GiB to a file while the transfer application is running. We then measure the transfer throughput and the execution time of the write process. Figure 3(c) shows that the transfer throughput reaches the maximum at the concurrency level of 10 at which point the write process attains 2.4Gbps throughput. While increasing concurrency does not increase the throughput of the transfer, it significantly degrades the performance of the write process due to increasing I/O contention. Specifically, the throughput of the write process drops down to less than 1Gbps when concurrency is set to 30 and 0.5 Gbps when the concurrency of the transfer is set to 50. *Consequently, there is a need for a high-speed file transfer optimization solution that can tune the level of transfer parallelism in real-time to provide* ***high-performance*** *and* ***fair resource sharing*** *with* ***minimal system overhead***. We, therefore, introduce `Falcon` that combines a game-theory-inspired utility function with state-of-the-art online optimization algorithms to address these limitations.

4 Falcon: Online High-Speed File Transfer Optimization Algorithm

`Falcon` implements online learning to find the optimal transfer settings in the runtime as illustrated in Figure 4. It adopts a black-box approach and uses sample transfers to evaluate different transfer settings. The benefit of this black-box approach is its generality and applicability to a diverse set of network systems without making any assumption about the underlying infrastructure. Such abstraction makes

1. Since each transfer thread uses the same congestion control algorithm to transfer similar size files between the same end points, they will attain similar throughput as most commonly used TCP variants (e.g., Reno, Cubic, HTCP, and BBR) guarantee fairness among competing flows as long as they all have the same round trip time [22], [15]

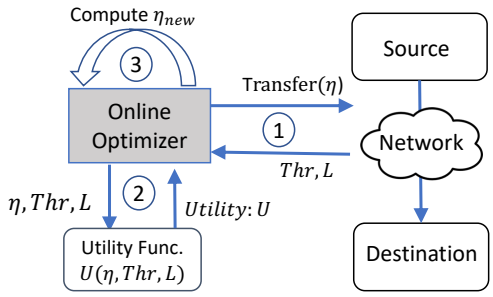


Fig. 4. Falcon uses online optimization to discover optimal transfer settings quickly. The utility function rewards high throughput while penalizing increased system overhead to ensure convergence to a fair and optimal solution.

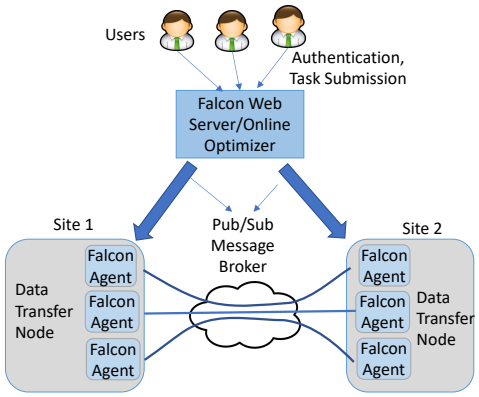


Fig. 5. Falcon can be used both as a decentralized and centralized transfer optimizer. In the centralized mode, multiple transfers can share a single online optimizer to find an optimal concurrency level for all transfers.

it possible to develop an intuitive understanding of the system conditions through simple performance metrics such as throughput and packet loss rate. Falcon first selects a transfer setting, η, and runs a sample transfer to evaluate its performance. Once the sample transfer is executed for a sufficient amount of time, we capture performance metrics and use a utility function, U, to find a scalar value that quantifies the efficacy of η. Finally, the utility value is used by an online optimization algorithm to predict a new transfer setting η_{new} that is closer to the optimal. Compared to the existing optimization algorithms for high-speed file transfers, Falcon makes two novel contributions. First, it innovates a *game-theory inspired utility function* that incorporates a regret (i.e., penalty) term for increased packet loss and concurrent transfer count to keep system overhead at a minimum and ensure **fairness among competing transfers**. Second, it implements the state-of-the-art *online optimization algorithms* (e.g., Gradient Descent and Bayesian Optimization) to quickly scan the search space and **converge to the optimal solution**.

We first apply Falcon to tune the number of concurrent transfers (i.e., $\eta = \{concurrency\}$) as it is shown to be the most effective parameter in the optimization of large-scale file transfers [8], [10], [5]. In § 5.4, we demonstrate that the utility function of Falcon can be modified to incorporate more parameters to configure multiple transfer settings simultaneously. Moreover, although we implemented a custom transfer application in Python, the online optimization module of Falcon can easily be integrated into other transfer applications such as FTP, GridFTP, and bbcp. In fact, the multiparameter optimization (§ 5.4) is evaluated using the GridFTP protocol which allows users to configure other transfer parameters such as network parallelism and command pipelining.

Falcon can be utilized in two modes as *decentralized* and *centralized* modes. In the decentralized mode, Falcon transfers act independently, so each transfer executes its own online search to discover the optimal. In the centralized mode, the online optimization module is offloaded to a central scheduler to control multiple Falcon transfers. The global optimizer still employs the same utility function as described in Equation 4 to search for optimal concurrency for all transfers as illustrated in Figure 5. It then decides on how to allocate the concurrency among the managed transfers based on defined policies. As an example, if the scheduler wants to evaluate the concurrency value of 20 when controlling four Falcon transfers with equal priority, it requests each transfer to test the concurrency value of 5. Once all four transfers finish testing the specified concurrency value, the scheduler aggregates the reports (i.e., throughput and packet loss values) from all transfers to feed them into the online search algorithm to estimate a new concurrency value.

While the decentralized mode provides several advantages such as increased resilience to single point-of-failures, improved user privacy, and better scalability; the centralized mode can be preferable due to offering higher stability, ease-of-use, ability to implement resources restrictions as follows: First, the centralized mode can significantly improve system stability compared to the decentralized mode by reducing convergence time and minimizing search attempts by individual transfers. Second, in the decentralized approach, users must log in to both the source and destination endpoints to schedule transfers, which is inconvenient and time-consuming. In contrast, the centralized scheduler simplifies this process by offering a single interface for users to initiate and monitor their transfers. Third, system administrators at individual sites may wish to define resource constraints for some users, such as maximum allowed concurrency and throughput. The central scheduler allows the implementation of such policies by adjusting its concurrency allocation decisions accordingly. In Section 5.7, we present evaluation results to demonstrate these benefits.

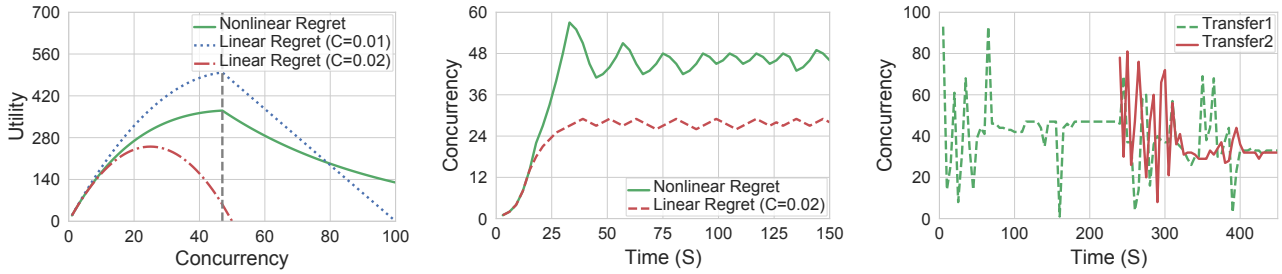
4.1 Utility Function

Utility functions need to involve a regret term to converge to a fair and optimal solution [21], [23], [24], thus Falcon incorporates packet loss into its utility function as

$$u(n_i, t_i, L_i) = n_i t_i - n_i t_i L_i \times B \quad (2)$$

where n_i is the number of concurrent files to transfer, t_i is an average throughput of each file transfer, and L_i is an aggregate packet loss rate for all concurrent transfers[2]. B is a constant coefficient that is used to determine the severity

2. Please note that concurrent transfers here refer to simultaneous transfers of the same transfer task. As an example, a transfer operation will have five concurrent file transfers when concurrency level is set to 5.



(a) Linear vs nonlinear regret for concurrency (b) Suboptimal performance of linear regret 2% (c) Suboptimal convergence for linear regret 1%

Fig. 6. Comparison of linear (Eq 3) and nonlinear (Eq 4) forms of regret for concurrency in the utility function. Linear form of regret either fails to yield high performance for single transfer ($C = 0.02$ in (a) and (b)) or causes suboptimal convergence when multiple agents compete (c). Thus, `Falcon` incorporates concurrency penalty into its utility function in a nonlinear form.

of punishment for packet loss penalty. While the value of B can be customized for specific application scenarios, we find that $B = 10$ works well with most commonly used TCP variants (i.e., TCP Cubic and Reno, and HSTCP) by keeping packet loss rate below $1 - 2\%$ while achieving over 95% network utilization. As a result, the utility function in the form of Equation 2 can be used to prevent high packet losses caused by suboptimal concurrency settings. However, it is not sufficient to avoid I/O and end host overheads. As high-performance networks with up to $40/100$ Gbps capacity are being built, transfer bottlenecks are shifting toward end hosts. For example, the network service provider of most research and education institutions in the U.S., Internet2, supports 100 Gbps connectivity to most sites and is upgrading its backbone capacity to 400 Gbps [25]. On the other hand, it is challenging, if not impossible, to attain 100 Gbps I/O throughput in production clusters due to inevitable resource interference. Moreover, most HPC clusters use data transfer nodes with $10/40$ Gbps Network Interface Cards (NICs), limiting maximum possible transfer rate to smaller values compared to network bandwidth. Consequently, little to no packet loss is observed in many production systems, necessitating an additional penalty term to limit the excessive use of concurrency. We therefore propose a cost function that penalizes the use of high concurrency by incorporating the value of concurrency into the utility function as

$$u(n_i, t_i, L_i) = n_i t_i - n_i t_i L_i \times B - n_i t_i \times n_i C \qquad (3)$$

where C is a constant coefficient that is used to adjust the rate of penalty for increased concurrency. Previous studies show that the utility functions that incorporate monotonically increasing penalty terms in linear form guarantee high performance for a single transfer and optimal and fair convergence for competing transfers (i.e., Nash Equilibrium) [23], [24]. However, we find that it is challenging to achieve both high-performance and fair and optimal convergence when penalty for concurrency is incorporated in a linear form similar to Equation 3. Figure 6 presents **estimated** utility value when C is set to 0.01 and 0.02 (nearly 1% and 2% punishment for each concurrency, respectively) when the optimal concurrency level is 48; i.e., 48 concurrent transfers are needed to reach full I/O and network utilization. When C is set to 0.02, the utility value peaks at concurrency value 25 which in turn results in low throughput. We experimentally validate this behavior by throttling I/O throughput of each process (as described in § 3) in a way that it requires 48 concurrent I/O threads to reach maximum transfer throughput. When using Equation 3 with $C = 0.02$ as a utility function, the transfer converges to a suboptimal concurrency value of 26, hence obtains 45% lower transfer throughput than the optimal. Smaller C values such as 0.01 are able to converge to optimal configurations for single transfer scenarios both theoretically (Figure 6(a)) and empirically(Figure 6(c)), but leads to suboptimal convergence behavior when there are multiple competing transfers due to increased sensitivity to measurement jitters. Figure 6(c) illustrates that although the utility function with linear penalty of 1% (i.e., $C = 0.01$) converges to the optimal solution when there is only one transfer in the system, it fails to do so when the second one joins. Although the optimal solution requires both transfers to create 24 concurrent transfers to yield maximum throughput with minimal overhead, they both settle at $36 - 38$ concurrent transfers and overburden system resources unnecessarily.

To address this issue, we test a nonlinear form of regret for concurrency as

$$u(n_i, t_i, L_i) = \frac{n_i t_i}{K^{n_i}} - n_i t_i L_i \times B \qquad (4)$$

where K is constant. As throughput improvement ratio is not directly proportional to increased concurrency (i.e., the ratio of gain starts to lower at higher concurrency values), the value of K can be tuned to require small but non-negligible gain (e.g., 1%) for increasing concurrency values. By doing so, we ensure that the utility will increase as long as non-negligible amount of throughput gain is observed and decrease upon exceeding the optimal concurrency value. Figure 6(a) and 6(b) show that the utility function that incorporates penalty for concurrency in a nonlinear form converges to the optimal both theoretically and empirically for single transfer optimization. It also converges to a fair and optimal solution when multiple transfers compete as presented in § 5.2.

When multiple `Falcon` transfers compete against each other, packet loss will stay the same for increased concurrency values. Thus, the term $1 - L_i \times B$ will follow monotonically decreasing pattern for the increasing number of concurrent transfers. Thus, the utility function given in Equation 4 is guaranteed to be concave as long as $\frac{n_i t_i}{K^{n_i}}$ is concave. It is also true when transfers are sender-limited

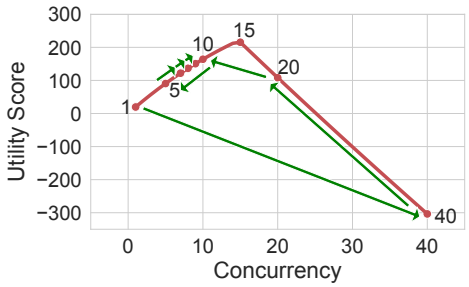


Fig. 7. Although the optimal concurrency is 15, binary search converges to a suboptimal solution 10.

(i.e., transfer bottleneck is I/O or NIC) since packet loss rate, L_i, will be nearly zero. Therefore, if the second derivative of $\frac{n_i t_i}{K^{n_i}}$ is negative (i.e., first derivative is strictly decreasing), then the utility function in the form of Equation 4 is guaranteed to be strictly concave, a condition that needs to be satisfied to converge to fair and optimal state.

Proof: Let's denote $\frac{n_i t_i}{K^{n_i}} = f(n)$, then second derivative of $f(n)$ becomes

$$f''(n) = t_i K^{-n_i} \ln K(-2 + n_i \ln K) \tag{5}$$

Since, t_i, n_i, and K are all non-negative values, $t_i K^{-n_i} \ln K$ will return greater than zero. Thus, $f''(n)$ can only be negative if the term $-2 + n_i \ln K$ is negative. Consequently, Equation 4 is guaranteed to be strictly concave as long as $n_i < \frac{2}{\ln K}$. Hence, the value of K defines the upper limit for the number of concurrent transfers, n, that can be created before $f''(n)$ moves out of the strictly concave region. Setting K to 1.01 will expect at least 1% increase in throughput to prefer higher concurrency values. It will also guarantee Nash Equilibrium as long as the optimal concurrency level is less than or equal to 200. Our experimental analysis shows that although lower K values help to increase the upper limit of the concave region, they cause stability issues in the case of competing transfers due to increased sensitivity to throughput fluctuations. We therefore set K to 1.02 (i.e., at least 2% throughput gain required for each new concurrent transfer) to strike a balance between transfer stability and reduced upper limit.

4.2 Online Search Algorithm

A naive approach to find the best transfer setting can be implemented by evaluating the performance of all possible configurations (i.e., brute-force method), but it is not a feasible method due to the large search space and expensive nature of evaluating different settings. Specifically, it takes several seconds to measure the performance of a transfer configuration due to connection establishment cost[3] as well as slow convergence of TCP transfers in high-speed wide-area networks. We tested multiple online search algorithms with different complexity including Hill Climbing, Binary Search, Bayesian Optimization, and Gradient Descent.

In the Hill-Climbing algorithm, the search process first determines the search direction to follow, then evaluates potential values in that direction one by one until the utility stops increasing at which point the search direction is reversed to initiate the search in the other direction. In the context of concurrency optimization, the search process starts with a minimum concurrency value of 1 and increments it by one as long as the utility is higher than the utility of the previous concurrency value. When the utility stops increasing, we start to evaluate lower concurrency values by decreasing the concurrency value by one. We observe that Hill Climbing takes significantly longer (up to 7× compared to Gradient Descent) to converge to the optimal. The impact of slow convergence is exacerbated when multiple transfers are executed in parallel. Although using a strictly concave function for utility calculations guarantees Nash Equilibrium between competing `Falcon` transfers, convergence time is extremely long using Hill Climbing due to its slow search speed.

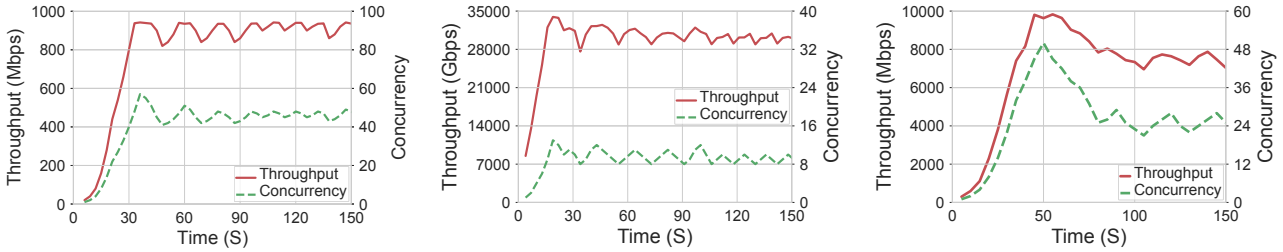
On the other hand, binary search is capable of efficiently scanning the search space and quickly converging to the optimal solution. However, the ability of binary search to find the optimal solution is not guaranteed because the search space (i.e., utility values) may not strictly increase or decrease. An illustration of this behavior can be seen in Figure 7, where the optimal concurrency is 15, but the binary search settles on 10. While it is possible to modify the binary search to continue searching until the optimal solution is discovered, this increases execution time. Moreover, the binary search may face stability issues when the concurrency value is drastically changed, such as in the above example where it increases from 1 to 40 in consecutive intervals. Since the concurrency value affects the number of transfer processes on endpoints and the number of connections in the network, testing extreme values could negatively impact other transfers in the network as well as other applications running on endpoints.

Bayesian Optimization is widely used for black-box optimizations especially when cost functions are expensive to evaluate [26], [27], [28], [29]. It aims to estimate the analytical form of black-box functions by processing observed events via surrogate models, such as Gaussian Process. It starts with a prior probability and calculates the posterior probability after an event is observed using Bayes' theorem. We show in a previous work [11] that while Bayesian Optimization is able to find the optimal solution within $4-5$ search attempts, it also faces stability issues similar to binary search due to making drastic changes in concurrency values. We, therefore, adopted Gradient Descent to quickly scan the search space and find the optimal configuration.

Online Gradient Descent (OGD) is extensively used for online convex function optimization due to its ability to adapt its step size dynamically. Since the utility function in Equation 4 is strictly concave when $n \leq 100$, we can apply OGD[4] to search for the optimal concurrency value for transfers. As OGD requires gradient (i.e., slope) calculation, we estimate it as follows: For a given concurrency value n, we test concurrency values $n + \epsilon$ and $n - \epsilon$ using sample transfers and calculate their utility values, u_1 and u_2, respectively. Then, the gradient can be approximated

3. Concurrency requires new processes and network connections to be created

4. We can convert the utility to cost function by multiplying it with -1 to apply Gradient Descent.



(a) Emulab (b) HPCLab (c) XSEDE

Fig. 8. Convergence analysis of `Falcon` in different networks when there is only one `Falcon` transfer in the network. It takes less than $20-35$ seconds to discover the optimal concurrency level in each network. As it keeps searching for the optimal to detect and adapt to changing transfer conditions, its performance fluctuates slightly after convergence.

by $\gamma = \frac{u_2-u_1}{2n\epsilon}$. Note that since concurrency can only take integer values, we use 1 for the ϵ. For example, to calculate the gradient at $n = 40$, `Falcon` will evaluate concurrency values of 39 and 41 through sample transfers and calculate γ using corresponding utility values.

Note that gradient cannot be used directly to estimate the next concurrency value as its scale is different than the scale of concurrency. Thus, we convert γ to the rate change for concurrency by dividing it to the utility of $n-\epsilon$ as $\Delta = \frac{\gamma}{u_{n-\epsilon}}$ and use it to predict next concurrency value, $n_{new} = n + \Delta$. To further improve the convergence speed while avoiding taking arbitrarily large steps due to sampling errors, we use a monotonically increasing learning factor θ to gradually adjust the steps size via $n_{new} = n + \theta\Delta$. We initiate θ to a relatively small value and increase it as long as the search moves in the same direction in consecutive intervals. Specifically, we initialize θ to 1 and increase it by one at each time step as long as the gradient is positive and reset to the initial value otherwise. `Falcon` runs Gradient Descent (GD) continuously even after it discovers the current optimal to adapt dynamic conditions. It does this by checking higher and lower values around the current optimal to find if they yield higher utility than the current value.

The dynamic nature of background load in networks and file systems requires the search to be repeated periodically to adapt to the changing conditions. Thus, we configured the OGD to keep exploring the search space throughout the transfer. Specifically, after finding an initial solution, say $cc_{optimal}$, it keeps evaluating $cc_{optimal}+1$ and $cc_{optimal}-1$ to see if any of them yield higher utility than $cc_{optimal}$. If they do, then OGD initiate a search in that direction. Note that `Falcon` uses a separate thread to gather and process performance metrics, thus the optimization process does not interfere with the transfer performance.

5 Evaluation

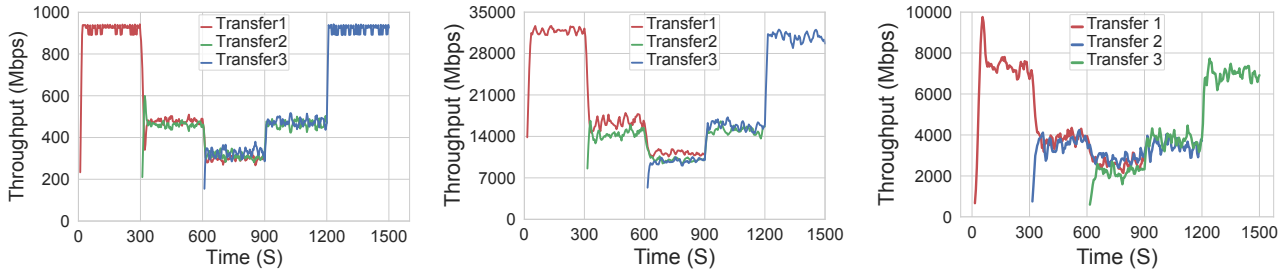
We first assess the performance of `Falcon` when it is implemented in a decentralized manner, meaning each transfer agent operates independently and uses its own optimization and scheduling modules. We run experiments in four high speed networks as listed in Table 1. We use Bridges2 [30] and Expanse [3] clusters for XSEDE experiments, which are connected via a shared high-speed network. HPCLab networks consist of two servers that are located in the same local-area network; thus delay between the hosts is less than a millisecond. ESnet testbed provides $100Gbps$ network bandwidth, 89ms round trip time, and around 80 Gbps I/O write throughput. Finally, Emulab is an emulated network testbed. XSEDE sites and HPCLab employ one or more RAID arrays in the storage system, so the use of concurrency is required to achieve full I/O performance. Since Emulab nodes have direct-attached single disk storage volumes, we throttle per process disk read throughput to necessitate concurrent I/O accesses to reach maximum performance, similar to parallel file systems. We also configured a topology in Emulab in a way that network bandwidth becomes the bottleneck once a sufficient number of concurrent transfers are created. To determine transfer bottlenecks in each testbed, we used profiling tools (e.g., iPerf [31], bonnie++ [32]) that can capture the "true" capacity of resources. We set the duration of sample transfers (i.e., evaluating the performance of a concurrency value) to 3 seconds in local area transfers and 5 seconds for wide-area transfers. Finally, we used a dataset containing 1000×1 GB files to conduct the transfers. § 5.4 presents results of `Falcon` when it's used for small and mixed datasets transfer optimizations.

Testbed	Storage	Bandwidth	RTT	Bottleneck
Emulab	SATA SSD	1G	30ms	Network
XSEDE	Lustre	10G	58ms	Disk Read
HPCLab	NVMe SSD	40G	0.1ms	Disk Write
ESnet	NVMe SSD	100G	89ms	Disk Write

TABLE 1
Specifications of test environments. OSG and Expanse sites are used for XSEDE experiments.

5.1 Efficiency for Single Transfer

We assess the performance of `Falcon` in terms of convergence speed and throughput when there is only one transfer in the network. Figure 8 presents the results for Emulab, HPCLab, and XSEDE (Bridges2-Expanse) networks. We limit the I/O performance to 20 Mbps per process for Emulab transfers and set the network bandwidth to 1 Gbps such that $45-50$ concurrent transfers are needed to reach maximum performance. The gradient descent-based search algorithm starts the search with an initial concurrency value of 2 and converges to the optimal in around 35 seconds by adjusting its step size. Upon convergence, the concurrency value bounces between 40 and 50 as it keeps evaluating higher



(a) Emulab (b) HPCLab (c) XSEDE

Fig. 9. Stability and fairness analysis of `Falcon` when multiple transfers compete for same bottleneck resources. With the help of the penalty terms in the utility function, `Falcon` agents back off when new transfers join the network, leading to fair resource sharing.

and lower values to detect any changes in the network. It converges to a solution ($cc = 10$) in around 20 seconds and attains over 25 Gbps throughput for HPCLab transfer. On the other hand, it takes around 80 seconds to converge to the optimal in the XSEDE network, for which the optimal concurrency is around 24. `Falcon` initially increases to concurrency to almost 50 as it observes a significant increase in the utility value in the previous round. However, once it notices that the throughput does not increase sufficiently when concurrency is increased from around 20 to around 50, it lowers it and stabilizes at around 24. While concurrency 49 yields around 10 Gbps, concurrency 24 yields around 7.5 Gbps. Hence, `Falcon` prefers concurrency 24 as it yields 75% of maximum throughput using 50% less concurrent transfers.

5.2 Convergence Analysis for Competing `Falcon` Transfers

We next assess the efficiency and fairness of `Falcon` when multiple transfers share a bottleneck network or I/O resource. Note that this is different from concurrency in a way that concurrency refers to transferring multiple files for the same transfer task whereas competing transfers refer to independent transfer tasks submitted by different users. We assume that all agents (i.e., transfer tasks) use the same utility function as in Equation 4 and employ the same gradient descent search algorithm. While the network is the bottleneck for competing transfers in Emulab and XSEDE transfers, write I/O is the bottleneck for HPCLab transfers. In all three cases, independent (i.e., decentralized) `Falcon` transfers reach maximum utilization when running alone, but they back off when other transfers join as shown in Figure 9. For example, while *Transfer 1* in XSEDE attains around 7.5 Gbps throughput when it is the only transfer in the network, it yields around $2.5-3$ Gbps throughput when second and third transfers join. Similarly, when second and third `Falcon` transfer agents join in HPCLab (Figure 9(b)), they can quickly seize their fair share (i.e., $12-13$ Gbps for two transfers and $7-8$ Gbps for three transfers). Moreover, when one of the transfers completes, the remaining one(s) quickly claim the available resources and sustain high resource utilization.

One can possibly attribute this fair resource sharing between competing `Falcon` agents to the adaptive nature of TCP congestion control algorithms, but Figure 10 shows that it is indeed because of `Falcon` dynamic concurrency adjustments. It presents concurrency values for `Falcon` agents

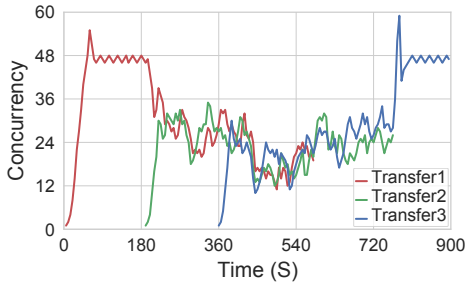


Fig. 10. `Falcon` agents reduce their concurrency values when new transfers join to ensure fairness among competing transfers.

when they compete against each other in Emulab, where the bottleneck link capacity is 1 Gbps and the I/O limit per process is 20 Mbps. When there is only one transfer in the system, it quickly converges to the optimal concurrency value of 48 to attain 1 Gbps throughput. Once the second transfer joins, the first transfer reduces its concurrency to the $20-33$ range to let the second transfer claim its fair share. Note that even if fair resource sharing can also be achieved when both transfers use the concurrency value of 48, it will result in higher packet loss despite obtaining the same throughput as illustrated in Figure 3(a). When the third transfer joins, they all select concurrency values around $10-23$ to make sure that total concurrency is large enough to fully utilize available resources yet not too high to cause high overhead on the network, end hosts, and file systems. Moreover, `Falcon` agents can also quickly notice the termination of competing transfers and increase their concurrency accordingly to claim available network bandwidth.

5.3 Comparison to State-of-the-Art

Figure 11 compares the single transfer performance of `Falcon` against two state-of-the-art file transfer optimization solutions and a static approach for the transfer of 1 TB dataset that consists of $1,000 \times 1$ GiB files. Globus [19] is a web-based transfer service based on the GridFTP protocol [33]. It is widely used to schedule large file transfers between HPC facilities. It relies on heuristic or predefined settings to tune the value of concurrency along with other transfer parameters such as parallelism and pipelining. HARP [8], on the other hand, uses historical data to derive regression models that can estimate transfer

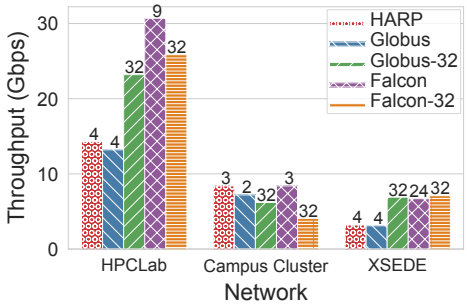


Fig. 11. Performance comparison of `Falcon` against the state-of-the-art solutions in different networks. The concurrency values on top of the bars correspond to the concurrency values used by applications.

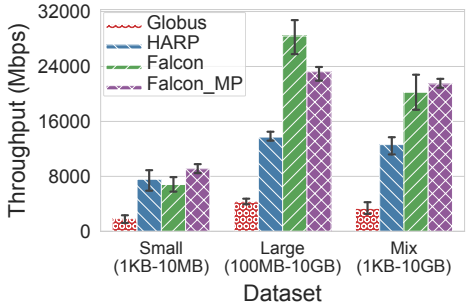


Fig. 12. Performance evaluation of `Falcon` for multiparameter optimization.

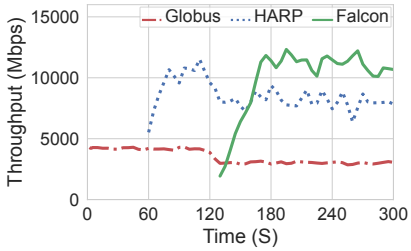


Fig. 13. Behaviour of `Falcon` when competing against non-`Falcon` transfers. It grabs its fair share plus unused capacity of the network to increase utilization without being too aggressive.

throughput based on the values of concurrency, parallelism, and pipelining parameters. We observe that while Globus is too conservative when selecting the number of concurrent transfers to minimize system overhead, HARP can be too aggressive to maximize throughput. Although HARP can reconfigure the concurrency in the runtime to adapt to changes, its performance is inherently limited to historical observations. We also evaluated the performance of two static solutions, Globus-32 and Falcon-32, that use a fixed high concurrency value of 32 for the transfers. They differ in terms of underlying transfer applications used to implement concurrency. Globus-32 relies on GridFTP protocol whereas Falcon-32 uses socket programming to execute file transfers. They demonstrate the performance of a simple method of using a predefined high concurrency value for all transfers to maximize the throughput.

Globus underperforms significantly in all three networks as it attains $2.3\times$, $1.16\times$, and $2.16\times$ less throughput than `Falcon` in HPCLab, Campus Cluster, and XSEDE transfers, respectively. This is mainly because Globus uses a predefined, suboptimal solution concurrency which falls short of maximizing I/O and network throughput. On the other hand, HARP yields similar throughput as with `Falcon` in Campus Cluster transfers while obtaining almost half of it in HPCLab and XSEDE transfers. Globus-32 outperforms Globus in HPCLab and XSEDE networks as high concurrency values result in higher I/O throughput despite worsening I/O contention. On the other hand, since the optimal concurrency is small in Campus Cluster, setting concurrency to a very high value increases I/O contention significantly thereby lowering effective I/O throughput. Falcon-32 yields 5% higher throughput than `Falcon` in XSEDE since the optimal concurrency (around 24) is very close to the concurrency value used by Falcon-32. However, its throughput is 18% and 47% lower than `Falcon` despite using a high concurrency value. This is mainly because of I/O performance degradation as a result of increased I/O contention. That is, effective I/O throughput degrades significantly as the number of competing processes increases. As a result, using a predefined high-concurrency value for file transfers increases system overhead (Figure 3) and leads to degraded throughput in some networks due to causing I/O contention.

5.4 Multiparameter Optimization

Although concurrency is the most effective parameter in increasing transfer throughput due to offering both I/O and network parallelism, additional transfer parameters such as network *parallelism* and command *pipelining* can be tuned to further improve transfer performance, especially for long-running transfers. *Parallelism* determines the number of concurrent network connections to transfer a file, which can be helpful to improve the performance in case the transfer dataset consists of very few large files. *Pipelining*, on the other hand, sends multiple file transfer commands to source and destination servers back-to-back so that the transfer of the next file can start immediately after the previous one is completed. *Pipelining* is mainly helpful when transferring many small files by eliminating the pauses between consecutive transfers. In terms of system overhead, *parallelism* can overburden network resources by creating too many concurrent flows. On the contrary, *pipelining* has negligible impact on system resources since it merely caches the name of the next file(s) to transfer. Therefore, we modified the cost function in Equation 4 to incorporate a penalty term for *parallelism* as

$$u(n_i, t_i, L_i) = \frac{(n_i \times p_i)t_i}{K^{n_i \times p_i}} - n_i t_i L_i \times B \qquad (6)$$

where p_i refers to the level of *parallelism*. Note that *parallelism* can be used together with concurrency, so $n_i \times p_i$ is used to calculate the total number of network connections created for a given transfer. As an example, if *parallelism* is set to 4 while concurrency is set to 5, then `Falcon` will transfer 5 files simultaneously and use 4 network connections for each file. We utilized conjugate gradient descent to optimize the search process as it provides an efficient solution for multiparameter optimization problems [34].

Figure 12 demonstrates the performance of different algorithms for multiparameter optimization to optimize transfers between Stampede2 to Expanse clusters. We evaluate the performance for the transfer of three different datasets

as *small* that contains files whose size range between 1 KiB to 10 MiB for a total of 120 GiB, *large* which contains files whose size range between 100 MiB to 10 GiB with a total of 1 TiB, and *mixed* that includes all files in *small* and *large* datasets with a total size of 1.2 TiB. We observe that Falcon yields up to 30% higher throughput when used to tune (*concurrency, parallelism, and pipelining* altogether (Falcon_MP) compared to its performance when tuning only *concurrency* (Falcon) for *small* and *mixed* datasets. This can be attributed to the importance of command *pipelining* when the dataset contains very small files. On the other hand, it results in 18% decrease in overall throughput for the *large* dataset which can be attributed to two reasons. First, the utility function for multiparameter optimization (Equation 6) is not strictly a concave function; thus, there is no guarantee that it will converge to the optimal solution. Second, multiparameter optimization takes a significantly longer time (up to 3× longer) to converge to a solution compared to single parameter optimization, causing more time to be spent in the search phase during which the transfer throughput is typically lower than the convergence throughput.

5.5 Friendliness Towards Non-Falcon Transfers

To evaluate the Falcon's friendliness to Globus and HARP, we run Falcon along with the others for the transfer of a 1.1TiB dataset (consisting of files whose size are between 100MiB and 10GiB) between Stampede2 and Expanse clusters. When the Globus transfer is started, is selects the concurrency value of 2 and obtains 4.9 Gbps throughput. Then, we initiate the HARP transfer which creates 11 concurrent transfers and attains 10.5 Gbps, which does not affect the performance of the Globus transfer as end-to-end transfer capacity is more than their cumulative throughput. Finally, we start the Falcon transfer at around 120s. The Falcon transfer increases its concurrency gradually and converges to 16−18, which returns 12−13 Gbps throughput as shown in Figure 13. Although it evaluates higher concurrency values and observes an increase in throughput, the improvement rate does not meet the desired level (i.e., nearly 2% for every concurrency value) once aggregate utilization is close to the capacity. As a result, it affects the performance of Globus and HARP transfers only marginally (around 15 − 20%). Therefore, it is fair to say that Falcon "plays well" in the presence of non-Falcon transfers by utilizing the spare capacity and avoiding aggressive behaviour against the competing flows.

5.6 Comparison Against a Static Solution

One may argue that using a fixed setting (e.g., a fixed concurrency value of 4) for a transfer may result in better overall throughput, especially for short transfers, due to Falcon's suboptimal performance during the search phase. In other words, the cost of searching for the optimal may outweigh the gain if a transfer does not last long enough. Figure 14 illustrates this by comparing the throughput of online search and a static solution (aka fixed concurrency). We pick the value of concurrency for the static solution somewhere between the optimal and Falcon's initial value such that the static solution will yield a better throughput than Falcon in

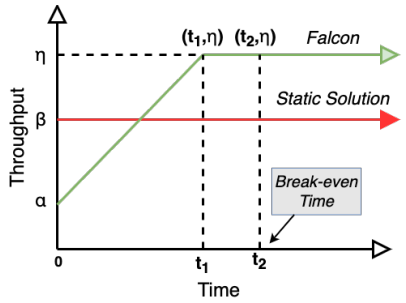


Fig. 14. Falcon requires transfers to run long enough to attain higher overall throughput than using a fixed, suboptimal concurrency value.

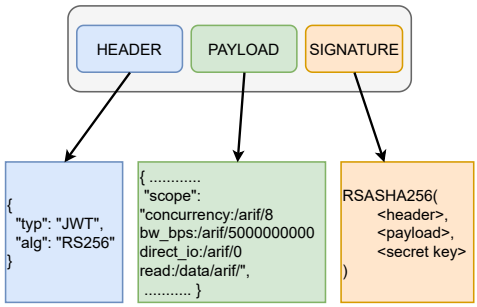


Fig. 15. Sample JSON Web Token structure. We extended the payload with transfer-specific scope attributes.

the search phase but worse than Falcon after it discovers the optimal. We believe this is a fair assumption since the optimal concurrency value is dependent on many factors that cannot be estimated ahead of time; thus we assume that the user will set a value such as 4 that is expected to return a reasonable performance in most scenarios while not imposing too much overhead to the system. On the other hand, Falcon starts with a small concurrency value (i.e., 1) and converges to the optimal using OGD. Hence, the average throughput of Falcon will be lower than the static setting until it finds the optimal and makes up the difference. Apparently, "the break-even time" depends on several factors, including the throughput of fixed concurrency, the throughput of Falcon at the beginning as well as upon converge, and the duration of the search phase. Equation 7 presents a condition that must satisfy to reach the break-even point. The break-eve time, t_2, can be calculated as

$$t_2 = \frac{(\eta - \alpha) \times t_1}{2 \times (\eta - \beta)} \tag{7}$$

where α and η are Falcon's initial and convergence throughput, β is the throughput of the static solution, and t_1 is the OGD's convergence time.

Please note that although Falcon converges to the optimal (i.e., η) almost at an exponential rate (as can be seen in Figure 8) with the help of gradient descent algorithm, we simplify the problem by assuming that it converges at a linear rate. Also, the break-even time can be reached before Falcon converges the optimal (i.e., $t_2 < t_1$) if difference $\frac{\eta-\alpha}{\eta-\beta}$ is less than 2; i.e., the difference between Falcon's convergence throughput and the throughput of the static solution ($\eta - \beta$) is larger than the difference between the throughput of the static solution and Falcon s initial throughput ($\beta - \alpha$). We validated the accuracy of this estimation in HPCLab where $t_1 = 20$s, $\alpha = 7.9$Gbps, $\eta =$

27.5Gbps, and $\beta = 20.5$Gbps for a fixed concurrency value of 4. Based on these values, Equation 7 returns $t_2 = 28s$. Our experimental results show that Falcon attains similar throughput as the fixed concurrency of 4 in $25-30$ seconds. Similarly in ESnet, t_1 is around 15 seconds, $\alpha = 16.5$Gbps, $\eta = 84$Gbps, $\beta = 48.5$Gbps when using concurrency value of 4 for the static solution. Equation 7 returns $t_2 = 14s$ while experimental results obtain $t_2 = 14-15s$. Hence, Equation 7 provides a fairly close estimation of minimum necessary duration for Falcon to outperform a static solution. One can translate the break-even time to dataset size by multiplying it with the throughput of the fixed concurrency method, β. In HPCLab and ESnet, minimum data sizes correspond to approximately 27GiB and 84GiB, respectively, which are relatively small numbers compared to the bandwidth of these networks.

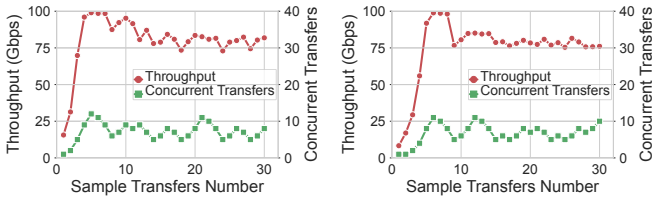
5.7 Falcon as a Central Scheduler

As mentioned in Section 4, the centralized mode allows user to submit their transfers using a cloud-hosted web server. As authentication/authorization is an important component for such a web-based transfer scheduler, we implemented the widely adopted authorization delegation protocol OAuth 2.0 [35]-based authentication using JSON Web Tokens (JWT). OAuth lets resource servers issue bearer tokens (e.g., JWTs) with minimal privileges to perform specific tasks such as retrieving user information or reading a file from the file system. Doing so minimizes attack surfaces for bad actors and resource consumption malpractices by authorized users. Upon receiving a JWT token (either directly from the end point or through an identity federation service such as CILogon), Falcon Web Server passes the token to Falcon agent running on the data transfer node using a pub/sub message channel. The agent can verify the authenticity of the token itself as token issuers sign JWT tokens with their private keys. It then can decode the token to access metadata and payload to check for resource limits. The unique benefit of JWT is the ability to define high-granularity user access privileges using attributes. In other words, JWTs let system administrators define complex policies to control user privileges at the time of token issuance such that user access can be controlled with great flexibility. For instance, a system administrator may define two different limits for the same user depending on the time of the transfer in a way that transfers that are scheduled for off-peak hours (e.g., night times, weekends, etc.) can be rewarded with higher throughput.

We implemented two transfer-specific attributes as *maximum throughput* and *maximum concurrency* that can be embedded in JWTs to control the throughput of transfers. Figure 15 presents a sample token issued with bandwidth limitation to throttle transfer throughput. We implemented an application-level approach to apply throughput limits, which periodically checks the transfer throughput and puts transfer threads into short sleeps (in the order of $100s$ of milliseconds) if their throughput is higher than the permitted rate.

5.7.1 Performance Analysis

Figure 16 demonstrates performance comparison for decentralized and centralized versions of Falcon when there is a single transfer in the system. We run transfers in ESnet testbed with $100Gbps$ bandwidth, 89ms RTT, and 80 Gbps I/O write throughput. We can observe that both centralized and distributed approaches are able to converge to the optimal in around 10 seconds. We observed similar behavior in other testbeds in terms of converging to the optimal solution as quickly as decentralized Falcon transfers. Thus, we can say that the centralized approach does not suffer from increased convergence times because of separating transfer executor (i.e., Falcon agent) and online optimizer components. Figure 17 shows the throughput of three competing flows when they are scheduled using decentralized and centralized Falcon implementations. Clearly, the centralized implementation does not only reduce convergence time but also minimizes throughput fluctuations. The primary reason behind this is the inverse relationship between the speed of convergence and the number of participants in non-cooperative multiplayer games [36]. In other words, although both decentralized and centralized implementations are guaranteed to converge to a fair and optimal solution, convergence speed is slower when agents are independent (i.e., decentralized). Therefore, scheduling and controlling transfers using a central server offers improved convergence speed and stability over a decentralized implementation.



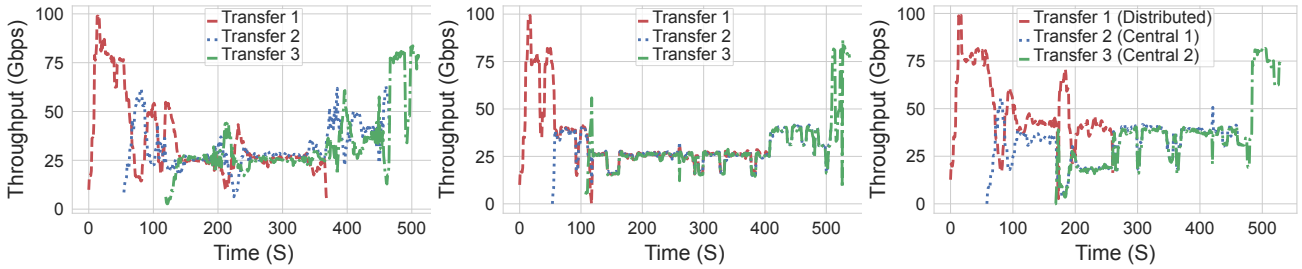
(a) Decentralized (b) Centralized

Fig. 16. Performance comparison of centralized and decentralized Falcon implementations when managing a single transfer. They both reach maximum network utilization using as minimum concurrency as possible.

We also assessed the impact of running centrally managed Falcon transfers along with decentralized Falcon transfers in Figure 17(c). In the example, *Transfer 1* is running a decentralized implementation while *Transfer 2* and *Transfer 3* are controlled by a central Falcon scheduler. A key observation we make here is that the decentralized Falcon agent (*Transfer 1*) grabs half of the total bandwidth even though there are three transfers running simultaneously. This is due to using a single optimizer for all centrally managed Falcon transfers, which effectively acts as a single decentralized Falcon transfer. Although it's possible to modify the utility function of the central Falcon server to incorporate the number of managed transfers to attain higher throughput, we defer it as a potential area for future investigation.

5.7.2 Policy Enforcement

We next demonstrate the policy enforcement capability of the central Falcon scheduler. Figure 18 shows the throughput of three transfers when bandwidth limitation is defined in the token of one of the transfers. The token for *Transfer 1* task specified bandwidth limitation of $5Gbps$, so the Falcon agent running at the source site throttles the



(a) Decentralized Competition (b) Centralized Competition (c) Decentralized and Centralized Competition

Fig. 17. Convergence analysis for (a) decentralized and (b) centralized `Falcon` implementations when managing multiple transfers. Centrally managed `Falcon` transfers converge to a fair and optimal solution quickly and experience fewer level of throughput fluctuations compared to the decentralized `Falcon` transfers. Additionally, centrally-managed transfers share resources fairly when competing against the decentralized `Falcon` transfer (c).

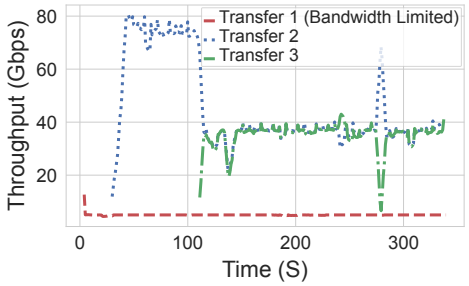


Fig. 18. `Falcon` agents can enforce throughput limitations defined in authorization tokens.

throughput accordingly even if the fair share of the transfer is higher. On the other hand, tokens of *Transfer 2* and *Transfer 3* do not include any limitations, so they share the available resources equally. We also implemented the concurrency attribute in tokens to limit how many concurrent transfer threads can be created for a transfer task. Different from throughput limitations, the concurrency limitation provides an opportunity to manage available compute capacity on transfer nodes. While it is possible to extend the decentralized `Falcon` implementation to realize and enforce resource limitations, the centralized version lends itself to this mission much better as it is designed to only work with tokens to run a transfer.

6 Conclusion And Future Work

File transfers in high-speed networks require end-to-end parallelism to efficiently utilize available resources. However, determining the optimal level of parallelism is a challenging task as suboptimal solutions can lead to underutilization or overwhelmed network and file systems. Previous work in this area implemented heuristic and supervised learning solutions both of which fail to satisfy high resource utilization while inducing low overhead to end systems and networks. To address this problem, we introduce `Falcon` that combines a novel utility function with state-of-the-art online optimization techniques to guarantee high performance, fair resource sharing, and minimal overhead. Specifically, `Falcon` innovates a novel utility function that rewards high throughput while penalizing for increased packet loss and the number of active concurrent processes. It also utilizes an online gradient descent algorithm to scan search space efficiently. The experimental results show that `Falcon` yields up to 6× higher throughput compared to state-of-the-art solutions while keeping its overhead at a minimum. More importantly, `Falcon` converges to a fair and stable state in the presence of multiple independent transfers. We also demonstrate that `Falcon` can be used as a central transfer scheduler to increase the convergence time and stability of competing transfers in addition to enforcing access policies defined by system administrators.

In the future, we plan to evaluate the performance of `Falcon` for emerging congestion control algorithms such as BBR [15] to check the feasibility of developing a congestion control algorithm-agnostic solution. Moreover, we aim to explore cross-layer optimization solutions to tune application- (e.g., the number of concurrent transfers) and transport- (e.g., loss and delay tolerance) layer parameters together to remove redundancies and increase the system stability.

Acknowledgement

The work in this study was supported in part by the NSF grants 2145742, 2007789, and 2209955.

References

[1] E. Dart, L. Rotman, B. Tierney, M. Hester, and J. Zurawski, "The science dmz: A network design pattern for data-intensive science," *Scientific Programming*, vol. 22, no. 2, pp. 173–185, 2014.

[2] "Stampede2," https://www.tacc.utexas.edu/systems/stampede2, 2023.

[3] "Expanse," https://www.sdsc.edu/services/hpc/expanse/, 2023.

[4] I. Alan, E. Arslan, and T. Kosar, "Energy-aware data transfer algorithms," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12.

[5] B. Allen, J. Bresnahan, L. Childers, I. Foster, G. Kandaswamy, R. Kettimuthu, J. Kordas, M. Link, S. Martin, K. Pickett, and S. Tuecke, "Software as a service for data scientists," *Communications of the ACM*, vol. 55:2, pp. 81–88, 2012.

[6] E. Arslan, B. Ross, and T. Kosar, "Dynamic protocol tuning algorithms for high performance data transfers," in *European Conference on Parallel Processing*. Springer, 2013, pp. 725–736.

[7] M. S. Z. Nine and T. Kosar, "A two-phase dynamic throughput optimization model for big data transfers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 2, pp. 269–280, 2020.

[8] E. Arslan, K. Guner, and T. Kosar, "Harp: predictive transfer optimization based on historical analysis and real-time probing," in *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 2016, pp. 288–299.

[9] D. Yun, C. Q. Wu, N. S. V. Rao, Q. Liu, R. Kettimuthu, and E. Jung, "Data transfer advisor with transport profiling optimization," in *2017 IEEE 42nd Conference on Local Computer Networks (LCN)*, 2017, pp. 269–277.

[10] E. Yildirim, E. Arslan, J. Kim, and T. Kosar, "Application-level optimization of big data transfers through pipelining, parallelism and concurrency," *IEEE Transactions on Cloud Computing*, vol. 4, no. 1, pp. 63–75, 2015.

[11] M. Arifuzzaman and E. Arslan, "Online optimization of file transfers in high-speed networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–13.

[12] J. Borrill, E. Dart, B. Gore, S. Habib, S. T. Myers, P. Nugent, D. Petravick, and R. Thomas, "Improving data mobility & management for international cosmology: Summary report of the crossconnects 2015 workshop," 2015.

[13] D. Leith and R. Shorten, "H-tcp: Tcp for high-speed and long-distance networks," in *Proceedings of PFLDnet*, vol. 2004, 2004.

[14] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, B. Godfrey, and M. Schapira, "{PCC} vivace: Online-learning congestion control," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 343–356.

[15] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "Bbr: Congestion-based congestion control," *Queue*, vol. 14, no. 5, p. 50, 2016.

[16] N. Freed, "SMTP service extension for command pipelining," http://tools.ietf.org/html/rfc2920.

[17] T. J. Hacker, B. D. Noble, and B. D. Atley, "Adaptive data block scheduling for parallel streams," in *Proceedings of HPDC '05*. ACM/IEEE, July 2005, pp. 265–275.

[18] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The globus striped gridftp framework and server," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2005, p. 54.

[19] "Globus," https://www.globus.org, 2021.

[20] Z. Liu, R. Kettimuthu, I. Foster, and N. S. Rao, "Cross-geography scientific data transferring trends and behavior," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2018, pp. 267–278.

[21] P. Thaker, M. Zaharia, and T. Hashimoto, "Learning and utility in multi-agent congestion control," *optimization*, vol. 24, no. 10, pp. 11–18.

[22] Y.-T. Li, D. Leith, and R. N. Shorten, "Experimental evaluation of tcp protocols for high-speed networks," *IEEE/ACM Transactions on networking*, vol. 15, no. 5, pp. 1109–1122, 2007.

[23] E. Hazan, "Introduction to online convex optimization," *Foundations and Trends® in Optimization*, vol. 2, no. 3-4, pp. 157–325, 2016.

[24] M. Zinkevich, "Online convex programming and generalized infinitesimal gradient ascent," in *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, 2003, pp. 928–936.

[25] "Internet2 Next Generation Infrastructure Update," https://internet2.edu/internet2-next-generation-infrastructure-update-29-packet-nodes-connected-by-40-400g-links-gdt-install-completes-on-schedule/, 2021.

[26] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient global optimization of expensive black-box functions," *Journal of Global optimization*, vol. 13, no. 4, pp. 455–492, 1998.

[27] A. Žilinskas and J. Žilinskas, "Global optimization based on a statistical model and simplicial partitioning," *Computers & Mathematics with Applications*, vol. 44, no. 7, pp. 957–967, 2002.

[28] A. Zhigljavsky and A. Zilinskas, *Stochastic global optimization*. Springer Science & Business Media, 2007, vol. 9.

[29] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2015.

[30] "Bridges-2," https://www.psc.edu/resources/bridges-2/, 2023.

[31] "iPerf3," https://github.com/esnet/iperf, 2023.

[32] "Bonnie++," https://linux.die.net/man/8/bonnie++, 2021.

[33] W. Allcock, "Gridftp: Protocol extensions to ftp for the grid," *Internet Draft, Mar. 2001*, 2001.

[34] Y.-H. Dai and Y. Yuan, "A nonlinear conjugate gradient method with a strong global convergence property," *SIAM Journal on optimization*, vol. 10, no. 1, pp. 177–182, 1999.

[35] "The oauth 2.0 authorization framework," https://www.rfc-editor.org/rfc/rfc6749, 2022.

[36] S. C. Wiese and T. Heinrich, "The frequency of convergent games under best-response dynamics," *Dynamic Games and Applications*, vol. 12, no. 2, pp. 689–700, Jun 2022.

Md Arifuzzaman is Ph.D. candidate in Computer Science and Engineering at University of Nevada, Reno. He received his BS in Computer Science from the Bangladesh University of Engineering and Technology in 2016 and his MS in Applied Statistics from East West University in 2019. His research focus is modeling, optimization, and anomaly detection for high-speed data transfers.

Brian Bockelman is currently an Associate Scientist at the Morgridge Institute for Research, University of Wisconsin–Madison. His research interests are in research computing and distributed high-throughput computing (DHTC). For over a decade, he has worked with the Open Science Grid on issues in distributed high-throughput computing. He now serves as the Technology Area Coordinator, leading the evolution of the technologies used by the OSG. Within Nebraska, he served as a Key Staff Member of the Holland Computing Center from 2008 to 2019 and as an Associate Research Professor with the Computer Science and Engineering Department and worked on the CMS Project, which hosts significant computing resources at the Holland Computing Center

James Basney is a senior research scientist in the cybersecurity group at the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign. Jim's area of expertise is identity management for scientific collaborations. He is the PI of the CILogon project and co-PI of the Center for Trustworthy Scientific Cyberinfrastructure and the Software Assurance Marketplace. Jim also contributes to the LIGO, LSST, and XSEDE projects. Jim received his Ph.D. in computer sciences from the University of Wisconsin-Madison.

Engin Arslan is an Assistant Professor at the Department of Computer Science and Engineering at the University of Nevada, Reno (UNR). He received Ph.D. from University at Buffalo in 2016 and worked at National Science for Supercomputing Applications (NCSA) as a postdoctoral research associate before joining UNR. His research interests include high-performance computing and networking, edge/cloud computing, and quantum networking. His work in these areas has been funded by the National Science Foundation, the Department of Energy, Amazon Web Services, and UNR. Most notably, he was the recipient of the prestigious NSF CAREER in 2022. He serves on several committees including the review board of IEEE TPDS and the UNR Cyberinfrastructure Committee.