CIFTS: A Coordinated Infrastructure for Fault-Tolerant Systems

R. Gupta*, P. Beckman*, B. H. Park†, E. Lusk*, P. Hargrove‡, A. Geist†, D. K. Panda§, A. Lumsdaine¶ and J. Dongarra‖

*Argonne National Laboratory {rgupta, beckman, lusk}@mcs.anl.gov

†Oak Ridge National Laboratory {parkbh, gst}@ornl.gov

‡Lawrence Berkeley National Laboratory phhargrove@lbl.gov

‡Ohio State University panda@cse.ohio-state.edu

¶Indiana University lums@osl.iu.edu

‖University of Tennessee, Knoxville dongarra@eecs.utk.edu

Abstract—**Considerable work has been done on providing fault tolerance capabilities for different software components on large-scale high-end computing systems. Thus far, however, these fault-tolerant components have worked insularly and independently and information about faults is rarely shared. Such lack of system-wide fault tolerance is emerging as one of the biggest problems on leadership-class systems. In this paper, we propose a *coordinated infrastructure*, named *CIFTS*, that enables system software components to share fault information with each other and adapt to faults in a holistic manner. Central to the CIFTS infrastructure is a Fault Tolerance Backplane (FTB) that enables fault notification and awareness throughout the software stack, including fault-aware libraries, middleware, and applications. We present details of the CIFTS infrastructure and the interface specification that has allowed various software programs, including MPICH2, MVAPICH, Open MPI, and PVFS, to plug into the CIFTS infrastructure. Further, through a detailed evaluation we demonstrate the nonintrusive low-overhead capability of CIFTS that lets applications run with minimal performance degradation.**

I. Introduction

With processor speeds no longer doubling every 18-24 months owing to the exponentially increasing power consumption and heat dissipation, modern high-end computing (HEC) systems no longer rely on the speed of a single processing element to achieve high performance. Instead, they utilize the parallelism of a massive number of moderately fast processing elements to achieve the required performance. Today, systems with hundreds of thousands of processing elements are already available. As we pass the era of petaflop computing and look forward to multi-petaflop and exaflop computing, systems with millions of processing elements (and tens to hundreds of millions of hardware components) are expected to arrive soon. With such massive scale systems already available and even larger systems on the horizon, failure resilience and fault tolerance are quickly emerging as the most relevant issues facing HEC systems.

Recently, considerable work has focused on fault tolerance for system software components, including the message passing interface (MPI), file systems, resource management infrastructure, and applications. However, most of this work has addressed the fault tolerance problem in an isolated and uncoordinated manner, in which each software component operating in the system independently detects and handles errors. While such an approach is reasonable for certain fault scenarios, its isolated nature typically limits the effectiveness of fault response. Without a holistic, full-system approach, current fault-tolerant software is ill-equipped to detect, diagnose, and adaptively respond to faults in ultrascale environments.

In this paper, we present our approach to addressing needs of current and emerging HEC systems by developing a *coordinated infrastructure* for fault-tolerant systems, named ***CIFTS*** (**C**oordinated **I**nfrastructure for **F**ault **T**olerant **S**ystems). As a part of this infrastructure, we propose a Fault Tolerance Backplane (FTB), which serves as a backbone for CIFTS. FTB provides a shared infrastructure for system software to coordinate fault information and responses through a uniform interface, usable throughout the HEC software stack. The ability to share information provides *FTB-enabled system software* with a basis for proactive fault management and fault handling. In addition, the availability of fault information in the system enables development of new software that can respond to certain kinds of faults, especially those requiring the coordination of multiple elements of the system, thereby enabling the system to recover from and alleviate faults they were unable to detect independently.

Together with the detailed description of the FTB design, we present an interface specification that enables various software programs—including MPICH2, MVAPICH, Open MPI, BLCR, Cobalt, PVFS, and the SWIM IPS application—to plug into the infrastructure. We also illustrate the nonintrusive, low-overhead capability of the FTB through a comprehensive empirical evaluation that involves using micro-benchmarks as well as applications on various systems. Our experiments demonstrate marginal overhead of FTB on applications in most cases, with several significant optimizations still possible.

II. Related Work

Most HEC fault-tolerant system software attempts to handle faults independently. This software handles faults either reactively (e.g., by automatic network path migration [1], job placement adaptation, check-pointing) or proactively (e.g., by preemptive process migration [2]). A widely used method to implement fault tolerance in middleware libraries and user-level applications is through check-pointing software [3], [4]. Various methods of checkpointing—application-level or system-level—have been studied, along with techniques to optimize them [5] [6]. Considerable research [7], [8], [9] has been devoted to building fault-tolerant MPI implementations. For example, MVAPICH [10] supports fault tolerance through checkpoint/restart mechanisms, MPICH-V [11] supports fault tolerance through checkpointing and message logging, FT-MPI [12] can survive multiple MPI process crashes and respawn MPI tasks and LA-MPI [13] supports fault tolerance at the network level using redundant data paths. Also being studied is fault tolerance using virtualization techniques [14], where processes can be proactively migrated from unhealthy nodes to healthy nodes based on health monitoring information. Software such as FENCE [15], which performs on-the-fly rescheduling of jobs upon detection of faults, is also being investigated. Moreover, researchers are developing approaches [16] to provide fault tolerance through a combination of adaptive proactive and reactive fault tolerance mechanisms in conjunction with system health monitoring and reliability analysis. Existing software such as Nagios [17] and IBM Tivoli [18] aim to provide system-wide monitoring and automatic fault detection and automatic diagnosis to some extent.

All these software efforts provide fault tolerance to some degree. However, they do not share their fault information with other software outside their stack. The FTB is *complementary* to such fault-tolerant system software because such system software can plug into the FTB and use its framework to share fault information with other software in the system. To the best of our knowledge, no attempt has yet been made to provide a generic framework to coordinate between different system software for fault tolerance on a systemwide basis.

The FTB framework and interface are based on the publish/subscribe framework. While considerable research on this framework has been done, no existing implementation of such a system can serve as a low-level, minimum-overhead communication layer for exchanging fault information on HEC systems. To meet this need, we have custom-built the FTB system and have provided an interface, based on accepted guidelines and experiences published by other publish/subscribe frameworks [19] [20] [21].

III. The CIFTS Framework

In this section, we discuss the CIFTS framework, the FTB design, and the way FTB-enabled software programs interact with each other.

The CIFTS framework is composed of the Fault Tolerance Backplane, which forms the core of CIFTS. The FTB is an asynchronous messaging backplane providing communication

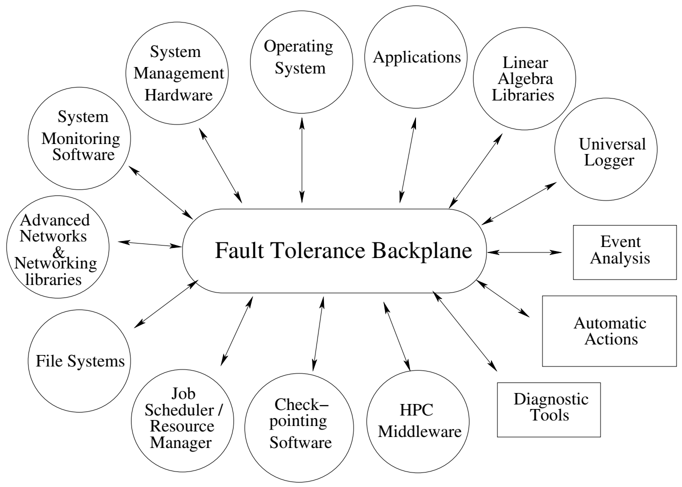

Fig. 1. CIFTS Framework

between system software programs, whether inherently fault-tolerant or not. As shown in Figure 1, the FTB can be used by system software ranging from operating systems and job schedulers to mathematics libraries, file systems, and high-level user applications. In addition to existing software, third-party developers can set up automatic scripts, diagnostic routines, fault-information analysis engines, and logging systems that can be FTB-enabled to communicate with other FTB-enabled software.

From the FTB's perspective, a *fault event* is defined as information about any condition in the system that has caused or can cause excessive errors or can stop the system from working. A fault need not be an error, but it can be a warning relevant to an end-user FTB-enabled software program, also referred to as an FTB client. Table I shows a CIFTS scenario with various FTB-enabled software systems where an FTB-enabled application encounters an error (a failed I/O node) while communicating with the FS1 file system. Instead of silently failing, the application uses FTB to publish a fault event detailing the error encountered. This event is received by an FTB-enabled job scheduler, which launches further jobs on an FS2 file system. The event is also received by other instances of the FTB-enabled FS1 file system, which can start an automatic recovery process (migration of the failed I/O node to a different I/O node). Moreover, this event is received by FTB-enabled monitoring software, which logs the event and notifies the administrator by email. Thus, the availability of fault information improves the overall resiliency of the system.

A. FTB Architecture

The FTB physical infrastructure is based on a distributed architecture, as shown in Figure 2. The FTB framework comprises a set of distributed daemons, called as FTB agents. These agents incorporate the bulk of the FTB logic and manage the bookkeeping as well as communication of events throughout the FTB system.

The FTB agents, on startup, connect and organize themselves into a tree-based topology. The initial topology con-

FTB-Enabled Software	Fault Events	Action Taken
Application	Publish event about error on FS1 file system	
Job Scheduler	Receives event about error on FS1 file system	Launches next jobs on FS2 file system
File System FS1	Receives event about error on FS1 file system	Starts recovery process of FS1
Monitoring Software	Receives event about error on FS1 file system	Logs and Emails administrator

TABLE I
SCENARIO USING THE CIFTS INFRASTRUCTURE

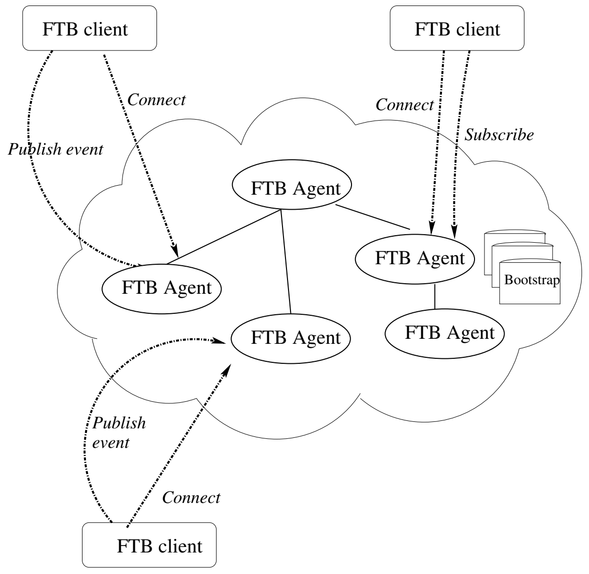

Fig. 2. The FTB Architecture

struction takes place with the assistance of the FTB bootstrap server which provides information that helps every FTB agent determine its parent FTB agent and position in the topology tree. During its lifetime, if an agent loses its parent, it can connect itself (and its children and its attached FTB clients) to a new parent in the topology tree, making the topology tree self-healing with a certain level of fault tolerance. The bootstrap server can also be made fault tolerant to a certain extent by keeping track of the topology information and specifying redundant bootstrap servers. The FTB agents subsequently connect to the existing agent topology tree when they startup.

An FTB client is linked to a lightweight FTB client library that provides it with the FTB Client API (described in the next section). The FTB client, on startup, connects to a local FTB agent by using routines provided by the FTB Client API. Alternatively, in the absence of a local FTB agent, the FTB client connects to a remote FTB agent by enlisting the assistance of the FTB bootstrap server. The FTB Client API is influenced by and based on publish/subscribe framework routines and semantics. Once a connection is established, the FTB client can publish events and subscribe to receive events using the FTB Client API.

The FTB agents keep track of all registered FTB clients. The agents also keep track of all FTB client subscription requests, along with the subscription criteria. They perform incoming event matching against subscription criteria and send events to the correct destinations and clients. In addition, they keep track of their tree topology and metadata associated with maintaining connections and routing information. In summary, the majority of the FTB logic lies with the FTB agent.

B. FTB Client API

The FTB clients interact with the FTB through a small set of simple routines provided by the FTB Client API. The FTB API provides a routine called *FTB_Connect* to be used by every FTB client to initialize itself and connect to the FTB system. The FTB client must specify various details including the namespace in which it plans to publish its events. The *FTB_Publish* routine can be called by the FTB client to publish events. Events currently can be published only in the namespace specified during the FTB_Connect call. The FTB client also associates a severity value (values for severity are defined by FTB to be fatal, warning, or info) with the event.

The FTB API provides a routine called *FTB_Subscribe* to allow an FTB client to subscribe to events. The client also needs to specify the subscription string that specifies the subscription criteria. For example, "jobid=47863; severity=fatal" means that the FTB client is subscribing to receive events of severity fatal from FTB clients that are part of jobid=47863. The FTB framework provides the clients with two mechanisms to receive events, both of which need to be specified in the *FTB_Subscribe* routine. The first is a callback mechanism. The FTB client must specify the callback function in the *FTB_Subscribe* call. When FTB encounters a fault event matching a subscription string, it calls the relevant callback function. The advantage of this approach is that it is asynchronous and does not require intervention by the FTB client to explicitly go and get the event. Alternatively, the FTB client can have fault events delivered to it by the FTB framework through a polling mechanism. In the polling mechanism, a queue is allocated for the FTB client. The FTB puts the fault events matching the subscription string in the queue. The FTB client is responsible for pulling the event from the polling queue. The FTB API provides a routine called *FTB_Poll_event* for this purpose. While the polling mechanism is rarely provided by publish/subscribe frameworks, it is useful for machines where callback function threads cannot be launched.

Complementary to the *FTB_Subscribe*, the FTB API provides a routine called *FTB_Unsubscribe* that allows a client to unsubscribe a subscription string from the FTB framework. The FTB API also provides a routine called *FTB_Disconnect* that allows the FTB client to disconnect from the FTB system.

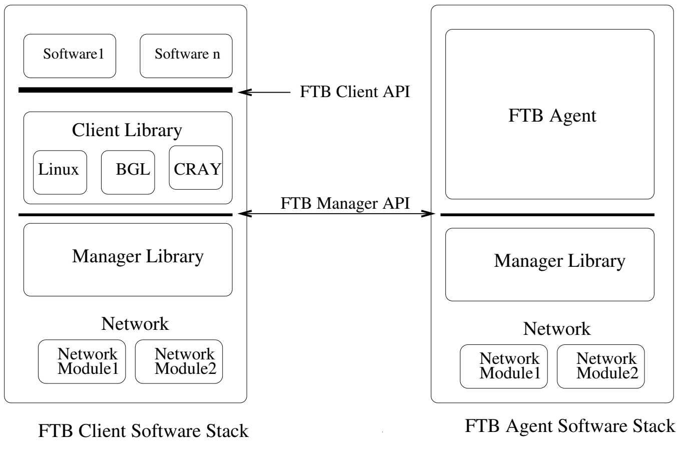

Fig. 3. Layered FTB Software Stack

In addition to these main routines, other routines provided by the FTB Client API can be found in the FTB Developers Guide [22].

C. FTB and Event Namespacing

FTB imposes no restrictions on the fault information that an FTB client can publish. While the FTB provides the infrastructure for communicating events between different FTB clients, the semantics of the events are independent of FTB and must be understood and defined prior to using FTB. To this end, the FTB design incorporates an *event namespace*, portions of which are reserved for the different FTB-enabled software programs. In the FTB framework, prior to publishing any event the FTB client must specify the namespace where it plans to publish its fault events. Similarly, FTB clients wishing to receive events need to ensure that they have registered their interest to receive events in the correct namespace. Conceptually, the namespace can be thought of as a hierarchical string. The FTB framework has reserved the leading string "ftb." to identify events for which the semantics have been formally agreed upon in advance by the CIFTS community. The rest of the namespace is not formally managed, though we expect conventions to develop, which we can codify as needed.

To understand this situation more clearly, consider a simplified example of the FTB-enabled MPICH software. If an FTB-enabled MPICH instance publishes an "MPI_ABORT" event in the "ftb.mpich" namespace, then all software that has registered interest in getting events from the "ftb.mpich" namespace can be assumed to have a common understanding of the semantics of the MPI_ABORT fault event. If, instead, the same event were published in the "test.mpich" namespace, different semantics might apply, which should be separately agreed on by the senders and receivers.

D. FTB Software Stack

The FTB has a layered architecture, as depicted in Figure 3. Each of the three layers—(a) FTB client layer, (b) manager layer, and network layer—has its own functionalities.

1) The Client Layer: The FTB client layer exposes a set of routines, that constitute the FTB Client API. This thin, lightweight layer resolves incompatibilities and ensures portability of the FTB Client API across various platforms such as Linux, IBM Blue Gene machines, and the Cray XTs. This layer uses the underlying FTB Manager API (exposed by the manager layer) to communicate with the manager layer. The FTB client layer library is typically linked to the FTB clients, who use the exposed FTB Client API to communicate with the FTB.
2) The Manager Layer: The FTB manager layer handles the bulk of the FTB bookkeeping and decision making. This layer exposes a set of routines through the FTB Manager API. The FTB Manager API is an internal API used by upper layers such as the FTB client layer and FTB agents and is not exposed to FTB-enabled software developers. The manager layer keep tracks of the FTB clients, their subscription criteria, and subscription mechanisms. It also includes the logic of matching published events to requested subscriptions and routing fault events to relevant receivers in the FTB framework, as well as handling the internal fault tolerance of FTB itself.
3) Network Layer: The network layer is the lowest layer of the software stack. The network layer deals with sending and receiving of data. The network layer is transparent to the upper layers and is designed to support multiple modes of communication using protocols such as TCP/IP or shared-memory communication. Current FTB implementations use TCP/IP to create the agent tree topology and connect FTB clients to the FTB agents.

E. Scalability Challenges

A framework such as the FTB presents many challenges from the scalability point of view. With multiple FTB-enabled software components publishing fault events simultaneously, event storming becomes an important issue. While network congestion may take place in any communication system, some situations specific to the coordinated infrastructure framework may lead to fault storms. Such situations occur, for example, when an FTB client sees repeated errors with a hardware component and sends an event over the FTB for every error, thereby flooding the network or when multiple FTB clients, perhaps belonging to different FTB-enabled software, detect a single system fault and flood the network with events. Reducing event storms on a system requires aggregation of these events into composite events. This is a common research topic in the area of log analysis [23] [24].

Aggregation can be handled either in the FTB framework or in the FTB-enabled software; it is less cumbersome if the user FTB-enabled software does not have to handle it. We discuss below our plans to have the FTB framework handle event aggregation.

1) Dealing with Same Symptom Fault Events: Consider a scenario where an FTB-enabled middleware sees "Disk I/O Write error" messages. For every message it sees, it publishes a fault event to the FTB system. Such events are called "same symptom fault events" because they represent the same fault. Clearly, forwarding of rapid, repeated fault events through the FTB to interested receivers can easily lead to network congestion.

Such situations can potentially be handled by the FTB framework by using time-stamps. Every fault event is time-stamped by the FTB client at the source. Fault events originating at the same source with the same fault information (i.e., "Disk I/O Write error") but narrowly different time-stamps are assumed to represent the same fault. Such "duplicate" events can be detected and quenched by the FTB agent connected to that FTB client by maintaining short-duration event publishing histories for each client.

2) Dealing with Dissimilar Symptom Fault Events: Consider a scenario where a network link fails. Various FTB-enabled software programs detect this link failure with different symptoms. The MPI library sees a "failure to communicate with rank r"; the FTB-enabled network protocol stack sees "port x down"; the network monitoring agent sees "link z down"; and the application sees "network timeout". A single fault manifests a variety of symptoms in different software components. In addition to contributing to network congestion, the many events produced in this scenario might evoke multiple conflicting (or at least uncoordinated) responses from different components if their relationship were not recognized.

Linking multiple symptoms to a single potential cause requires root analysis, and in this aspect, FTB faces challenges similar to those seen in the area of event log analysis and event correlation [25] [26] [27]. Such situations can potentially be handled by FTB by segregating every fault event in organized hierarchical categories called *event categories*, which helps FTB determine the probabilistic similarity between different faults. For example: The above fault events would probably be categorized as "network – link failure". All events originating from a common source in the same time frame and belonging to the same category could be aggregated and replaced by a composite event. The biggest challenge with this approach is determining how to avoid manual categorization and move toward automatic categorization of fault events into fault categories. More sophisticated implementations and approaches to event aggregation are currently under development in the FTB infrastructure.

IV. Performance Evaluation

In this section, we describe various experiments to understand the performance and characteristics of the current FTB implementation on a 24-node Linux-based cluster and the Cray XT machine at Oak Ridge National Laboratory. The Linux cluster is a dual-processor, dual-core cluster with 4 GB memory and 1GB cache, interconnected through a Gigabit Ethernet network. The Cray XT4 system is a 7000+ node system with quad-core processors and 8 GB memory each. The Compute Node Linux (CNL) OS runs on each compute node. Each node is connected to a Cray SeaStar router through HyperTransport, and the SeaStars are all interconnected in a 3-D-torus topology.

A. FTB Event Publish Performance

FTB clients, on startup, attempt to connect either to a local FTB agent or to a remote FTB agent. In this section, we measure the time taken by an FTB client to publish an FTB event and evaluate whether the location and number of FTB agents on a system have any impact on the event publish time. The micro-benchmark test consecutively publishes 2,000 events on one of the nodes of the Linux cluster and calculates the average time taken to publish one event by a client. At the same, the number of agents in the system is increased. For tests with a local FTB agent, an FTB agent is present on the same node on which the test is being run. For tests with remote FTB agents, none of the FTB agents are present on the node on which the test is being run. From Figure 4(a), we can see that the location and number of FTB agents have little impact (the small variations in time are attributed to benchmarking noise) on the event publish time. Thus, system administrators and users need not be concerned about the location and number of agents if their FTB client mostly publishes events.

B. FTB Event Poll Performance

An FTB client can receive events using FTB's event notification mechanism or by polling for events. Figure 4(b) shows the poll time for varying numbers of events. We measure this time in the presence and absence of FTB traffic. In the "No FTB" traffic scenario, FTB agents are started on two nodes with an FTB client publishing events on one node and an FTB-enabled monitoring software polling for events on the second node. In the "FTB traffic" scenario, agents run on all 24 nodes, with an FTB client publishing events on one node and 24 instances (one on each node) of FTB-enabled monitoring software polling for all events. The event poll time is calculated based on the average time seen across all the nodes. From Figure 4(b), we see that the polling time for both cases is the same for approximately 128 events or fewer, but increases in the presence of FTB traffic for higher numbers of events (around 256). The reason is that 256 events are propagated by every FTB agent to the local FTB-enabled monitoring software process as well as the other agents connected to it through the tree topology. Thus, events take longer to reach all the FTB-enabled monitoring software instances, and they might not be readily available in the poll queue, resulting in higher poll times.

C. Impact of FTB Traffic on Non-FTB Application Performance

FTB-agnostic applications, in addition to FTB-enabled applications, will exist on any system supporting FTB. In this section, we evaluate the impact of FTB traffic (generated by FTB-enabled applications) on FTB-agnostic application performance. The test environment consists of FTB agents running on all the 24 nodes of the Linux cluster. These agents form a tree topology. Agents that are leaves of the topology tree are involved in less FTB communication than agents that are intermediate nodes of the tree, because intermediate nodes agents have to receive and forward events to their children as

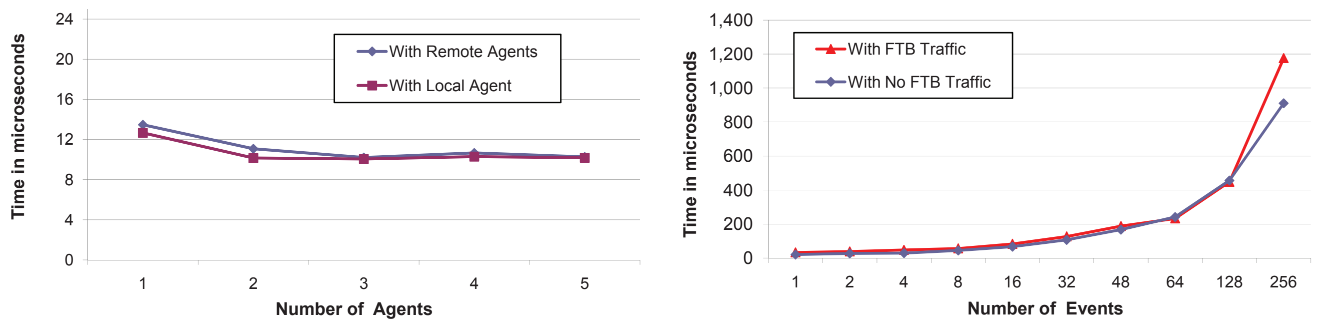

Fig. 4. (a) Event Publish Performance; (b) Event Poll Performance

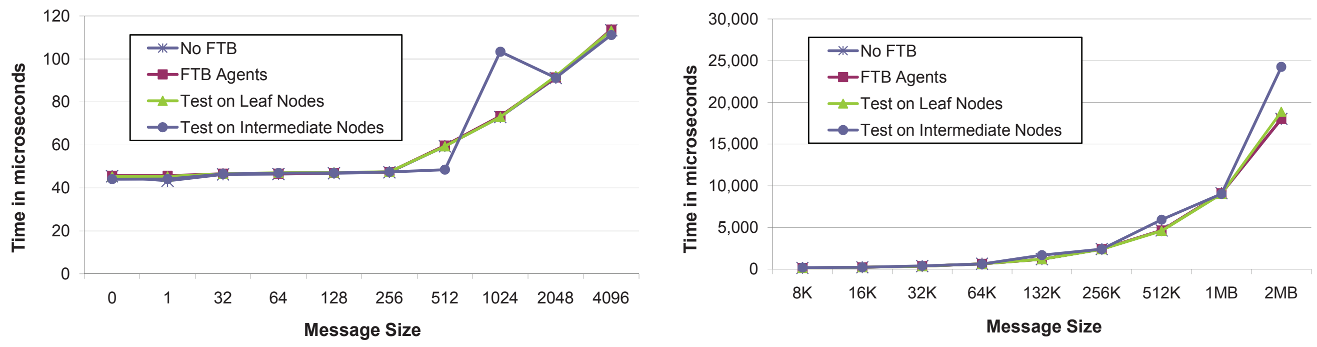

Fig. 5. Impact of FTB Traffic on MPI Latency Benchmark: (a) Small Messages; (b) Large Messages

well as their parent. An FTB-enabled all-to-all application is run on 22 nodes of the 24-node cluster. Each of the 22 nodes involved in the all-to-all process, connect to their local FTB agent, publish 2000 events, and poll for all events from all nodes (i.e., 2000 events from 22 nodes = 44,000 events). This generates an immense amount of FTB traffic because every agent is involved in forwarding these events to its children or parent. On the remaining two nodes of the cluster, we run a non FTB-enabled MPI latency micro-benchmark from the Ohio State University OMB benchmark suite.

We evaluate the impact of the FTB-enabled all-to-all application traffic on the performance of the MPI latency micro-benchmark. We consider two scenarios: (1) MPI latency run on the leaf nodes of the FTB agent topology tree, and (2) MPI latency run on the intermediate nodes of the FTB agent topology tree. As seen from the curves in Figure 5(a) and 5(b), we run the MPI latency test with (a) no FTB infrastructure, (b) FTB agents but no FTB-enabled software (labeled FTB Agents), (c) two leaf nodes of the agent topology tree, and (d) two intermediate nodes of the agent topology tree. The performance is the same for cases (a), (b), and (c) for both small and large messages. For case (d), however, the performance of the MPI latency benchmark is impacted. This performance degradation is due to the contention arising from a single network on a machine that is shared both by the FTB agent and the MPI latency benchmark. In this test, we selected the root node of the agent topology tree and its immediate child as the two intermediate nodes to run MPI latency test. Thus, the FTB traffic seen by agents on these two nodes was very heavy (as they were serving multiple children and grand-children) and consequently the network on these two machines was also being heavily utilized leading to network contention.

D. Analysis of FTB Traffic Patterns

In this section, we study the behavior of the FTB infrastructure in dealing with FTB-enabled applications having various communication patterns. We evaluate two traffic patterns: all-to-all and multiple groups.

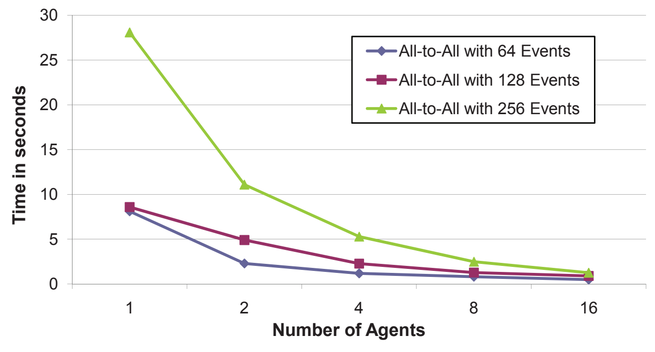

Fig. 6. Impact of All-to-All Patterns with FTB

Figure 6 shows the execution time for the all-to-all application, running on 64 cores (16 Linux nodes with 4 cores each), with varying numbers of agents and varying numbers of events. Each of the 64 instances of the all-to-all benchmark publish k events and poll for (k * 64 clients) events, with k varying from 64 to 128 to 256. We see that execution time is about 8 seconds (for 64 and 128 events) and 28 seconds (for 256 events) when only one agent is in the system. The reason is that the single agent gets overloaded when multiple clients publish large numbers of messages. In the extreme case of 256 events, the single agent receives 256 events from each of the 64 cores and forwards (256 x 64) events to *each* of these 64 cores. As we increase the number of agents from 2 to 4 to 8, we see that execution time decreases as event distribution work gets distributed among all the agents. As we approach 16 agents (i.e., one agent per node), we obtain the best performance, since each agent node handles only its local FTB clients. Thus, for FTB-enabled software with communication trends similar to all-to-all communication, having an local FTB agent on every node may be beneficial.

The all-to-all application has heavy FTB communication between *all* the nodes in a cluster. In the next experiment, we study the FTB infrastructure with multiple smaller groups, each performing FTB communication only between its member nodes. Examples of such groups in real life include FTB-enabled file system sharing its fault events among its processes and an FTB-enabled application sharing information with its peer instances. The multiple groups in our experiment individually perform all-to-all communication (with each instance publishing 100 events) in order to simulate heavy FTB event traffic. In our 64 core cluster (16 nodes with 4 cores each), we evaluate the group sizes of 4 to 64. We evaluate this benchmark under three scenarios.

1. The first scenario, labeled as "multiple groups", consists of FTB agents running on each of the 16 nodes of the Linux cluster. For a group size of 4, we divide the 64-core cluster into 16 groups, which each group localized to 1 node. For a group size of 8, we divide the 64-core cluster into 8 groups, with each group localized to 2 nodes. Similarly, for group size of 16, 32, and 64 we form groups of 4, 2, and 1, respectively. In Figure 7, the x-axis shows the various group sizes and the y-axis shows the execution time (averaged across the multiple groups) to complete the all-to-all communication for each group size and total number of publish events as 64 and 128. Note that since multiple groups exist on the cluster at any time (except when the group size is 64), the FTB agents on every node are involved in handling FTB traffic belonging to multiple groups, in addition to serving the group their local FTB clients belong to.

2. The second scenario, labeled as "one group", serves as a baseline to compare the first scenario. We measure the execution time for the all-to-all benchmark for various group sizes of 4, 8, 16, 32, and 64, but *only one* group exists on the cluster at a given time. Thus, each agent is involved in handling FTB traffic belonging to the group its local FTB clients belong to.

As seen in Figure 7(a) and 7(b), execution time can be heavily impacted when there are multiple groups performing localized FTB communication. For example, for a group size of 8, each process in the all-to-all test sends 100 events and receives 800 events. When multiple groups are present (8 groups for a 64-core cluster), an FTB agent on a node of the tree topology can receive and forward around (16 agents x 800 events) events each. Thus the test with multiple groups takes double the amount of time (from Figure 7(a) for 64 events and 8, 16, and 32 processes) and more than double for 128 events (from 7(b)) as compared to the all-to-all test when there is only one group.

3. The third scenario, labeled "event aggregation", presents a case when event aggregation is applied to reduce the event storming witnessed in scenario (1). In this benchmark, multiple groups are still formed on the cluster. The agent receives 100 events from each node in the group. It aggregates the 100 events, however, and sends back only 1 composite event for every node it received events from. Thus, for a group size of '8', every node publishes 100 events but receives only '8' events instead of '800'. As can be seen from the results in Figure 7(a) and 7(b), event aggregation can dramatically improve performance and reduce traffic overhead in the system.

E. Impact of FTB on Applications

In this section, we evaluate the scalability and FTB overhead on two FTB-enabled applications, running on the Linux cluster and the Cray XT machine, respectively.

We first evaluate the Integer Sort (IS) benchmark from the NAS Parallel Benchmarks (NPB) suite [28] on the 16-node Linux cluster. The NPB suite consists of 5 kernels and 3 simulated application benchmarks, which emulate the computation and data movement characteristics of large-scale computational fluid dynamics (CFD) applications. The Integer Sort application is a popular MPI-based benchmark from this suite, which performs a sorting operation that is considered important in particle physics code. We evaluate the original IS benchmark (version 3.3, largest class size C) with no FTB infrastructure and compare it against the modified FTB-enabled benchmark. In the FTB-enabled IS, every instance of IS publishes events and polls back for those events. We test the FTB-enabled IS with events varying from 16 to 64 to 96 events. FTB agents are present on all nodes of the cluster, and an FTB-enabled monitoring software runs on one of the nodes of the cluster. The presence of the monitoring software ensures that the FTB agents are involved in forwarding events to other agents, in addition to their attached FTB clients. Figure 8(a) shows the results of this evaluation. The execution time for FTB-enabled IS as well as the original non-FTB IS is similar, barring the benchmarking noise. The original and FTB results for the IS benchmark on 16x1 system size are higher than 8x1 system because the communication overhead of the IS benchmark exceeds the parallel computational benefit obtained.

We next evaluate a parallel maximal clique enumeration application [29] [30], a classical graph theory problem, on the Cray XT supercomputer. This MPI-based application finds all maximal cliques in a given graph and is used in many fields including bioinformatics for the study of protein-protein interaction and protein-protein homology affinity maps. A maximal clique is a complete subgraph that is not a subset of any larger complete subgraph. Each MPI node is given a disjoint search space so that the entire clique enumeration can be performed in parallel. Load balancing is achieved by

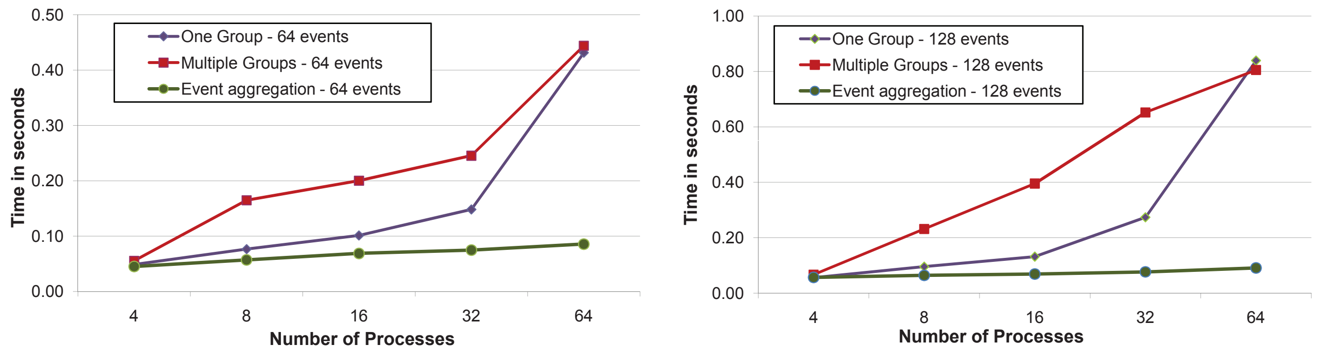

Fig. 7. Analyzing FTB traffic patterns: (a) Multiple groups – 64 events; (b) Multiple Groups – 128 events

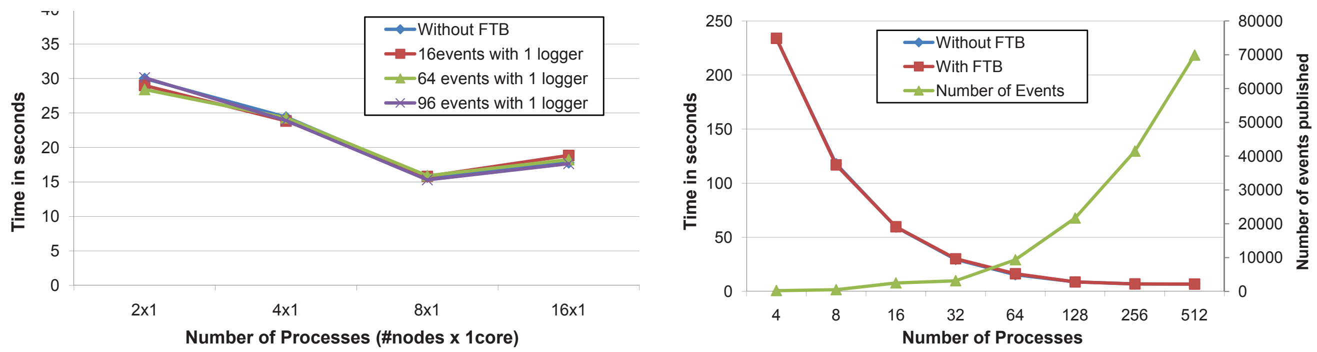

Fig. 8. FTB-enabled Applications : (a) NPB Integer Sort; (b) Maximal Clique Application

exchanging search spaces between busy and idle nodes. The overhead of FTB is measured by making the parallel maximal clique enumeration code publish FTB events during its course of run. More specifically, each MPI node publishes an FTB event at every occurrence of search space exchange. For this test, a graph of 4,087 vertices and 193,637 edges was used as the input, which embeds 3,429,816 maximal cliques. The execution times of the parallel maximal enumeration code were measured using up to 512 nodes. The FTB system had 1 agent serving 32 nodes. Figure 8(b) shows the number of Cray XT processes on the x-axis. As clearly delineated in the figure the overhead imposed by the FTB is negligible in most (if not all) cases. These results suggest that the FTB does not affect the performance of an application in practice.

V. Conclusions

Current high-end computing system software handle faults insularly and in an uncoordinated fashion. The lack of a coordinated infrastructure prevents software from sharing fault-related information and taking timely proactive decisions that could improve the overall reliability of the system. The goal of the CIFTS project is to develop a coordinated infrastructure that enables system software to actively share fault-related information. We achieve this by developing a fault tolerance backplane (FTB), which provides the messaging and communication infrastructure for this coordinated environment. In addition, we provide an interface specification that enables various system softwares to talk to each other. In this paper, we discussed the CIFTS framework along with the FTB architecture and interface specification. We also presented a detailed evaluation of the non-intrusive, low-overhead capabilities of CIFTS that allow applications to run with minimal performance degradation. Future work within CIFTS includes FTB-enabling popular and widely-used high performance computing software. In addition, the availability of fault information within FTB can allow users to develop automatic system-wide fault diagnosis, analysis and notification tools. More information about current and future publicly available FTB-enabled software can be found at [31]

VI. Acknowledgements

The CIFTS project is an on-going collaborative project between six organizations and several individuals. Teams from Argonne National Laboratory (working on ZeptoOS, MPICH2, PVFS and Cobalt), Oak Ridge National Laboratory (working on Common Component Architecture (CCA) frameworks and scientific applications), Lawrence Berkeley National Laboratory (working on BLCR), Ohio State University (working on MVAPICH2, FTB-InfiniBand monitoring software), Indiana University (working on the Open MPI) and University of Tennessee, Knoxville (working on the FT-LA math library) are a part of this project. We would like to thank each organization for their contribution towards this project. We would like to specifically acknowledge the following individuals for continuous effort on the CIFTS project: Argonne National Laboratory (E. Lusk, R. Thakur, R. Ross, P. Balaji, D. Buntinas and N. Desai), Oak Ridge National Laboratory (D. Bernholdt, P. Agarwal, S. Hampton, and A. Shet), Ohio State University (K. Gopalakrishnan), Indiana University (J. Hursey and T. Mattox), University of Tennessee, Knoxville (G. Bosilca,

A. Bouteiller), Lawrence Berkeley National Laboratory (P. Hargrove)

This work is supported by the U.S. Dept. of Energy, Office of Science, Advanced Computational Science Research Division. This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. Oak Ridge National Laboratory is managed by UT-Battelle, LLC for the U. S. Department of Energy

References

[1] A. Vishnu, A.R. Mamidala, S. Narravula, and D.K. Panda. Automatic Path Migration over InfiniBand: Early Experiences. In *IPDPS*, 2007.

[2] C. Engelmann, G. Vallee, T. Naughton, and S.L. Scott. Proactive Fault Tolerance using Preemptive Migration. In *PDP*, 2009.

[3] M. Litzkow and M. Solomon. Checkpointing and Migration of UNIX Processes in the Condor Distributed Processing System, 1992.

[4] J. Duell, P. Hargrove, and E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Technical Report LBNL-54941, 2002. Available at https://ftg.lbl.gov/CheckpointRestart/Pubs/blcr.pdf.

[5] E. Roman. A Survey of Checkpoint/Restart Implementations. Technical Report LBNL-54942, Lawrence Berkeley National Laboratory, 2002. Available at https://ftg.lbl.gov/CheckpointRestart/CheckpointPapers.shtml.

[6] J.G. Silva and L.M. Silva. System-level versus user-defined checkpointing. In *SRDS*, 1998.

[7] A. Luckow and B. Schnor. Migol: A Fault-Tolerant Service Framework for MPI Applications in the Grid. In *Journal of Future Generation Computer Systems '08*, volume 24, pages 142–152, 2008.

[8] J. Hursey, J. Squyres, T. Mattox, and A. Lumsdaine. The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI. In *IPDPS*, 2007.

[9] R. Subramaniyan, V. Aggarwal, A. Jacobs, and A. George. FEMPI: A Lightweight Fault-Tolerant MPI for Embedded Cluster Systems. In *ESA*, 2006.

[10] Q. Gao, W. Yu, W. Huang, and D.K. Panda. Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand. In *ICPP*, 2006.

[11] A. Bouteiler, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello. MPICH-V project: a multiprotocol automatic fault tolerant MPI. *The International Journal of High Performance Computing Applications*, 20:319–333, 2006.

[12] G. Fagg and J. Dongarra. Building and using a Fault Tolerant MPI implementation. In *International Journal of High Performance Applications and Supercomputing*, volume 18, pages 353–361, 2004.

[13] R. Aulwes, D. Daniel, N. Desai, and R. Graham et al. Network Fault Tolerance in LA-MPI. In *Euro PVM/MPI*, 2003.

[14] A. Nagarajan, F. Mueller, C. Engelmann, and S.L. Scott. Proactive Fault Tolerance for HPC with Xen Virtualization. In *ICS*, 2007.

[15] Z. Zheng, P. Gujrati, Z. Lan, and Y. Li. Enhancing Application Robustness through Adaptive Fault Tolerance. In *IPDPS*, 2008.

[16] S. Scott, C. Engelmann, G. Vallee, and T. Naughton et al. A tunable holistic resiliency approach for high-performance computing systems. In *PPoPP*, 2009.

[17] Barth Wolfgang. Nagios: System and network monitoring. 2005.

[18] G. Karjoth. Access control with ibm tivoli access manager. *ACM Transactions on Information and System Security (TISSEC)*, 6, 2003.

[19] P. Rousselle. Implementing the JMS publish/subscribe API. In *Dr. Dobb's Journal*, volume 27, pages 28–32, 2002.

[20] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. In *ACM Computing Surveys*, 2003.

[21] G. Muhl. Large-Scale Content-Based Publish/Subscribe Systems. In *Dissertation, Darmstadt University of Technology, Germany*, 2002.

[22] R. Gupta and P. Beckman et al. Fault Tolerance Backplane API. 2007. Available at http://www.mcs.anl.gov/research/cifts/docs.

[23] R. Vaarandi. Tools and Techniques for Event Log Analysis.

[24] S. Hansen and E. Arkins. Automated System Monitoring and Notification with Swatch. In *USENIX System Administration Conference*, pages 145–152, 1993.

[25] S. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie. High speed and robust event correlation. 34, 1996.

[26] R. Vaarandi. SEC a Lightweight Event Correlation Tool. In *IEEE Workshop on IP Operations and Management*, pages 111–115, 2002.

[27] M. Bing and C. Brickson. Extending UNIX System Logging with SHARP. In *USENIX System Administration Conference*, pages 101–108, 2000.

[28] D. Bailey, T. Harris, and W. Saphir et al. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, 1995.

[29] B.H. Park, N.F. Samatova, A. Jallouk, S. Molony, S. Horton, and S. Arcangeli. Data-driven, data-intensive computing for modelling and analysis of biological networks: application to bioethanol production. *Journal of Physics: Conference Series*, 78, 2007.

[30] B. Zhang, B.H. Park, T. Karpinets, and N. Samatova. From pull-down data to protein interaction networks and complexes with biological relevance. *Journal of Bioinformatics*, (24), 2008.

[31] Cifts website: http://www.mcs.anl.gov/research/cifts/.