GAIA: Strengthening the Reliability of Datacenter Computing via Fast Distributed Consensus

Abstract:

To process the rapidly increasing amount of data, more and more software applications run within a datacenter containing massive computing resources. To harness these resources, applications are typically ran by either of two independent infrastructures: schedulers and virtual machines (VM). Unfortunately, as an application runs on more computers, computer errors will occur more likely at runtime and can turn down the entire application, causing disasters such as the 2015 NYSE trading halts and recent Facebook outages. Existing infrastructures lack a high availability support for applications.

This GAIA project takes a holistic methodology to greatly improve application availability via three objectives. First, we will create APUS, a fast distributed consensus protocol for general applications. Distributed consensus, a strong fault-tolerance theory, runs multiple replications of the same application and makes these replications behave consistently as long as a majority of them work normally. An open challenge is that traditional consensus protocols are slow because their protocol messages go through various software layers (e.g., OS kernels). APUS will introduce a new consensus algorithm with an ultra-fast OS kernel bypassing technique called Remote Direct Memory Access (RDMA). Preliminary results presented in [SOSP '15] show that APUS supports unmodified real-world applications, and its latest development is 32.3X to 85.8X faster than traditional consensus protocols.

Second, we will construct a first scheduler for application availability by integrating APUS with popular schedulers. A main challenge is that existing schedulers' resource allocation schemes can be conflicting with replication schemes, because the former schemes abstract away computer identity, but the later schemes must deploy replications on different computers. Our scheduler introduces a replication-aware resource allocation scheme to resolve conflicts efficiently. Our prototype scheduler presented in [APSys '16] incurs only 3.22% performance overhead on real-world applications.

Third, we will build a new VM for application availability by integrating APUS with popular VMs. One performance problem in existing VM fault-tolerance techniques is that they need to transfer all memory modified by applications across VM replications. Our new VM integrates APUS into the hypervisor layer to enforce same application inputs across VM replications, so that VM replications maintain mostly same memory. We also propose a new algorithm to efficiently track minor different memory.

This GAIA project will greatly strengthen the reliability of many real-world applications and will benefit almost all computer users and software vendors, including many HK financial platforms. GAIA will also advance broad reliability techniques (e.g., VM migration) and infrastructures.

Long term impact:
The emergence of big data with its increasing computational demand is pushing software applications to embrace more and more computing resources. Therefore, applications now often run within a datacenter containing numerous computers. Many applications are mission-critical (e.g., financial platforms, social network platforms, and medical services), so they naturally desire both high reliability and performance.

To harness the massive computing resources within a datacenter, applications are typically ran by either of two independent infrastructures: schedulers and VMs. Depending on runtime trade-off such as performance or security isolation, some applications are ran solely by schedulers (e.g., Mesos [NSDI '11] usually uses lightweight containers, not VMs), and some other applications are ran solely by VMs (e.g., Amazon EC2).

Unfortunately, as an application runs on more computers, computer errors will occur more likely at runtime. If a failed computer happens to run an essential application component, the entire application can be turned down. A downtime often causes severe disasters, especially for online or long-running applications. For instance, due to minor computer errors, New York Stock Exchange (NYSE) was down for a whole day in 2015. Moreover, although social network applications tend to be online 24-7, minor computer errors have turned down the Facebook site for several times in recent years.

A key problem causing these disasters is that datacenter infrastructures lack a high availability support for applications. To tackle this problem, this GAIA project takes a holistic methodology: it first builds a fast consensus protocol, it then integrates this protocol into the two typical infrastructures. By doing so, applications ran by either of these infrastructures can enjoy two significant benefits.

First, the availability of mission-critical applications can be greatly strengthened. Distributed consensus is recognized a strong fault-tolerance theory because it maintains multiple consistent replications of the same application to overcome computer failures in minor replications. Although consensus consumes extra computing resources for fault-tolerance, it has been widely adopted in industry, because resource capacity is often not a bottleneck for mission-critical applications.

Second, it is easy to make fault-tolerance itself robust. Distributed consensus is notoriously difficult to understand, build, or test. For instance, although some genius companies (e.g., Microsoft) have built consensus protocols for individual applications, ironically, recent research tools have detected numerous bugs in these protocols. Building one consensus protocol for each application could be a nightmare for application developers. Fortunately, with GAIA, people only need to carefully test our protocol and infrastructures, then applications can enjoy robust fault-tolerance.

We envision significant impacts from this GAIA project in three terms.

In the near term, our fast consensus algorithm and its implementation protocol APUS (Objective 1) can drastically improve the performance of many applications that use distributed consensus. For instance, Scatter [SOSP '11], a remarkable key-value store application, deploys consensus groups using a traditional consensus protocol. By using APUS, the latency of Scatter can be one or two orders of magnitude faster.

In the intermediate term, by realizing the new infrastructures in Objective 2 and 3, GAIA will greatly strengthen the reliability of general applications and benefit almost all computer users and software vendors. For instance, HK has many financial applications which naturally demand stringent availability in operational hours, and GAIA can meet this demand. As per our tradition, we will make all source code public for further study and deployments.

In the long term, we anticipate that this GAIA project will advance broad datacenter techniques (e.g., VM migration) and attract researchers to build more reliable datacenter infrastructures. As datacenter emerges to be a "giant computer", a fast and reliable datacenter OS for such a novel computer will gradually come up. Therefore, reliable consensus protocols, schedulers, and VMs will become essential datacenter OS techniques, and the outcomes of this project will be adopted in a future datacenter OS.

Overall, due to these prospective impacts, we name our project after gaia, an ancient Greek goddess who takes good care of everything on earth, including software applications.

Objectives:

1.

[To create a fast distributed consensus protocol].

We will develop an RDMA-powered distributed consensus algorithm and its implementation protocol APUS. This new algorithm aims to be one or two orders of magnitude faster than traditional consensus protocols.

2.

[To construct a first datacenter scheduler for improving application availability].

This scheduler will replicate all or essential application components by integrating APUS with popular schedulers. We will develop a new scheme to seamlessly manage both resource allocation and replication logic, so that this scheduler can efficiently schedule applications.

3.

[To build a new fault-tolerant VM for improving application availability].

We will leverage the VM hypervisor layer to transparently enforce same application inputs across VM replications. Compared to existing VM fault-tolerance techniques (e.g., primary-backup), our VM will greatly reduce the amount of transferred memory across VM replications, saving most time and network bandwidth. We will also develop a new algorithm to efficiently track and transfer minor different memory across replications.

1 Research Background

This section presents the background of consensus (§1.1) and datacenter computing infrastructures (§1.2), motivation of objectives (§1.3), others' related work (§1.4), and PI's related work (§1.5).

1.1 Paxos Consensus

Consensus protocols (typically, PAXOS [45, 46, 49, 63]) play a core role in distributed systems, including ordering services [31, 42, 51], leader election [7, 17], and fault-tolerance [25, 33, 43]. A PAXOS protocol replicates the same application on a group of computers (or *replicas*) and enforces the same order of inputs for this application, as long as a majority of replicas are still alive. If leader fails, it elects a new leader [49]. Therefore, PAXOS tolerates various faults, including minor replica failures and packet losses.

Unfortunately, an open problem is that the consensus latency of existing PAXOS protocols is too high, because their consensus messages go through OS kernels and software TCP/IP layers (a `ping` round-trip takes about 200 μs in a 10Gbps network). §2.1.1 presents this problem in detail.

Recently, as Remote Direct Memory Access (RDMA) becomes popular in datacenter networks (e.g., Infiniband [4]), it becomes a promising solution to address the PAXOS performance problem. A fastest type of RDMA operations called "one-sided RDMA writes" (for short, *WRITE* in the rest of this proposal) round-trip takes only about 3 μs [52], because it allows a local computer to directly writes to a remote computer's memory without involing OS kernel or CPU on the remote computer. This ultra low latency not only comes from its kernel bypassing feature, but also its dedicated network stack implemented in hardware. Therefore, RDMA is considered the fastest kernel bypassing technique [41, 52, 59]; it is several times faster than software-only kernel bypassing techniques (e.g., DPDK [2] and Arrakis [58]).

1.2 Datacenter Computing Infrastructures

More and more applications are ran in a datacenter by two independent types of infrastructures. The first type is schedulers [10, 16, 36, 37, 39, 64, 65, 73]. Although existing schedulers themselves have been made available via PAXOS (e.g., [36]), their applications are not. Therefore, if failures such as computer hardware errors occur, these schedulers have to re-launch applications, leading to a substantial application downtime and thus a huge lost for mission-critical applications (e.g., financial platforms and social networks).

The second infrastructure type is VM [11, 13, 44, 57, 66]. VM abstracts away the heterogeneous physical computing resources, making computing resources easy to utilize and balance loads (e.g., via live migration [19, 55]). VM is also known for its secure isolation on resources [13, 44, 62]. Although some VM fault-tolerance approaches such as primary-backup [26] exist, they consume much time and network bandwidth as they need to transfer all memory modified by applications across VM replications.

These two infrastructures usually work independently depending on runtime trade-off such as performance or security isolation. Some applications are ran solely by schedulers (e.g., Mesos [36] usually uses lightweight Linux containers, not VMs), and some other applications are ran solely by VMs (e.g., Amazon EC2). Therefore, this GAIA project strengthens the two infrastructures respectively. If people need to run an application with both infrastructures, they can choose either infrastructure developed from GAIA and the other one from outside. For instance, people can choose our TRIPOD scheduler (§2.2.1) and VMWare.

1.3 Motivation of Objectives

GAIA objectives stem from two research problems in datacenter computing. First, despite the core role and wide deployments of PAXOS, it suffers from high consensus latency. Therefore, **Objective 1** addresses this problem by leveraging RDMA to create a fast consensus algorithm and implementation protocol. Second, although many mission-critical applications demand stringent availability, existing infrastructures lack such support. To benefit these applications, **Object 2 and 3** take a holistic methodology to integrate our protocol with two major infrastructures, potentially benefiting almost all applications.

1.4 Related Work by Others

Various Consensus Protocols. There are a rich set of PAXOS algorithms [45, 46, 49, 54, 63] and implementations [17, 18, 25, 49]. Since consensus protocols play a core role in datacenters [1, 36, 72] and worldwide distributed systems [20, 48], various works have been conducted to improve specific aspects of consensus protocols, including commutativity [54], understandability [46, 56], and verifiable rules [32, 70]. PAXOS is notoriously difficult to be fast and scalable to a large consensus group [31, 42, 51].

To make PAXOS's throughput scalable (i.e., more replicas, higher throughput), various systems leverage PAXOS as a core building block to develop advanced replication approaches, including partitioning program states [14, 31], splitting consensus leadership [15, 48], and hierarchical replication [31, 42]. Theses approaches have improved throughput. However, the core of these systems, PAXOS, still faces an unscalable consensus latency [31, 42, 51]. By realizing **Objective 1** (§2.1), these systems can scale even better.

Fault-tolerance in schedulers. Datacenter schedulers [10, 16, 36, 37, 39, 64, 65, 73] support diverse applications (e.g., Hadoop [34], Dryad [38], and key-value stores [61]). Existing schedulers mainly focus on obtaining high availability for themselves by replicating their own essential components, or focus on application recovery [73] instead of availability. To the best of our knowledge, no existing schedulers provide a high availability support to general applications. **Objective 2** (§2.2) aims to provide this support.

Fault-tolerance in VM. Two approaches, primary-backup and live migration, exist for improving application reliability in VM. Primary-backup uses the hypervisor on a primary VM to track memory pages modified during handling each request, and then transfers these pages to a backup VM. Live migration uses a similar hypervisor technique as primary-backup, and it is also used to improve computer load balance and consolidating many VMs on fewer computers to save datacenter energy. A common problem in these two approaches is that they incur substantial application downtime (e.g., 8 seconds in vMotion [55]) and network bandwidth during transfering memory. **Objective 3** (§2.3) aims to address this problem.

RDMA techniques. Recently RDMA is deployed in various types of datacenter networks, including Infiniband [4] and RoCE [6]. RDMA has been leveraged in many software systems to improve different performance aspects, including high performance computing [30], key-value stores [28, 40, 41, 52], distributed transactions [29, 68], and file systems [69]. These systems are complementary to GAIA objectives.

1.5 Related Work by the PI

The PI is an expert on improving the reliability of both datacenter software systems [25, 67] and multithreading runtime systems [21, 22, 24, 71]. The PI's works are published in premier conferences on systems software (OSDI 2010, SOSP 2011, SOSP 2013, and SOSP 2015) and programming languages (PLDI 2012 and ASPLOS 2013). As preliminary results for this GAIA proposal, the PI has developed a fast consensus protocol [25] (part of **Objective 1**) and a fault-tolerant datacenter scheduler [67] (part of **Objective 2**).

2 Research Plan and Methodology

This section first proposes APUS (§2.1), a fast consensus protocol, it then leverages APUS to construct a scheduler (§2.2) and a VM (§2.3) for application availability. Finally, it describes research plan (§2.4).

2.1 Objective 1: Building Fast Distributed Consensus via RDMA

This section describes a performance problem (§2.1.1) in existing PAXOS protocols and presents APUS (§2.1.2), a fast PAXOS protocol by leveraging RDMA.

2.1.1 Problem: consensus latency of existing PAXOS protocols are slow and unscalable

Despite the wide deployments of PAXOS (§1.1), its high consensus latency makes many software applications suffer. For efficiency, PAXOS typically assigns one replica as the leader to propose consensus requests, and the other replicas as backups to agree on requests. To agree on an input, at least one message round-trip is required between the leader and a backup. A round-trip causes big latency (hundreds of μs) as it goes through various software layers (e.g., OS kernels).

As replica group size increases, Paxos consensus latency scales poorly: it increases drastically [31] due to the linearly increasing number of consensus messages. One common approach to improve Paxos scalability is leveraging parallel techniques such as multithreading [7, 15] or asynchronous IO [25, 60]. However, the high TCP/IP round-trip latency still exists, and synchronizations in these techniques frequently invoke expensive OS events such as context switches.

Our preliminary study (Figure 3) ran four Paxos-like protocols [7, 15, 25, 60] on 40Gbps network with only one client sending consensus requests. When changing the replica group size from 3 to 9, the consensus latency of three protocols increased by 30.3% to 156.8%, and 36.5% to 63.7% of this increase was in OS kernel. The only exception is SPaxos [15] because it batches requests from backups and starts consensus when the batch is full. Mmore replicas causes shorter time to form a batch, but batching aggravates latency.

RDMA appears a promising solution (§1.1) to speed up Paxos. However, fully exploiting RDMA speed in software systems is widely considered challenging by the community [29, 41, 52, 59]. For instance, DARE [59] presents a two-round, RDMA-based Paxos protocol in a sole-leader manner: leader does all RDMA workloads and backups do nothing. Although DARE is fast with 3∼5 replicas, our study (Figure 3) shows that DARE's sole-leader nature incurred a scalability bottleneck and thus a linearly increasing latency.

2.1.2 Apus: a fast, scalable RDMA-based Paxos protocol

Our key observation is that we should carefully separate RDMA workloads among the leader and backups, especially in a scalability-sensitive context. Intuitively, we can let both leader and backups do RDMA writes directly on destination replicas' memory, and let all replicas poll their local memory to receive messages.

Although doing so will consume more CPU resources than a sole-leader protocol [59], it has two major benefits. First, both leader and backups participate in consensus, making it possible to reach consensus with only one round [49]. Second, all replicas can just receive consensus messages on their bare, local memory. An analogy is threads receiving other threads' data via bare memory, a fast and scalable computation pattern.

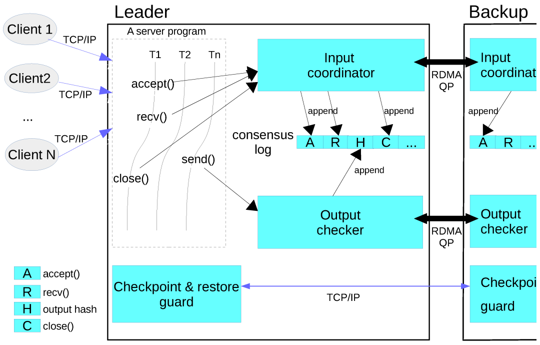

Figure 1: The Apus protocol architecture. Protocol components are shaded (and in blue color).

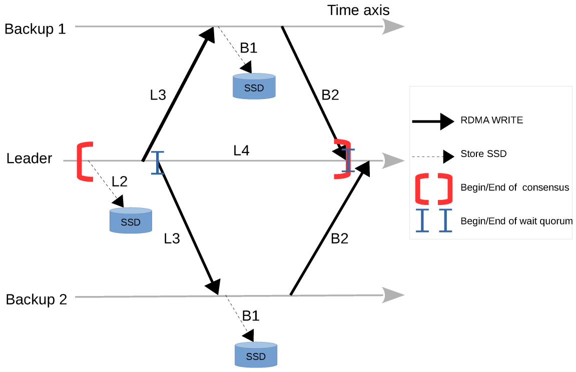

Figure 2: RDMA-powered consensus algorithm.

With this observation, we propose the design of Apus,[1] a new RDMA-based Paxos protocol and its runtime system. Figure 1 shows Apus's architecture. It is designed to support unmodified applications. Its runtime system will automatically deploy the same application on multiple replicas, intercept the application's network inputs from its inbound socket calls (e.g., `recv`) with a Linux technique called LD_PRELOAD, and invoke a RDMA-powered algorithm (Figure 2) to enforce same network inputs across replicas. Apus architecture has four key components: an input consensus coordinator and an output checker (both are built on the algorithm), an in-memory consensus log, and a guard that handles application checkpoints and recovery.

[1] We name our protocol after apus, one of the fastest birds.

Figure 2 depicts APUS's consensus algorithm in normal case. The leader first executes the actual inbound socket call (**L1**, now shown) to get actual inputs, it then invokes consensus across replicas. All solid arrows (**L3**, **B2**) are direct RDMA writes to remote replicas' memory. All replicas poll from their own bare memory to receive consensus messages. To handle replica failures, all replicas store inputs to local SSD (**L2**, **B1**). For these WRITEs, sending replicas need only copy data to local RDMA NIC and the WRITEs finish, APUS is scalable to consensus group size. In abnormal (rare) case, APUS uses an existing algorithm [49].

Preliminary results. We developed a APUS initial prototype in two phases. First, to support unmodified applications, we implemented its socket-intercepting runtime system, presented in SOSP 2015 [25]. APUS was able to support five widely used server applications (e.g., MySQL) without modifying them. Second, we implemented its RDMA-powered consensus algorithm on a 40bps RDMA network. Figure 3 shows APUS performance with 5 existing consensus protocols. To analyze the scalability of our algorithm, we ran APUS and DARE on up to 105 replicas. APUS's consensus latency outperforms 4 popular PAXOS protocols by 32.3x to 85.8x on 3 to 9 replicas. APUS is faster than DARE by up to 3.3x.

Future work. We will further improve APUS's practicality in two dimensions. First, we will evaluate its performance on broad latency-critical applications using diverse input workloads. Second, we will further torture its scalability on at least 10X more replicas, identify its scalability bottleneck, and refine our algorithm.

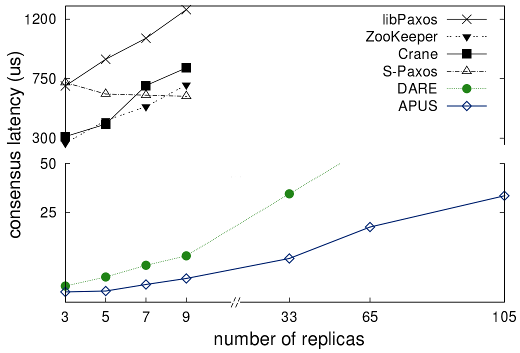

Figure 3: Consensus latency of six PAXOS protocols. Both X and Y axes are broken to fit in all these protocols. APUS achieves the smallest consensus latency.

2.2 Objective 2: building a fault-tolerant scheduler to improve application availability

Many existing schedulers have adopted PAXOS to improve availability for themselves. Typically, a scheduler use PAXOS to replicate its essential components such as a controller, which handles scheduling computation jobs to run on resources. In normal case, only one leading controller does the real work, and the other controllers act as backup to handle the leader's failure. Unfortunately, most applications running by schedulers have not been made highly-available (although individual applications carry a replication approach [12, 27]).

A strawman approach to achieve high application availability could be implementing a PAXOS protocol for every application. However, this will cause two major issues. First, PAXOS is notoriously difficult to understand [45, 56], implement [18, 49], or test [32, 70]. Developing a PAXOS protocol for every application is widely considered a nightmare [18, 32, 70] for application developers.

The second issue is, the scheduler can be conflicting with PAXOS due to unawareness of the application's PAXOS replication logic. For instance, if an application submits multiple replications of the same computation job to the scheduler, the scheduler may incorrectly schedule several replications on the same computer (it should schedule each copy on different computers to achieve PAXOS fault-tolerance).

2.2.1 TRIPOD: a first fault-tolerant scheduler architecture

This section proposes the design of TRIPOD, a scheduler infrastructure that can replicate all or essential computations of a general application. TRIPOD is formed by a widely used scheduler MESOS [36] and APUS (**Objective 1**). To avoid the two aforementioned issues (§2.2), TRIPOD's design integrates PAXOS in a scheduler, not in applications. To achieve high application availability, unlike existing schedulers which let only one controller schedule jobs, TRIPOD runs replicas of the same job using replicas of controllers: after controllers agree on a new job with APUS, TRIPOD lets each controller independently schedule a replication of this job. To seamlessly resolve conflicts between the resource allocation scheme and repliation scheme, TRIPOD introduces a replication-aware resource allocation scheme (§2.2.2).

Figure 4 depicts TRIPOD's architecture, and its key components are shaded (and in blue). To illustrate how TRIPOD works in an application perspective, this figure shows two applications, Hadoop and MPI. Each job has a *replica strength* (R) to denote the level of fault-tolerance it demands. This R value is either 1 (default, no replication) or no more than the number of controllers in TRIPOD. An application can assign other R numbers to a job when it submits this job to TRIPOD.

It is often easy for an application to determine which computation jobs require high availability and thus a bigger R value. For instance, Alluxio [9], a distributed storage application ran in MESOS, has an essential manager component to manage storages across computers. When running in our TRIPOD scheduler, Alluxio can just assign R=3 to the launching job of this manager.

Figure 4 shows an example of replicating all Hadoop jobs with R=3. Suppose Hadoop submits two jobs to TRIPOD's leader controller, each has different shapes (triangle or hexagon). The leader controller then invokes a consensus on each job across controllers. Once a consensus is reached, each controller assigns the same job on different slave machines. The leader controller directly returns its computation result to the Hadoop scheduler. A standby controller ignores computation results unless it becomes a new leader.

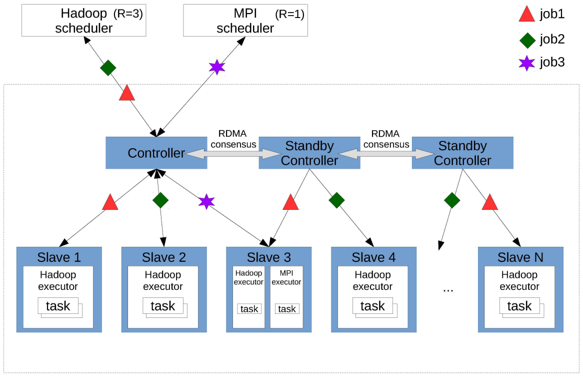

Figure 4: The TRIPOD fault-tolerant scheduler.

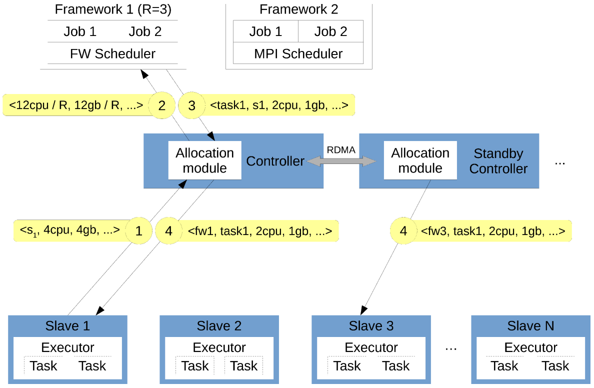

Figure 5: The replication-aware resource allocation scheme.

2.2.2 Replication-aware resource allocation scheme

Figure 5 shows the design of TRIPOD's scheme on scheduling jobs with four steps. This scheme is similar to that in MESOS except the second and fourth steps. These two steps TRIPOD abstract away the replication logic in its resource offers and allocations from the application. An application runs as if GAIA does not replicate any of its jobs, and TRIPOD transparently handles all the replication logic.

In the first step, slave machines periodically report their available computing resources (e.g., CPU cores and memory) to the leader controller. In the second step, instead of offering the available resources aggregated from slave machines, TRIPOD conservatively divides the amount of resources by each application's R value and then sends a resource offer to the application. The goal is to reserve enough resources for TRIPOD to replicate a job with R copies, although some jobs have only R=1.

In the third step, an application scheduler submits jobs to the leader controller. If R is 1, the leader schedules the job without consensus and revokes R-1 resources specified in this job from the application. By doing so, the conservatively reserved resources are given back to TRIPOD. If vvR is larger than 1, The leader controller then invokes a consensus on this job across controllers.

Once a majority of controllers agrees on executing a job, each controller does the fourth step. It schedules this job on an available slave machine according to the resource offer. To prevent controllers putting the same job on the same slave machine, the leader controller first makes an assignment on which controller should run this job on which slave machine, it then carries this assignment in its consensus request. Once a consensus on this job is reached, each controller follows this assignment.

TRIPOD is designed to make a mission-critical application highly available by using R-1 more resources than that consumed by the application's essential computations. Non-essential computations use same resources as those in MESOS. Given that minor computer failures have caused severe outgage disasters (e.g., 2015 NYSE trading halts [5] and several Facebook outage events in recent years [3]), we consider these extra resource consumptions worthwhile on improving applicaiton availability.

Preliminary results. We presented an early prototype of TRIPOD in [67]. To evaluate a typical social-networking application, we ran TRIPOD with Memcached [50], a popular key-value store used by Twitter and financial platforms [35]. Compared to Memcached's unreplicated execution, TRIPOD incurred merely a 3.22% overhead in throughput and 3.31% in response time.

Future directions. We forsee three directions for TRIPOD's further development. First, we will fully implement it and conduct an extensive study on diverse applications. Second, currently our replication and resource allocation scheme is tied to MESOS. We will study other popular schedulers, summarize general resource allocation scheme patterns, then we will develop a general, scheduler-agnostic scheme for diverse datacenter schedulers. Third, currently our scheme requires application developers to manually assign R values to application components. We will leverage our expertise on precise program analysis [22, 23] to create new schemes that can automatically analyze an application and assign precise R values to different components. We believe that all the three directions will lead to innovative work.

2.3 Objective 3: building a fault-tolerant VM to improve application availability

As mentioned in related work (§1.4), primary-backup and live migration are two similar techniques on improving VM fault-tolerance. For instance, after serving each request, the primary-backup technique transfers all memory pages modified by applications (i.e., dirty pages) from the primary to the backup. This technique consumes 2*R resources (R is the amount of resources consumed by unreplicated execution). There can be GBs of dirty bytes to be transfered (§2.3.2), causing substantial time and network bandwidth consumption. A key reason for this problem is that existing techniques have only one actual application execution.

We will use APUS to tackle this problem as it maintains multiple actual executions. By using APUS to enforce same inputs across VM replicas, most dirty pages across replicas are the same and needn't transfer.

2.3.1 Integraging APUS with KVM

We propose a new fault-tolerant VM by integrating APUS into the hypervisor layer of popular VMs. Specifically, we choose the popular KVM [44] for three main reasons. First, KVM is an open source hypervisor in Linux. Second, it provides `tap_send()`, an input capturing API in its hypervisor. This API allows APUS to intercept network inputs on the leader VM and transparently replicates them on backup VMs. Third, this KVM API runs in userspace, which allows RDMA to work (RDMA runs in userspace).

We practively design our VM to achieve same 2*R resource consumption as existing VM fault-tolerance techniques. Although PAXOS normally consumes 3*R resources, our VM design reduces this consumption by letting one backup act as the *major backup* with a high priority to become a new leader if current leader fails. The major backup agrees on and executes input requests, while the other backup acts as the *standby backup* which only agrees on requests without executing them. The major backup periodically checkpoints application states and send them to the standby backup. Checkpoints do not affect performance because the leader and the standby backup still form a majority and agree on requests rapidly. Even if the standby backup is elected a new leader, it extracts checkpoints and uses the aggreed inputs to catch up with old leader quickly.

2.3.2 A new synchronization technique to track divergent pages across VM replicas

Although our fault-tolerant VM has enforced same inputs across VM replicas, minor memory pages may still diverge across replicas (e.g., due to data races [47]). We propose a synchronization technique to efficiently track and transfer divergent pages, including (1) an efficient `log(N)` algorithm to track divergent pages, and (2) leveraging APUS's RDMA-powered protocol to efficiently and reliably compare divergent pages.

A naive way is linearly comparing all `N` dirty pages between replicas to find divergent ones, but it is as slow as prior primary-backup techniques. Instead, our algorithm uses a Merkle tree [43] to track dirty pages on each replica. As shown in Figure 6, if a physical page is modified, our VM hypervisor appends it as a leaf to a Merkle and incrementally recomputes a hash with the page content towards all parents. This tree is efficient because if the roots of two trees have same hash, all leaves (pages) are the same.

However, our VM design poses a new interesting challenge: unlike previous works that tree leaves are static (e.g., Eve [43] treats static global variables in application code as leaves), the leaves in our trees are dynamic as they are dirty pages. Because dirty pages across replicas can be different, the structures (i.e., the number of internal nodes) of Merkle trees between two replicas can be different and hard to compare.

Our synchronization technique tackles this challenge by introducing a *tree adjustment* phase. It works with four steps. First, after processing one request, leader sends a dirty page bitmap provided by KVM to the major backup. Major backup does a union on this bitmap and its own one and sends back to leader. Second, both leader and the major backup do a tree adjustment with the union bitmap (e.g., a non-dirty page in the union is still appended to the tree), then leader and the backup's trees have same leaves and same structure. Third, leader invokes **Algorithm 1** to do a `log(N)` traversal from tree root to find divergent leaves. To compare internal node hashes between leader and backup, the algorithm invokes an augmented APUS protocol as **APUS()**, which returns divergent nodes carried in the backup's consensus reply. Fourth, leader transfers divergent pages to the major backup, then they both clear trees and dirty page flags.

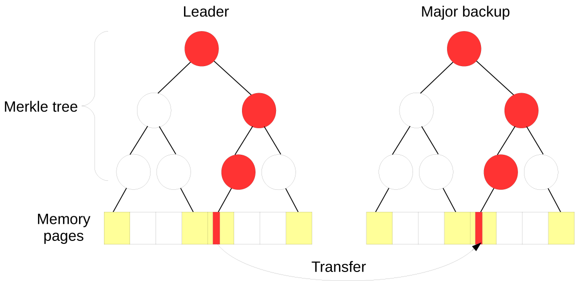

Figure 6: Data structures on tracking divergent pages across replicas. **Algorithm 1** identifies divergent pages and their parent nodes as red color.

Algorithm 1: Tracking different memory pages

```
Input  : The root of merkle tree on leader, root
Output: Different memory pages between leader and replica, pages
Track(root)
    pages ← empty sequence
    diverged ← APUS(root, root.left(), root.right())
    for i ← diverged.push_front() do
        if i is_leaf() then
            pages.append(i)
        else
            diff ← APUS(i, i.left(), i.right())
            diverged.append(diff)
    return pages
```

To check whether divergent pages are rare, we ran three appliations, MongoDB [53], Redis [61], and Memcached [50], on two KVMs with same real-world workloads [8]. We found 6,010∼14,295 application pages dirty. Primary-backup techniques have to transfer all dirty pages from primary to backup. We also found that only 471∼983 of these dirty pages diverged between two VMs. These initial results imply that **Algorithm 1** can reduce up to 96.7% transfered pages compared to prior primary-backup techniques.

Future directions. We anticipate broad outcomes from this objective by going along two directions. First, we will implement our fault-tolerant VM, compare its performance with prior VM primary-backup techniques (e.g., Remus [26]), and quantify our reduction on transfered pages. Second, we will apply our tracking algorithm to develop new lightweight VM live migration techniques (§1.4). Because VM primary-backup and migration techniques have been used in broad areas (e.g., improving resource utilization, balancing computer loads, and saving datacenter energy), our techniques can be applied in these areas, too.

2.4 Research Plan

This project will require two PhD students S1 and S2 to work for three years. In the first year, S1 will design and fully implement the APUS protocol (part of **Objective 1**), and S2 will evaluate its performance and robustness on various real-world storage applications (part of **Objective 1**). In the second year, S1 will integrate APUS to a scheduler MESOS (part of **Objective 2**), and S2 will make APUS and KVM form an eco-system (part of **Objective 3**). In the third year, S1 and S2 will respectively evaluate the efficacy of their infrastructures built from **Object 2** and **Object 3** on diverse real-world applications.

References

[1] Why the data center needs an operating system. http://radar.oreilly.com/2014/12/why-the-data-center-needs-an-operating-system.html.

[2] Data Plane Development Kit (DPDK). http://dpdk.org/.

[3] Is facebook down? a history of outages. https://www.theguardian.com/technology/2015/jan/27/is-facebook-down-outages.

[4] An Introduction to the InfiniBand Architecture. http://buyya.com/superstorage/chap42.pdf.

[5] This is why the nyse shut down today. http://fortune.com/2015/07/08/nyse-halt/.

[6] Mellanox Products: RDMA over Converged Ethernet (RoCE). http://www.mellanox.com/page/products_dyn?product_family=79.

[7] ZooKeeper. https://zookeeper.apache.org/.

[8] Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB, 2004.

[9] Alluxio - Open Source Memory Speed Virtual Distributed Storage. http://www.alluxio.org/, 2014.

[10] Tupperware. https://www.youtube.com/watch?v=C_WuUgTqgOc, 2014.

[11] Amazon Virtual Private Cloud (VPC). https://aws.amazon.com/vpc/.

[12] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI'13*, 2013.

[13] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, 2003.

[14] C. E. Bezerra, F. Pedone, and R. V. Renesse. Scalable state-machine replication. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, 2014.

[15] M. Biely, Z. Milosevic, N. Santos, and A. Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *Proceedings of the 2012 IEEE 31st Symposium on Reliable Distributed Systems*, SRDS '12, 2012.

[16] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 285–300, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL http://dl.acm.org/citation.cfm?id=2685048.2685071.

[17] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 335–350, 2006.

[18] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing (PODC '07)*, Aug. 2007.

[19] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, 2005.

[20] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI '16)*, Oct. 2012.

[21] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[22] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 337–351, Oct. 2011.

[23] H. Cui, G. Hu, J. Wu, and J. Yang. Verifying systems rules using rule-directed symbolic execution. In *Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '13)*, 2013.

[24] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nov. 2013.

[25] H. Cui, R. Gu, C. Liu, and J. Yang. Paxos made transparent. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Oct. 2015.

[26] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174. San Francisco, 2008.

[27] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, 2004.

[28] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, 2014.

[29] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Oct. 2015.

[30] M. P. I. Forum. Open mpi: Open source high performance computing, Sept. 2009.

[31] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in scatter. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011.

[32] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 265–278, Oct. 2011.

[33] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: Replication at the speed of multi-core. In *Proceedings of the 2014 ACM European Conference on Computer Systems (EUROSYS '14)*, page 11. ACM, 2014.

[34] Hadoop. Hadoop. `http://hadoop.apache.org/core/`.

[35] R. Hecht and S. Jablonski. Nosql evaluation: A use case oriented survey. *2012 International Conference on Cloud and Service Computing*, 0:336–341, 2011. doi: http://doi.ieeecomputersociety.org/10.1109/CSC.2011.6138544.

[36] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation*, NSDI'11, Berkeley, CA, USA, 2011. USENIX Association.

[37] M. Isard. Autopilot: Automatic data center management. *SIGOPS Oper. Syst. Rev.*, 41(2):60–67, Apr. 2007. ISSN 0163-5980. doi: 10.1145/1243418.1243426. URL `http://doi.acm.org/10.1145/1243418.1243426`.

[38] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the Second ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, 2007.

[39] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629601. URL `http://doi.acm.org/10.1145/1629575.1629601`.

[40] J. Jose, H. Subramoni, K. Kandalla, M. Wasi-ur Rahman, H. Wang, S. Narravula, and D. K. Panda. Scalable memcached design for infiniband clusters using hybrid transports. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, 2012.

[41] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. Aug. 2014.

[42] M. Kapritsos and F. P. Junqueira. Scalable agreement: Toward ordering as a service. In *Proceedings of the Sixth International Conference on Hot Topics in System Dependability*, HotDep'10, 2010.

[43] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, et al. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, volume 12, pages 237–250, 2012.

[44] Kernel Virtual Machine (KVM). `http://www.linux-kvm.org/`.

[45] L. Lamport. Paxos made simple. `http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf`.

[46] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[47] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Thirteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 329–339, Mar. 2008.

[48] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, volume 8, pages 369–384, 2008.

[49] D. Mazieres. Paxos made practical. Technical report, Technical report, 2007. http://www. scs. stanford. edu/dm/home/papers, 2007.

[50] Memcached. `https://memcached.org/`.

[51] E. Michael. *Scaling Leader-Based Protocols for State Machine Replication.* PhD thesis, University of Texas at Austin, 2015.

[52] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, June 2013.

[53] mongodb. Mongodb. `http://www.mongodb.org`, 2012.

[54] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, Nov. 2013.

[55] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, 2005.

[56] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, June 2014.

[57] OpenStack: Open Source Cloud Computing Software. https://www.openstack.org/.

[58] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the Eleventh Symposium on Operating Systems Design and Implementation (OSDI '14)*, Oct. 2014.

[59] M. Poke and T. Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, 2015.

[60] M. Primi. LibPaxos. http://libpaxos.sourceforge.net/.

[61] Redis. http://redis.io/.

[62] J. Sugerman. Private communication, Dec. 2005.

[63] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):42:1–42:36, 2015.

[64] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2428-1. doi: 10.1145/2523616.2523633. URL http://doi.acm.org/10.1145/2523616.2523633.

[65] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 18:1–18:17, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3238-5. doi: 10.1145/2741948.2741964. URL http://doi.acm.org/10.1145/2741948.2741964.

[66] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, 2002.

[67] C. Wang, J. Yang, N. Yi, and H. Cui. Tripod: An efficient, highly-available cluster management system. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, 2016.

[68] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, SOSP '15, Oct. 2015.

[69] G. G. Wittawat Tantisiriroj. Network file system (nfs) in high performance networks. Technical Report CMU-PDLSVD08-02, Carnegie Mellon University, Jan. 2008.

[70] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the Sixth Symposium on Networked Systems Design and Implementation (NSDI '09)*, pages 213–228, Apr. 2009.

[71] J. Yang, H. Cui, J. Wu, Y. Tang, and G. Hu. Determinism is not enough: Making parallel programs reliable with stable multithreading. *Communications of the ACM*, 2014.

[72] M. Zaharia, B. Hindman, A. Konwinski, A. Ghodsi, A. D. Joesph, R. Katz, S. Shenker, and I. Stoica. The datacenter needs an operating system. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, 2011.

[73] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu. Fuxi: A fault-tolerant resource management and job scheduling system at internet scale. *Proc. VLDB Endow.*, 7(13):1393–1404, Aug. 2014. ISSN 2150-8097. doi: 10.14778/2733004.2733012. URL http://dx.doi.org/10.14778/2733004.2733012.