A Non-Invasive Approach for Realizing Resilience in MPI

Umar Kalim, Mark K. Gardner, Wu Feng
Department of Computer Science
Virginia Tech
{umar,mkg,wfeng}@vt.edu

ABSTRACT

As the computational capabilities of a supercomputer transition from petaflops to exaflops, more compute processes work concurrently to accomplish tasks, requiring more communication. This results in using an increasing number of software and hardware components, which in turn, increases the probability of abnormal events and failures. We present a solution that improves resilience against transient events in network communication.

We observe that the coupling of the session and transport semantics in implementations inhibits recovery from transient failures. Our proposal, a session-layer intermediary (SLIM), serves as a shim layer on top of the interconnect's interface and enables separation of session and transport semantics. We use Open MPI as a case study where SLIM exposes an interface to the Byte Transfer Layer framework. This approach manages transient faults with the underlying transport, by trapping and resolving them and thus not allowing them to cascade into failed MPI primitives. Preliminary results show that the introduction of SLIM delivers resilience and does so without incurring any performance impact, either in latency or throughput. In future, we plan to include other interconnects, such as OpenIB, and enable tolerance for transient network failures.

KEYWORDS

Resilience, Fault Tolerance, Message Passing Interface (MPI), Extensions, Open MPI, Byte Transfer Layer (BTL)

ACM Reference format:
Umar Kalim, Mark K. Gardner, Wu Feng. 2017. A Non-Invasive Approach for Realizing Resilience in MPI. In *Proceedings of ACM Symposium on Operating Systems Principles, Washington DC, June 26, 2017 (FTXS'17),* 8 pages.
DOI: http://dx.doi.org/10.1145/3086157.3086166

1 INTRODUCTION

Large-scale computational problems (e.g., weather simulations) that use MPI are typically structured by breaking down the job into subproblems. A set of processes is then responsible for solving one or more of the subproblems. When solutions to all subproblems are obtained, the partial results are aggregated to obtain the overall results. With such work distribution, each process' role is important. If results from all worker processes are not available the program's results will be incomplete or invalid. Therefore, if such a process fails the entire job is bound to fail — unless there are mechanisms in place to mitigate those failures [10, 19].

Therefore, mitigating the impact of *transient* or *localized* faults, which might cascade into system-wide collapse, and recovering from these failures is of significant importance. We define transient failures as fleeting events (e.g., those caused by interconnect congestion) and localized failures as faults confined to a limited set of hardware or software resources (e.g., those caused by node or process failure).

Enabling resilience for large-scale parallel computations is particularly important as we scale up from compute capabilities of petaflops to exaflops [7]. Scaling up to exaflops will inevitably involve increasing the number of compute processes working in parallel, and it is well established that as the number of cores increase, so do the number of faults [10, 28] — we can imagine the increase in points of failure with the increase in number of constituent components of an exascale supercomputer.

Researchers have investigated various dimensions of enabling resilience in MPI programs [10, 19]. These efforts include checkpoint and recovery [6, 25], user-level fault mitigation (ULFM) [5, 14, 24], process-level redundancy [9], log-based recovery [4, 27], dynamic process management [16], modified MPI semantics [11, 12], algorithm-based fault tolerance (ABFT) [3, 8], use of intercommunicators with master-worker configuration [18], transactional communication [15], and renewed MPI implementations to include resilience [13]. We will discuss some of these proposals in the related work section in § 8.

Here we propose, a session-layer intermediary (SLIM), a shim layer towards enabling resilient communications that mitigate and eventually resolve transient or localized faults, which are introduced by faulty network communication. Examples of such faults include single or multiple failures of MPI primitives caused by failing network paths or congested interconnects. We enable resilience by separating session and transport semantics, which are conflated in the implementations. We make a case that it is this conflation that results in strong coupling and thus inhibits adaptations and recovery in the face of transient events. Our proposal here is informed by our experience with enabling resilient communications for traditional network communications [22, 23], where we demonstrate the efficacy through virtual machine migrations beyond networks while maintaining connectivity [20, 22]. While our current contributions are geared towards Open MPI's TCP component [26], the same can be applied to the BTL OpenIB component. This proposal of separation of session and transport semantics for OpenIB has also garnered interest by vendors [2].

Since our proposal serves as a shim layer that envelopes the underlying interconnects and interfaces with MPI frameworks (e.g., Open MPI components [26]), all user programs would run without

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FTXS'17, Washington DC
© 2017 ACM. 978-1-4503-5001-3/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3086157.3086166

any change to their code. We do not propose any changes to the MPI standard either.

Our contributions in this paper are as follows:

- A characterization of faults as well as approaches towards mitigating faults. We also explain the assumptions necessary for enabling resilience (§ 2 & § 3);
- The integration of the SLIM with Open MPI's TCP module, and plans for future extensions to include the OpenIB module (§ 5); and
- A prototype implementation (§ 6) that exposes an API to the Open MPI's BTL framework along with an evaluation of the proposed extensions (§ 7).

2 CHARACTERIZATION OF FAULTS

To better highlight the scope of our work and contribution, we characterize the types of faults in relation to MPI.

We define faults as events that result in abnormal operation. We classify faults into two categories: *software* and *hardware* faults.

Software faults are induced by bugs in either users' source code or the MPI implementations. We do not address software faults in this paper.

On the other hand, **hardware faults** pertain to the underlying infrastructure. We further classify these as *soft* or *hard* faults. We define **soft faults** as those that do not necessarily indicate imminent failure of the MPI program, although they may cause individual ranks to fail. These faults can be detected and possibly resolved. In contrast, we define **hard faults** as those that interrupt the execution of the MPI program in a manner that results in failure, and they may not be detected. Hard faults may at times result in system-wide failures (e.g., failure of GPFS).

Here we focus on soft faults induced by network communication which include **transient** and **localized** failures. We aim to resolve transient faults — those that do not cause an MPI rank to fail — without having to trigger user-level failure mitigation (ULFM), checkpoint and restore, or other fault tolerance mechanisms. As mentioned in § 1, we define transient failures as fleeting events (e.g., those caused by interconnect congestion) and localized failures as faults confined to a limited set of hardware or software resources (e.g., those caused by node or process failure).

3 APPROACHES TOWARDS RESILIENCE

Fault tolerance for MPI is a well-studied domain [10]. As we briefly highlighted in § 1, there have been multiple notable contributions along different dimensions of this challenge — e.g., [4, 8, 9, 11, 24, 27].

3.1 Assumptions of Fault-Tolerant Methods

Although different fault-tolerant mechanisms address challenges along different (and sometimes orthogonal) dimensions, all methods tend to make certain assumptions about the process [17]. The assumptions are as follows: (1) we are able to detect the failure; (2) we have enough state information to able to recover from the failure; and (3) we are allowed to instantiate recovery mechanisms to mitigate the faults.

Taking these assumptions into account, we model the detection of communication faults, mitigation, and finally recovery process by a state-transition diagram (see Figure 1).

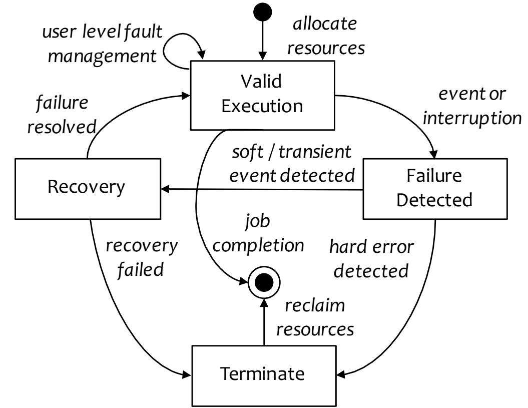

Figure 1: State diagram illustrating fault detection, mitigation, and recovery.

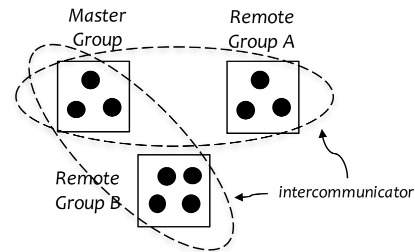

Figure 2: Master and worker configuration between groups of processes using MPI intercommunicators.

3.2 Types of Fault-Tolerant Methods

Here we briefly describe the types of fault-tolerant methods, in the context of soft faults that are transient.

Checkpoint and Restore: The foremost and most widely known method of enabling resilience is through checkpointing and restore (e.g., [6]). Checkpoint and restore entails saving the state of the program at regular intervals so that the application may be restarted from there onwards, instead of having to start from the very beginning. Most of the ULFM methods involve different forms of checkpoint-and-restore methods.

Checkpoint-and-restore methods are typically considered to be expensive ways to enable resilience, particularly at scale. Although there are scenarios where the most efficient approach is to use checkpoint and restore since the overheads depend entirely on the type of the application, the frequency of checkpoints, and the ability to restore states from various stages of the computation.

Due to the inherent nature of the approach, checkpoint and restore allows recovery from transient communication faults, whether it is in terms of recovering from a failed primitive, a failed rank, or even a total failure. However, we argue that such an approach is a

very heavyweight way to mitigate transient faults. Such faults can be resolved by much more lightweight approaches.

Methods Supported by Standard MPI Implementations: The primary reason for the misunderstanding that "if a rank fails, then the entire MPI program *will* fail" stems from the interpretation that all MPI ranks exist within one communicator. The roots of such practices originate from the use of collective operations while solving compute problems. Collective operations communicate within one communicator. To minimize the complexity of the MPI program, the typical practice is to use the default communicator, which is MPI_COMM_WORLD. When a rank within a communicator fails, or a collective operation fails, it may cascade into the failure of the program if not handled through the users' source code.

The use of intercommunicators [18] encourages compartmentalizing ranks into groups and following the manager/worker paradigm, as illustrated in Figure 2. Intercommunicators enable communication of ranks between groups. When a rank fails, for whatever reason, the fault is compartmentalized. This compartmentalization allows the user to manage complexity of the source code and handle the error gracefully using suitable methods (e.g., redundant processes [9] or dynamic process management [16]).

We argue that using intercommunicators to mitigate transient communication failures is also heavier weight than it needs to be because it incurs significant overheads.

Modifying the MPI Standard to Revise Semantics: Another rarely used approach is to revise the semantics of MPI primitives in order to enable fault tolerance. Such approaches have been tried in the past to enable fault tolerance (e.g., [11]). However these approaches have the potential of rendering user codes, written for such implementations, incompatible with other MPI implementations that comply with the MPI standard and therefore limiting application portability.

MPI Extensions: An alternate approach to revising MPI semantics is to add extensions to MPI (e.g., [17]). Doing so not only maintains compatibility of the implementation with the MPI Standard, but also allows additional functionality that applications may choose to leverage. This includes, for example, defining suitable error codes and corresponding error handlers.

We adopt the approach of adding an MPI extension to enable fault tolerance, since this not only complies with the MPI standard but also avoids the need for having to recompile MPI programs. We discuss the details further in § 5.

4 ENABLING SEPARATION OF SESSION AND TRANSPORT SEMANTICS VIA SLIM

In this paper, we leverage our experience in enabling resilient communications for traditional networking [23] to provide fault tolerance in MPI. Below is a brief overview of SLIM and how it reintroduces the distinction between session and transport from the OSI model into the network stack. Communications can continue in the face of transient faults because the session continues to exist even though the TCP connection fails. SLIM re-establishes a transport connection for the session to use and the fault is not observed by MPI due to the separation into session and transport.

4.1 SLIM as a Session-Layer Intermediary

SLIM [23] is an extensible session-layer intermediary that provides session semantics for improved resilience, among other things. SLIM defines abstractions that it uses to enable resilience and an out-of-band channel for the exchange of control messages that it uses to restore communications in the event of transient faults.

SLIM defines the endpoint, flow, and session abstractions that form the session layer. They are illustrated in Figures 3 and 4.

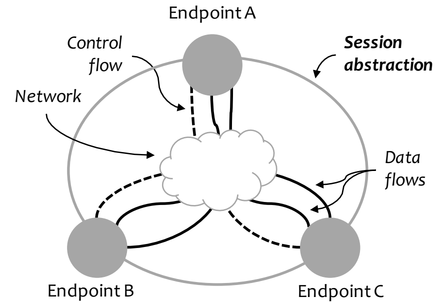

Figure 3: SLIM's session, flow and endpoint abstractions in the context of TCP/IP stack.

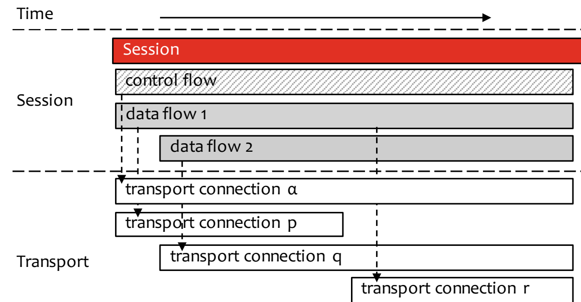

Figure 4: The flow abstractions and their mappings onto underlying transports in relation to time.

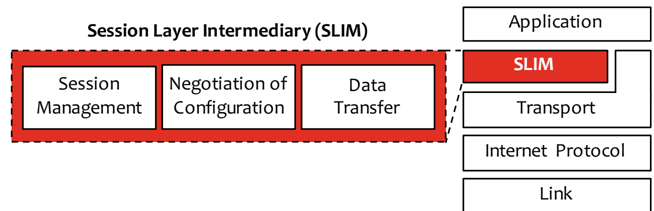

Figure 5: SLIM in relation to the TCP/IP stack.

An endpoint is defined as an entity participating in a conversation and represents a source and destination of communications.

A flow represents a data exchange between a set of endpoints. It gives a name to the concept of communication but requires mapping onto underlying transport connections before communication

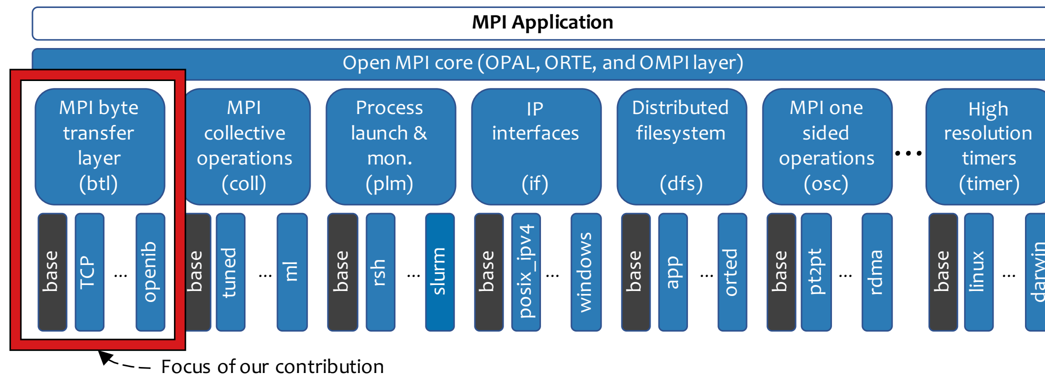

Figure 6: Open MPI Architecture (recreated from [26]).

actually occurs. Because flows are independent of transport connections, the concept of a flow can precede the creation of a transport connection and can persist after the transport connection has been closed. This separation allows us to distinguish between session and transport semantics. Doing so further enables reconfiguration of flows on the fly (and subsequently reconfiguration of underlying transports). This is illustrated in Figure 4 where a flow may be mapped onto a transport connection p and later mapped onto a transport connection r (e.g., when transport p is disrupted due to a transient fault).

A session represents the complete conversation between participants in an agreed-upon context. It encapsulates endpoints and flows that constitute the conversation and allows them to be reasoned about together. This is illustrated in Figure 3.

As Figure 5 illustrates, SLIM exposes an API while providing three sets of services to the application to assist with communication setup and management. These services fulfill three roles: 1) session management, 2) negotiation of configuration, and 3) data transfer. SLIM uses the underlying transport services to realize the session abstractions. Since its is only active during communication setup (in the beginning) and management (in case of recovery), it does not interfere with data exchange. Therefore, it has negligible impact on bandwidth and latency (see § 7).

Further details of SLIM, its backward compatible extensions to TCP, and how it is used to support modern communcation needs, such as virtual machine migration across subnets, see our prior research [21–23].

5 SLIM'S INTEGRATION WITH MPI

To begin, we chose to add our SLIM extension to Open MPI primarily because of its modular component architecture (MCA) [26]. The design of Open MPI, frameworks coupled with components, and the manner in which they are interfaced, instantiated, and used, allows us to confine our proposed extensions to select components. As illustrated in Figure 6, the focus of our contribution is on interfacing with the Byte Transfer Layer (BTL) framework. Initially our contribution is geared towards interactions with the TCP component. In future we intend to include interfacing with other components, such as OpenIB.

5.1 Open MPI Byte Transfer Layer (BTL)

The BTL framework works alongside the BTL Management Layer (BML), Point-to- Point Messaging Layer (PML) and the MCA frameworks. BTL is geared towards proving a uniform method of data transfer between participants. The data transfer may be over different interconnects. For this paper, we focus on TCP alone.

5.2 Conflation of Session and Transport Semantics by Legacy BTL-TCP

The coupling of session and transport semantics in legacy TCP causes difficulty in implementing fault tolerance for MPI communications. If there is a loss of network path between endpoints or a transport connection faces a timeout, the BTL TCP connection will drop. This will result in a failed MPI primitive, which may then cascade to a failure of the MPI program.

Thus, it is imperative that we consider session semantics, which is the notion of communication between endpoints, independent of the underlying transport or sending of data across the network.

5.3 Enabling Fault Tolerance

Decoupling of session and transport semantics allows us to handle transient failures (related to transport implementations) without inducing a failure of an MPI primitive. If a transport connection fails or faces a timeout for some reason, i.e., it was not cleanly terminated, SLIM recognizes the fault and attempts to setup a new transport connection to serve as a replacement. The flow is then mapped on to the new transport connection, the sequence spaces are synchronized to ensure that no data is lost, and communications resumes. Details of how this mapping of sequence space is managed is discussed in our prior work [23]. The correct synchronization of sequence spaces ensures that no data is lost — which may have been in flight. An illustration of such a disruption, followed by a successful reconfiguration is illustrated in Figure 4.

As we will discuss further in § 7, this decoupling incurs no overhead during fault-free operation since the indirection comes into play only during connection setup or transport recovery. During communication, latency and performance are negligibly impacted since SLIM passes packets through.

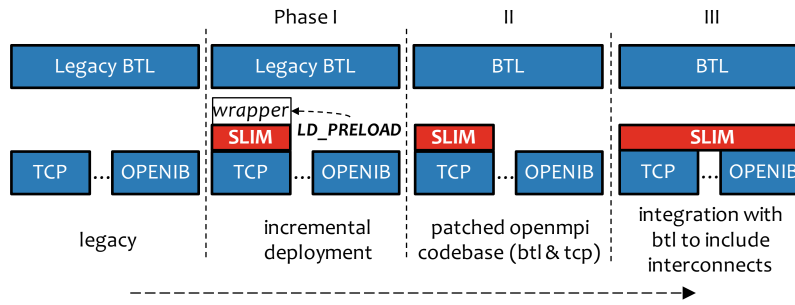

Figure 7: Incremental deployment and integration with Open MPI Byte Transfer Layer (BTL).

Transient faults vs. rank failures It is important to note that SLIM enables resilience in the face of transient failures alone. This is in contrast with the scenario where a rank fails. When a rank fails, it is likely that program state will be lost. Therefore, resuming connectivity with a replacement process may not suffice. Although there may be rare cases where the nature of the application allows such possibilities. In order to resume communication following a rank failure, a mechanism for automatic recovery must be in place — e.g., application directed recovery, message logging and replay, and so forth.

SLIM makes three attempts to recover from a transient failure. If all three attempts to resume connectivity fail, the error is escalated and may result in the MPI program's failure, unless alternate mechanisms are in place.

5.4 SLIM's Integration with BTL

Figure 7 summarizes our incremental integration and deployment approach. Initially we have implemented SLIM as a user-space library. We setup the library to intercept the Socket API calls [1] using LD_PRELOAD. The benefit of this approach is that we do not force the recompilation of either the MPI program or the Open MPI implementation. All socket interface calls go through the LD_PRELOAD wrapper, through SLIM, to the underlying TCP implementation all the while providing resilient flow implementations to BTL. In the future, we will extend the BTL implementation to interface directly with SLIM.

The goal is for session-based abstractions to be exposed to BTL while SLIM manages the mappings to underlying transports. Following along the same lines, we plan to use SLIM to enable resilience to transient faults for all underlying interconnects.As the next step, we plan to integrate SLIM with OpenIB alongside TCP.

6 PROTOTYPE IMPLEMENTATION

Here we discuss our contributions with reference to the prototype implementation.

6.1 Prototype for BTL TCP Component

The prototype for the BTL TCP component is implemented as a user-space library in C. The implementation includes 3189 lines of source code. SLIM's interface, which exposes the session primitives, is exposed to BTL and is illustrated in Figure 7. Details of SLIM's implementation that are specific to TCP are documented in [21, 23].

Interfacing with BTL and TCP SLIM serves as a wrapper around the Socket API. As part of Phase I of our development, the Socket API calls are intercepted by SLIM through LD_PRELOAD, where SLIM maintains the session, flow, and endpoint state. These abstractions are mapped onto the underlying transport (e.g., a flow is eventually mapped to an underlying socket that serves as an input and output stream).

This indirection for communications enables fault tolerance. Consider the scenario where a transport connection faces a timeout due to a transient failure. The termination of the connection results in a failed TCP socket. SLIM recognizes the abnormality and attempts to reconnect to the destination. The assumption here is that if the fault was transient, the network will be available for later communication. Spawning a new transport connection and having the flow mapped onto this new transport avoids the failure of the MPI primitive that generated the communication event. Thus, the indirection allows us to catch and recover from a failure that may have caused the program to fail.

Note that SLIM recognizes failures by evaluating the error codes returned due to failed read and writes to underlying sockets (or file descriptors).

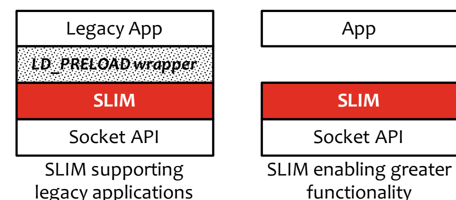

Figure 8: SLIM in relation to legacy applications and those using the library.

As part of Phase II, we plan to extend the BTL implementation and integrate SLIM without having to use LD_PRELOAD to intercept Socket calls.

Backwards Compatibility To enable backwards compatibility with legacy TCP stacks, we use custom TCP options [21, 22]. If the peer stacks are unable to exchange the custom options during the 3-way handshake, SLIM recognizes that the peer stack does not support SLIM and subsequently falls back to legacy TCP behavior.

6.2 Prototype for BTL OpenIB Component

As part of Phase III, we plan to expand the SLIM implementation to integrate the OpenIB interconnect. The plan is to have a uniform interface exposed to BTL, and have SLIM interact with the underlying BTL module when instantiated by MCA — be it TCP or OpenIB. The objective is to have a separation of session and transport semantics so that transient faults may be caught in time to allow suitable recovery (or graceful degradation), instead of having the application fail.

7 DISCUSSION

Here we discuss preliminary performance results and overheads when testing SLIM in a controlled environment as well as the concerns of deployment and interactions with the infrastructure.

7.1 Performance and Overheads

As part of the performance evaluation, we try to understand the impact the addition of SLIM has on performance, in particular latency. Since SLIM is only involved during communication setup and does not play a significant role during on going communications, we do not expect to see any significant overheads. The only role that SLIM plays during ongoing communications is an added level of indirection (i.e., flow to socket descriptor mapping), which should not incur a significant overhead, even for latency sensitive systems — e.g., MPI applications.

Figures 9 and 10 summarize the latency and throughput measurements from microbenchmarks ran on an unprimed configuration. We see that the SLIM implementation — analogous to a BTL, SLIM, and TCP component — has statistically similar performance to that of a legacy configuration with the socket API — analogous to the BTL and TCP component implementation.

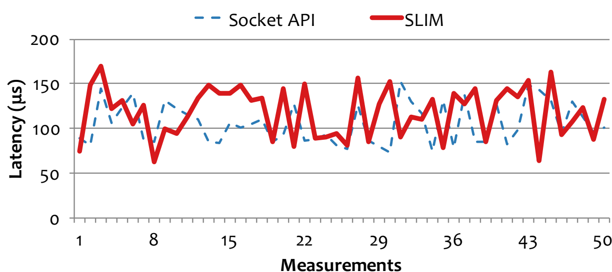

Figure 9: Trace of average latency for BTL+TCP (Socket API) vs BTL+SLIM+TCP (SLIM) using unprimed long-running microbenchmarks (1 *Gbps* link capacity, 0% loss)

The round trip latencies for application's point to point communication hover around an average of 120 μs for both SLIM and the legacy implementations. In case of bandwidth tests, the point-to-point tests are able to saturate the link up to the achievable link capacity of nearly 94%. The variability in results is due to the aggressive back-off mechanism of the TCP New Reno implementations, which were used for these tests. Newer congestion control implementations (such as TCP BBR or CUBIC may show lesser variations).

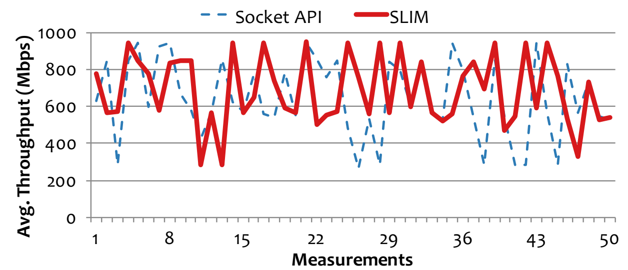

Figure 10: Trace of average throughput for BTL+TCP (Socket API) vs BTL+SLIM+TCP (SLIM) using long-running microbenchmarks (1 *Gbps* link capacity, 0% loss)

7.2 Collective Operations

While the focus of our discussions have been on a shim between BTL and TCP for point to point communications, note that SLIM supports communication between multiple participants as part of the session. This is because the abstractions have been developed to support separate session from transport semantics and therefore mitigating the limitations of underlying transport mechanisms that inhibit extensions (such as fault tolerance). However, using the multi-party session semantics of SLIM as a replacement for the COLL framework will not be efficient. This is because the multi-party session semantics are geared towards supporting participants for traditional networking and not high-performance collective operations. However, SLIM when used as part of the BTL framework — as means for reliable data transfer — performs at par with the legacy implementations.

As we move towards Phase III of our development, we plan to include the OpenIB interconnect. There we will study the impact of separation of session and transport semantics and its influence on collective operations.

7.3 Incremental Deployment and Integration with Open MPI

In § 5, we discussed our development plan and summarized it in Figure 7. We see that initially with the wrapper library, we may use LD_PRELOAD to deploy the library. This would not require rewriting or recompiling any code, whether the application or the Open MPI implementation. Doing so enables incremental deployment for the BTL TCP component. However, as we move to Phase II and III where we not only extend BTL to interface directly with SLIM, but also expand SLIM to interface with OpenIB, there would be a need to recompile and deploy the updated Open MPI implementation. The applications would not require any recompilation.

7.4 Interaction with Middleboxes

Unlike traditional networks, interaction with middleboxes is not a concern here, since data center deployments are typically devoid of middleboxes between compute nodes. Nevertheless, we've demonstrated in our prior work [20, 22, 23] that SLIM is not adversely impacted by middleboxes even if they exist in traditional networks.

8 RELATED WORK

In the last decade or so, researchers have investigated various dimensions of enabling resilience in MPI programs [10, 19]. It appears that the philosophy of fault tolerance has moved towards enabling users to mitigate the impact of faults and recover by trapping errors in user code and implementing suitable solutions. This is understandable as applications have different characteristics and the impact of the a fault may be severe for one application but not be as severe for the other. Nevertheless, we observe that all these faults typically lie in the category of what we classify as hard faults. Handling transient faults is left entirely up to the users, who tend to use methods such as dynamic process management [9] or checkpoint and restore [6] to deal with them. While this approach yields results, we've argued that they are expensive for transient faults. We focus on transient network communication fault for the BTL TCP component as a case study and suggest SLIM as a suitable solution.

Below we summarize some of the notable and representative approaches that enable fault tolerance for MPI.

Fagg et al. [11] propose FT-MPI (Fault Tolerant MPI) which augments the MPI implementation and maintains more state to determine what actions can be taken when processes in the communicator encounter failures. This changes the standard MPI semantics. For example, the MPI communicator is allowed to have different states (other than the original valid and invalid states), which are determined by the failure scenario it is experiencing. On detecting a failure in the communicator, the application can go into a failure recovery mode that is specified by the application developer. Thus, FT-MPI allows application developers to have different failure recovery modes, other than simply checkpointing and recovery. While, FT-MPI sacrifices a great deal in terms of the time-tested semantics of standard MPI, the lessons learned have been incorporated in current Open MPI implementations.

The User Level Failure Mitigation (ULFM) [5] interface has been proposed to provide fault-tolerant semantics in MPI. The interface focuses fail-stop failures only and allows application-level failure detection, and local failure migration based on removing the failed processes from shrinking communicators. Laguna et al. [24] show that as processes continue to fail, the time to revoke and shrink the communicator increases linearly with increasing number of nodes. In addition, the paper shows that the interface is only suitable for jobs that have work-decomposition flexibility which exists for instance in a master-slave application model. However, for more general applications such as bulk synchronous MPI applications, the interface has few benefits.

MPICH-V [6] is a fault-tolerant MPI implementation designed for large clusters, where failures or disconnection between nodes are common events, resulting from human error, and hardware or software faults. MPICH-V adds fault-tolerance by using uncoordinated checkpointing and distributed pessimistic message logging. An essential goal of the work is to allow ease of use that allows running old applications without modification, ensuring transparent fault tolerance for users, etc. However, the paper shows that MPICH-V incurs an overhead of about 23% on a job's runtime when no failures occur. Also, the uncoordinated checkpointing is not user directed, instead it is system directed, which may result in significantly poor performance as application characteristics are not taken into account.

Dynamic process management can provide fault tolerance in MPI programs. Gropp et al. [16] show how the existing MPI specification, which usually serves as a message-passing system, can be extended to include an application interface to the system's job scheduler and process manager, or even to write those functions if they are not already provided. The specification allows running processes to spawn new processes and communicate with them. However, developers still need to handle explicitly issues such as resource discovery, resource allocation, scheduling, profiling, and load balancing.

Another dimension of dealing with faults is to develop algorithms that are inherently tolerant. To help matrix factorizations algorithms survive fail-stop failures during parallel execution, Du et al, [8] propose a hybrid approach based on algorithm-based fault tolerance (ABFT) that can be applied to several ubiquitous one-sided dense linear factorizations. Using LU factorization, the authors prove that this scheme successfully applies to the three well known one-sided factorizations, Cholesky, LU and QR. The algorithm protects both the left and right factorization results. ABFT protects the right factor with checksum generated before, and carried along during the factorizations. A scalable checkpointing method protects the left factor. However, the work does not support multiple simultaneous failures.

9 FUTURE WORK

In future we plan to explore the following directions of research and development:

(1) In Phase II of the development, we plan to extend the BTL framework to directly interface with SLIM and integrate SLIM as a patch for the Open MPI code base.
(2) In Phase III we plan to expand SLIM to interface with other interconnects, particularly OpenIB. This would allow BTL to access the underlying interconnects with SLIM's uniform interface. This work has been of interest to interconnect vendors [2].
(3) We plan to study Open MPI's COLL framework and see if we can unify the both COLL and SLIM's multi-party communication mechanisms to help with collective operations. This may either be in terms of semantics or in terms of implementation optimizations.
(4) We also will plan to investigate how we may apply the lessons learned through SLIM and Open MPI implementations to other MPI implementations (e.g., MPICH).

10 SUMMARY

As computational capabilities scale from petaflops to exaflops, more compute processes work concurrently to accomplish tasks, requiring more communication. Increase in the number of software and hardware components is strongly correlated with potential increase in faults. In this paper, we presented SLIM, a shim layer, that improve resiliences against transient faults in network communication. We make a case that the coupling of the session and transport layer in implementations inhibit recovery from transient failures. SLIM

decouples the two semantics by acting as shim underneath BTL, allowing users to run their code without change.

ACKNOWLEDGMENTS

The authors would like to thank Eric Brown for his consultation, feedback, and support over the years, particularly with regards to research and development in relation to SLIM. We are also grateful to the anonymous reviewers for their valuable feedback and guidance in improving our ongoing research.

REFERENCES

[1] 2015. POSIX.1-2008 Specification. http://pubs.opengroup.org/onlinepubs/9699919799/functions/contents.html. (2015). Accessed December 16, 2016.

[2] 2016. Personal Communication with Mellanox Representative at Supercomputing. (2016). http://sc16.supercomputing.org

[3] Al Geist and Christian Engelmann. 2002. Development of Naturally Fault Tolerant Algorithms for Computing on 100,000 Processors. (2002). http://www.csm.ornl.gov/~geist/Lyon2002-geist.pdf

[4] R. Batchu, J. P. Neelamegam, Zhenqian Cui, M. Beddhu, A. Skjellum, Y. Dandass, and M. Apte. 2001. MPI/FTTM: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel Computing. In *IEEE/ACM International Symposium on Cluster Computing and the Grid*. 26–33. DOI:http://dx.doi.org/10.1109/CCGRID.2001.923171

[5] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. 2013. Post-Failure Recovery of MPI Communication Capability. *The International Journal of High Performance Computing Applications* 27, 3 (2013), 244–254. DOI:http://dx.doi.org/10.1177/1094342013488238

[6] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. 2002. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *The International Conference for High Performance Computing, Networking, Storage and Analysis*. 29–29. DOI:http://dx.doi.org/10.1109/SC.2002.10048

[7] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. 2014. Toward Exascale Resilience: 2014 update. *Supercomputing Frontiers and Innovations* 1, 1 (2014). http://superfri.org/superfri/article/view/14

[8] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. 2012. Algorithm-based Fault Tolerance for Dense Matrix Factorizations. In *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. 225–234. DOI:http://dx.doi.org/10.1145/2145816.2145845

[9] Ifeanyi P. Egwutuoha, Shiping Chen, David Levy, and Bran Selic. 2012. A Fault Tolerance Framework for High Performance Computing in Cloud. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID '12)*. 709–710. DOI:http://dx.doi.org/10.1109/CCGrid.2012.80

[10] Ifeanyi P. Egwutuoha, David Levy, Bran Selic, and Shiping Chen. 2013. A Survey of Fault Tolerance Mechanisms and Checkpoint/Restart Implementations for High Performance Computing Systems. *The Journal of Supercomputing* 65, 3 (2013), 1302–1326. DOI:http://dx.doi.org/10.1007/s11227-013-0884-0

[11] Graham E Fagg, Antonin Bukovsky, and Jack J Dongarra. 2001. HARNESS and Fault Tolerant MPI. *Parallel Computing* 27, 11 (2001), 1479–1495. DOI:http://dx.doi.org/10.1016/S0167-8191(01)00100-4

[12] Graham E. Fagg and Jack J. Dongarra. 2004. Building and Using a Fault-Tolerant MPI Implementation. *The International Journal of High Performance Computing Applications* 18, 3 (2004), 353–361. DOI:http://dx.doi.org/10.1177/1094342004046052

[13] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, and others. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*. Springer, 97–104.

[14] Marc Gamell, Daniel S. Katz, Hemanth Kolla, Jacqueline Chen, Scott Klasky, and Manish Parashar. 2014. Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. 895–906. DOI:http://dx.doi.org/10.1109/SC.2014.78

[15] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[16] W. Gropp and E. Lusk. 1995. Dynamic Process Management in an MPI Setting. In *7th IEEE Symposium on Parallel and Distributed Processing*. 530–533. DOI:http://dx.doi.org/10.1109/SPDP.1995.530729

[17] William Gropp and Ewing Lusk. 2004. Fault Tolerance in Message Passing Interface Programs. *The International Journal of High Performance Computing Applications* 18, 3 (2004), 363–372.

[18] William Gropp, Ewing Lusk, and Anthony Skjellum. 1999. *Using MPI: Portable Parallel Programming with the Message Passing Interface* (2nd ed.). MIT Press.

[19] Thomas Hérault and Yves Robert. 2015. *Fault-Tolerance Techniques for High-Performance Computing*. Springer.

[20] Umar Kalim. 2011. Demonstration Video of Seamless Virtual Machine Migration. http://www.cs.vt.edu/ umar/vm-demo. (2011).

[21] Umar Kalim, Eric Brown, Mark Gardner, and Wu Feng. 2010. Enabling Renewed Innovation in TCP by Establishing an Isolation Boundary. http://pfld.net/2010/technical.php. In *8th International Workshop on Protocols for Future, Large-Scale and Diverse Network Transports (PFLDNeT)*.

[22] Umar Kalim, Mark Gardner, Eric Brown, and et al. 2013. Seamless Migration of Virtual Machines Across Networks. In *IEEE International Conference on Computer Communication and Networks (ICCCN)*.

[23] Umar Kalim, Mark Gardner, Eric Brown, and Wu Feng. 2017. SLIM: Enabling Transparent Extensibility and Dynamic Configuration via Session-Layer Abstractions. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*.

[24] Ignacio Laguna, David F. Richards, Todd Gamblin, Martin Schulz, and Bronis R. de Supinski. 2014. Evaluating User-Level Fault Tolerance for MPI Applications. In *Proceedings of the 21st European MPI Users' Group Meeting (EuroMPI/ASIA '14)*. Article 57, 6 pages. DOI:http://dx.doi.org/10.1145/2642769.2642775

[25] Soulla Louca, Neophytos Neophytou, Adrianos Lachanas, and Paraskevas Evripidou. 2000. MPI-FT: Portable Fault Tolerance Scheme for MPI. *Parallel Processing Letters* 10, 04 (2000), 371–382. DOI:http://dx.doi.org/10.1142/S0129626400000342

[26] Jeffrey M. Squyres and Andrew Lumsdaine. 2005. The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms. (2005), 167–185. DOI:http://dx.doi.org/10.1007/0-387-23352-0_11

[27] S. Rao, L. Alvisi, and H. M. Vin. 1999. Egida: an extensible toolkit for low-overhead fault-tolerance. In *29th Annual International Symposium on Fault-Tolerant Computing*. 48–55. DOI:http://dx.doi.org/10.1109/FTCS.1999.781033

[28] B. Schroeder and G. Gibson. 2010. A Large-Scale Study of Failures in High-Performance Computing Systems. *IEEE Transactions on Dependable and Secure Computing* 7, 4 (2010), 337–350. DOI:http://dx.doi.org/10.1109/TDSC.2009.4