# Assured Cloud-Based Data Analysis with ClusterBFT*

Julian James Stephen and Patrick Eugster

Purdue University

**Abstract.** The shift to cloud technologies is a paradigm change that offers considerable financial and administrative gains. However governmental and business institutions wanting to tap into these gains are concerned with security issues. The cloud presents new vulnerabilities and is dominated by new kinds of applications, which calls for new security solutions.

Intuitively, Byzantine fault tolerant (BFT) replication has many benefits to enforce integrity and availability in clouds. Existing BFT systems, however, are not suited for typical "data-flow processing" cloud applications which analyze large amounts of data in a parallelizable manner: indeed, existing BFT solutions focus on replicating single monolithic servers, whilst data-flow applications consist in several different stages, each of which may give rise to multiple components at runtime to exploit cheap hardware parallelism; similarly, BFT replication hinges on comparison of redundant outputs generated, which in the case of data-flow processing can represent huge amounts of data. In fact, current limits of data processing directly depend on the amount of data that can be processed per time unit.

In this paper we present ClusterBFT, a system that secures computations being run in the cloud by leveraging BFT replication coupled with fault isolation. In short, ClusterBFT leverages a combination of variable-degree clustering, approximated and offline output comparison, smart deployment, and separation of duty, to achieve a parameterized tradeoff between fault tolerance and overhead in practice. We demonstrate the low overhead achieved with ClusterBFT when securing data-flow computations expressed in Apache Pig, and Hadoop. Our solution allows assured computation with less than 10 percent latency overhead as shown by our evaluation.

**Keywords:** Cloud, Byzantine fault, replication, integrity, data analysis.

## 1 Introduction

The cloud as a computing platform is getting more popular and mature every day. Computational needs of industry, academia, and government are being increasingly met by processing data in the cloud. Recently announced government

* This work has been financially supported by DARPA grant # N11AP20014, Northrop Grumman Information Systems, Purdue Research Foundation grant # 204533, and Google Research Award "Geo-Distributed Big Data Processing".

D. Eyers and K. Schwan (Eds.): Middleware 2013, LNCS 8275, pp. 82–102, 2013.

policies [4] clearly show an urgent economic requirement for processing data in the cloud. Yet, a major roadblock to adopting cloud technologies is the lack of trust on the various facets of cloud computing. The fact that a potentially malicious entity can legally access computing resources in the same datacenter or even on the same machines as well-intended users increases the risk associated with moving computations into the cloud. Malicious programs, faulty hardware, and software bugs can lead to corrupt data or cause services to fail.

**Model.** In many scenarios, institutions can trust the cloud providers themselves, but not the users of the system. If we take the example of the US intelligence community, different agencies have their own *inhouse* clouds. They want to improve sharing of information with each other without exposing their own systems to potential weaknesses or infections in their peer systems [4]. Such a partial trust model also applies to many large corporations which include subdivisions hosting their own datacenters. Within this scenario, the present paper is concerned with ensuring (a) *integrity* and (b) *availability*, i.e., that computations indeed perform what they were supposed to (e.g., to avoid obfuscating terrorist activities), and that these computations can be performed in a timely manner (e.g., to be able to react to real threats on time). While our solution also includes mechanisms for *confidentiality* we focus on (a) and (b) in this paper.

**State of the Art.** Most approaches to cloud security focus by and large on either (a) *communication*, (b) *data*, or (c) *computation*. Communication-centric approaches (a) to security in public or inhouse clouds focus on setting up thick firewalls, which monitor in - and outbound traffic. Typically, ports that accept incoming data and specific protocols and services are allowed or disallowed based on the configuration of the firewall. Though required, such perimeter security is not sufficient to secure computations because, zero-day attacks may compromise one or many of the internal nodes. Once an internal node is compromised, it can alter computation output even without any communication across the perimeter. This can break the integrity of the system. In addition, as illustrated over and over again by the alleged organized attacks of chinese hacker groups on US installations, if there is a vulnerability in the perimeter defense system, it is very difficult to detect an ongoing attack. Data-centric approaches (b) protect data from malicious and benign failures but mainly focus on data at rest. In all functional systems, data is under constant churn. Computation adds, deletes and morphs data into new forms. Further, in many cloud storage systems data modification is replaced with data creation (append-only semantics) for performance and reliability reasons. Under such conditions, it is impossible to ensure integrity of data without assuring computations that work on data. Typical data-centric approaches focus on ensuring confidentiality when that data is accessed or computed on but do not verify computations themselves. Computation-centric approaches (c) to securing computation focus on fine-grained information-flow [17]. As with data-centric approaches, information-flow approaches aim at protecting (sensitive) data from leaking. However, they do not ensure that the computation is behaving according to specification, i.e., ensuring the computation is doing

what it was intended to. In typical cloud data-flow processing applications, where new data-sets are generated as outcome of analysis and correlation of existing data-sets and stored for subsequent use, these outcomes must be trustworthy. In fact, since in the larger picture data-sets are derived from earlier data-sets, any false results computed violate integrity of the semantic information in the original data-sets.

**BFT in Clouds.** Intuitively, *Byzantine fault tolerant* (BFT) replication [23] is a powerful means of securing computation and thus achieving integrity and availability in cloud-based computing. BFT suggests the use of multiple replicas of a sensitive component, and hinges on the comparison of outcomes produced by these replicas to determine components with erratic behavior (assuming a correct "majority"). While several fundamental assumptions of BFT replication — e.g., determinism in replicas for comparisons, possibility of exploiting redundant hardware — are largely met by typical cloud-based data-flow applications, *existing* BFT *systems* are inapplicable to such applications: these focus on securing single monolithic servers, and only little work exists on applying BFT replication beyond such stand-alone servers. Cloud-based data-flow processing systems, inversely, leverage cheap hardware by breaking down applications into small components which are amenable to parallel execution. When applying BFT replication to any one of these components by running multiple replicas of each and comparing their respective outcomes overheads sum up very quickly.

**ClusterBFT.** This paper presents ClusterBFT for cloud-based assured data processing and analysis. ClusterBFT utilizes BFT techniques which impose less overhead than existing cryptographic primitives, but breaks away from the mold of individually replicating every client request. More precisely, ClusterBFT creates sub-graphs from acyclic data-flow graphs that are then replicated. This means, rather than enduring the overhead of BFT consensus at each component involved in the data-flow processing, we have a system with much less overhead that can dynamically adapt to changes in required responsiveness and perceived threat level as well as to dynamic deployment (elasticity). We use a combination of *variable-grain clustering* with *approximated* and *offline comparison*, *separation of duty*, and *smart deployment* to keep overheads of BFT repliation low while providing good fault isolation properties. In summary, the main contributions of the paper are (1) identification of challenges and solutions for achieving availability and integrity of cloud-based data-flow computations with BFT replication, (2) the architecture and implementation of a BFT solution for such computations, and (3) the evaluation of this solution. Our evaluation shows less than 10 percent latency overhead in most cases for even complex data analysis jobs.

**Roadmap.** The remainder of this paper is structured as follows. Section 2 provides background information. Section 3 lists design principles and challenges. Section 4 describes the ClusterBFT architecture in detail. Section 5 describes its implementation. Section 6 presents evaluation results. Section 7 presents related work. Section 8 draws conclusions.

## 2 Background and Preliminaries

This section presents information pertinent to the remainder of the paper.

### 2.1 BFT

Byzantine failures [23] model arbitrary faults that may occur in a process during execution, including malign and benign faults. In order to explain our system better, we further distinguish Byzantine failures by classifying them based on how they allow deviation from correct execution. We use the categorization of Kihlstrom et al. [20] which classifies Byzantine failures as follows:

- Omission (detectable): An omission failure occurs when a process does not send a message that it is expected to send. These can be detected by setting timeouts for messages. It is important to note here that in an asynchronous system, a timeout does not necessarily imply a faulty component.
- Commission (detectable): A commission failure occurs when a process sends a message it is not supposed to send. Such failures can be detected by checking if the message is in agreement with at least *f+1* other replicas.
- Unobservable (non-detectable): Unobservable failures are those which other processes cannot detect based on the messages they receive.
- Undiagnosable (non-detectable): Undiagnosable failures are those that cannot be attributed to a specific process.

### 2.2 MapReduce and Pig

Big data analysis is one of the major use cases for moving towards cloud computing and most cloud-specific programming models reflect this. Corresponding runtime systems try to make use of large numbers of nodes available for data analysis to decrease latency. The popular MapReduce [16] framework partitions input data and assigns a mapper process to each input partition. These mapper processes produce "intermediate" key-value pairs as output which are grouped by key and fed by key to reducer processes which use these to generate final output. Hadoop [37] is a popular open source implementation of MapReduce.

Apache Pig [3] is a platform for data analysis that consists of the PigLatin [28] high-level language for expressing data analysis programs, and a runtime system. Pig Latin scripts are typically compiled to MapReduce jobs that are executed using a MapReduce engine such as Hadoop for Pig. To illustrate the benefits of our concepts, we focus in this paper on Pig data analysis jobs.

Throughout the paper, unless otherwise specified, we use the term *script* to refer to a Pig (Latin) script. We use the term *job* to refer to a MapReduce job and *task* to refer to map or reduce tasks within a MapReduce job. We use the term *job cluster* to refer to the group of nodes involved in executing a specific job.

### 2.3 System Model

We assume that the system is deployed on a cloud service that leases out virtual machines to users. We refer to one such virtual computation unit as a *node*. This means that there could be multiple nodes on the same physical machine. We assume that the number of nodes that are faulty at a given time is bounded. For the purpose of this paper we focus on computation and assume a trusted storage layer. We are aware that assuming correctness of a storage system prone to Byzantine faults is ambitious, but it is not unrealistic either. Systems like DepSky [8] show its feasibility. Further, the challenges that need to be solved even with the presence of a trusted storage are tough and warrant investigation. We present a system for two *adversary models*. For both models, we assume that the adversary cannot manipulate the cloud service provider or violate its specifications. This includes preventing communication between any two nodes, spawning new Byzantine nodes, and breaking computationally hard cryptographic primitives. A *strong adversary* can manipulate all internal aspects of a node and collude with other adversaries. This includes full control over the executing processes, physical memory and messages being sent out of the node. A *weak adversary* shares the same properties of a strong adversary, but may only cause omission or commission faults.

## 3 ClusterBFT Design

This section presents first our motivation for using BFT techniques in the cloud, before outlining challenges in such adoption and finally our solutions for overcoming these.

### 3.1 BFT and the Cloud

We decided to adopt BFT replication due to several intuitive benefits:

**Attribution:** Along with tolerating benign or malign failures, BFT techniques can also point to potentially faulty components which helps for attribution as well as auditing. Indeed, being able to shield computation from malicious entities is one thing, but in a sea of nodes such as a cloud datacenter it is also necessary to keep track of where such accesses were attempted, as these may hint to exploited leaks and intruders.

**Portability and interoperability:** BFT techniques can be applied at a higher level in the protocol stack — here typically at the level of data-flow program execution — which allows them to be deployed easily across different cloud platforms and infrastructures, thus supporting portability, cloud interoperability, and the cloud-of-clouds paradigm [38].

**Determinism:** Popularity of data-flow languages like PigLatin or DryadLINQ [40] shows the relevance of data analysis jobs that can be modeled as direct acyclic graphs (DAGs). These computations and their constituents are by-and-large deterministic, which simplifies the comparison of redundant

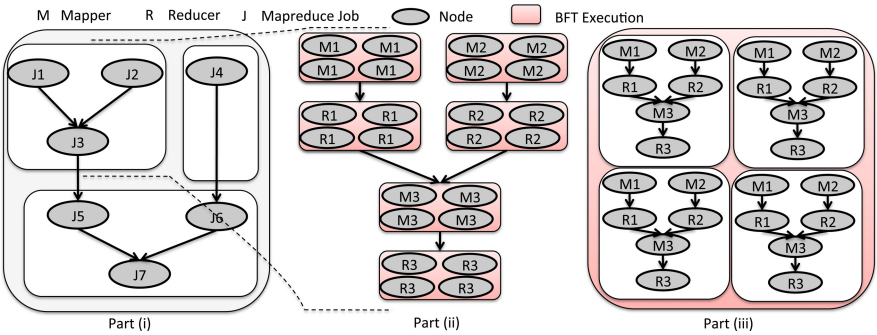


**Fig. 1.** Part (i) shows a data-flow graph with 7 phases. (ii) focuses on $n \times m$ replication of jobs J1, J2 and J3. (iii) shows clustered replication of J1, J2, J3 requiring only one round of BFT consensus. For simplicity we only show one map and one reduce task per MapReduce job.

results. Inversely, concurrent client accesses pose challenges when replicating large monolithic servers. Recent trends in cloud-based data processing include support for iterative and incremental jobs which contradict the straightforward DAG model [41] but do not hamper determinism.

**Heterogeneity:** BFT relies on heterogeneity of replicas to ensure that a majority of replicas are not compromised simultaneously by means of the same vulnerability. Cloud platforms do expose a uniform hypervisor layer on which operating systems are deployed, but cloud providers offer a variety of operating system images that can be deployed on these nodes. Within an operating system itself, adoption of address space layout randomization (ADSLR) introduces further heterogeneity. DARPA's Mission-Resilient Cloud program [25] funds several projects aiming at creating moving targets specifically to further narrow this gap [27].

### 3.2 Challenges in Adopting BFT in the Cloud

Though intuitively BFT seems like a good match in many ways for ensuring computation in the cloud, it has thus far not been adopted widely in such a context due to a variety of open challenges:

C1. *Scalability:* Datasets are typically many magnitudes higher in cloud-based programming than in previous scenarios. As BFT replication protocols hinge on comparison of redundant outcomes, this translates to large overheads.

C2. *Granularity:* Data analysis scripts also tend to have multiple *jobs* where output of one is fed to the second. This creates a *job-chain* in which a process that was a server for one job acts as a client for the second job. Ideally, every process is fine-grained and can be deployed dynamically. This means, naïve BFT replication of each job will result in $R = 3f+1$ replicas for each task, with $n \times m$ communication [31] and synchronization after every

stage. This is illustrated by Figure 1. The left part (i) shows a Pig-style data-flow graph, while the middle part (ii) illustrates the $n \times m$ interaction [31] occurring as a result of replicating every node in the (sub)graph obtained after compilation to MapReduce jobs: every edge corresponds here in fact corresponds to $4 \times 4$ interactions. This causes very high resource usage, limiting availability and increasing cost for huge data-flows.

C3. *Rigidity:* Clouds represent very dynamic environments, being marketed to meet instantaneous demands rather than having to over-provision constantly to meet occasional spikes. This calls for solutions that are flexible and can be adapted to some degree. The main knob to turn in BFT is the replication degree, which however represents a coarse granularity: typically a replication degree of $3f + 1 = 4$ with $f = 1$ already leads to substantial overhead. The next larger step, $3f + 1 = 7$, to tolerate up to 2 failures already leads to prohibitively larger overhead.

There are are also non-technical factors deterring BFT adoption in the cloud. As explained by Birman et al. [9], many cloud middleware service providers have an inherent "fear of synchronization" irrespective of the existence of fast consensus protocols and success stories like Chubby [11].

### 3.3 ClusterBFT Principles and Architecture Overview

ClusterBFT addresses the challenges C1-C3 above as follows:

**Variable Granularity:** Observe that nothing forces us to replicate every individual node in the data-flow graph. We could in fact replicate the execution of an entire data-flow graph $3f + 1$-fold, and compare the outcomes at the very end. More generally speaking, we can choose any intermediate level for clustering nodes in the graph and replicate these subgraphs (addressing C2 above). This is illustrated by the right part (iii) of Figure 1, where the sub-graph of (ii) is replicated as a whole and comparison only occurs at the end of this sub-graph. The potential downside of such regrouping is that it may diminish the degree of fault tolerance and precision of fault attribution: a single deviant node in a group hampers the outcome of that replica, and identifying *which* node(s) in the group exhibit Byzantine behavior becomes harder. In addition, if we do not end up having sufficiently many identical replica responses, it takes longer to run additional replicas thus increasing job latency. This tradeoff leads to an additional knob for users to tweak (C3).

**Variable Replication:** The BFT replication model allows control over how resources are utilized. Based on the user's confidence in the cluster, different degrees of replication can be adopted with different guarantees. A user can specify an optimistic, $f+1$ replicas. In this case, the execution ensures safety, but may require repeated runs to get correct output. If the user specifies $2f + 1$ replicas, a correct result can be guaranteed if all replicas always reply (no omission failures). If $3f + 1$ replicas are specified, a correct result can be guaranteed under combination of any kind of Byzantine failure.

**Approximate, Offline Redundancy:** Instead of comparing the entire outputs of a replica set in one go upon sub-job completion, we can choose to (1) only compare *digests*, (2) start doing so *before sub-job completion*, and (3) allow the follow-up sub-job to proceed based on the complete output *before comparison completes*. This reduces the overhead of putting redundancy to work (addressing C1) in a way allowing further fine-tuning of tradeoffs between performance and security by control of the resilience of the digests (C3).

**Separation of Duty:** Rather than baking the entire data-flow handling logic into every node, we can separate architecturally the "front-end" of a data-flow processing system which accepts jobs from the actual cluster of worker nodes such as MapReduce nodes executing the jobs. This architectural division is illustrated in Figure 2 which outlines the architecture of our solution ClusterBFT detailed in the next section. Components in the *control* tier are command and control processes that provide direction and coordinate computations in the *computation* tier. The former tier is trusted, which is achievable by BFT replication or by implicitly trusting the nodes, i.e., by closely (even manually) monitoring nodes, or using nodes in the client network or private cloud. The benefit of this separation is that it limits certain strong assumptions and expensive mechanisms to the front-end, allowing the cluster to focus on work (cf. [39]) and to be handled more dynamically (C3). This in turns allows the worker node cluster to be adapted dynamically, by adding and removing nodes based on resource requirements, measured performance, and of course suspicions.

**Fault Isolation:** Another net advantage of the separation of duty is that the front-end can keep track of suspicions observed, and can use specific deployment policies to, for instance, narrow down the (set of) faulty node(s) in a replication group delivering a faulty response by intentionally partly overlaying the replication group of a different job on the same nodes. Similarly, dummy jobs can be used to further probe nodes in such a suspicious replication group. Thus the tradeoff with attribution precision introduced by variable granularity does not become a one-way path but becomes a tradeoff with the *time* it takes to recover precision (C3).

In the following section we describe the architecture of ClusterBFT and how we put these principles to work.

## 4 ClusterBFT Architecture and Components

In this section we look at the different components that make up ClusterBFT (see Figure 2). Table 1 presents a summary of the symbols used in the following.

### 4.1 Request Handler

The *request handler* component is in the control tier. It accepts scripts submitted by the client and submits the script for execution. It consists of three logical subcomponents outlined below.

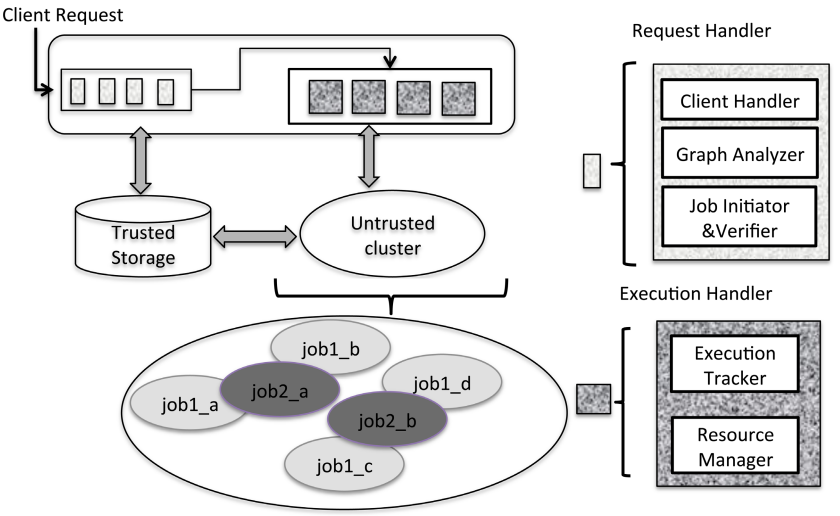


**Fig. 2.** Architecture

**Client Handler.** The client submits the script to the *client handler*. The client also specifies the number of expected failures $f$, a replication factor $r$ and the total number of verification points $n$ based on the perceived threat level. Verification points are vertices in the data-flow graph after which output from different replicas are matched. The client handler generates a logical plan from the script. This is given as input to the graph analyzer described below.

**Table 1.** Symbols

| Symbol | Meaning |
|---|---|
| $r$ | Replication factor |
| $n$ | Number of verification points |
| $f$ | Number of expected failures |
| $s$ | Suspicion level |

**Graph Analyzer.** In order to reduce overhead and improve utilization, we need to identify verification points in the data-flow graph that are most effective. Running verification after every operation will cause very high overhead and running verification scarcely will result in more re-computations (hence higher resource usage) when failures occur. The *graph analyzer* component, based on the adversary model, identifies points in the data-flow graph for performing verification. Under the strong adversary model, only points that correspond to data-flow between jobs are considered for verification. Under a weak adversary model, any point in the data flow graph can be considered for verification.

With $n$ verification points requested by the user, we use the marker function defined in Figure 3 to identify the actual points. We explain the intuition behind the marker function using an example. Consider the data-flow graph in Figure 4 and assume the user specified one verification point. If we decide to perform verification right after the vertex *Load*1, then the probability of identifying a fault is very low. There is a much higher probability that at least one of the

**Table 2.** Notation

| Notation | Meaning |
|---|---|
| $ir[v]$ | Input ratio of $v$ |
| $parents(v)$ | All parents of vertex $v$ |
| $level(v)$ | $\begin{cases} 1 & \text{if } v = Load \\ \max\limits_{p \in parents(v)} 1 + level(p) & \text{else} \end{cases}$ |
| $min(v, M)$ | Number of edges between $v$ and the vertex closest to $v$ in $M$ |

```
V                                   ▷ Vertex set
n          ▷ Number of verification points
function MARK(V, n )
    M ← ∅        ▷ Set of marked vertices
    for 1.. n do
        max ← 0
        for all v ∈ V do
            score_v ← ir[v] + min(v, M)
            if score_v > max then
                m ← v
                max ← score_v
            end if
        end for
        M ← M ∪ {m}
    end for
end function
```

**Fig. 3.** Marker function

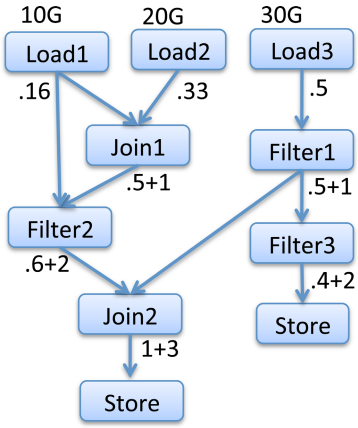


**Fig. 4.** Annotated data-flow graph

```
v                                   ▷ a vertex ∈ V
function INPUT_RATIO(v)
    if v is Load then
        ir[v] ← input_size(v) / total_input_size
    else
        ir[v] ← (Σ_{p∈parents(v)} ipr[p]) / (Σ_{level(n)=level(v)−1} ipr[n])
    end if
end function
```

**Fig. 5.** Computing input ratios

nodes that execute the vertices below *Load*1 is faulty simply because there are more of them. On the other end, if we run verification after *Join*2, then we most probably will know if result is going to be faulty, but the cost of re-computation, in case $f+1$ replicas do not agree becomes high; the entire sequence of operation needs to be recomputed. The marker function considers two main parameters to arrive at a verification point that is a good tradeoff between these two extremes. The ratio of input data that flows through a vertex and distance of a vertex from another verification point. Using these two values, the marker function arrives at a mid point suitable for verification. Once the verification point is identified, the logical plan is instrumented with a verification function and given to the *job initiator*. Details of what a verification function are described next.

**Job Initiator and Verifier.** The instrumented script gets compiled into one or more MapReduce jobs and the job initiator associates a sub graph identifier *sid* with each such job. The job initiator submits a total of $r$ replicas of the job for execution to the execution handler. All replicas are configured to have the same number of reduce tasks. The verification function instrumented into the MapReduce job uses a cryptographic hash function (SHA-256 in our prototype)

to compute a digest of the data streaming through the verification point and sends this digest to the verifier. The verifier compares corresponding digests from different replicas and asserts that at least $f + 1$ are same. The verifier is also responsible for isolating failures and updating the suspicion level $s$ for each node.The suspicion level of a node is defined as total number of faults associated with the node divided by the total number of jobs executed on the node. For clarity, details of fault isolation is specified as a separate section (4.2), after we introduce the remaining components in our architecture.

### 4.2 Execution Handler

Figure 6 shows the internals of the *execution handler* and how it interacts with the request handler.

**Execution Tracker.** The job submitted by the request handler is executed by the execution tracker. Resources available in nodes are partitioned into uniform resource units *ru*. A list of all resources is initially loaded from an administrator-provided inclusion list into the *resource table* as a tuple $\langle nid, \#ru, \langle sid...\rangle, s\rangle$. One tuple represents a node id *nid*, the number of resource units *ru* in that node, the current allocation of *sid*s and suspicion level *s* of a node. When the job initiator submits a job, the job is first added to the job queue. The main sequence of operations that take place after this is shown in Figure 6 (others are omitted for simplicity), and detailed below:

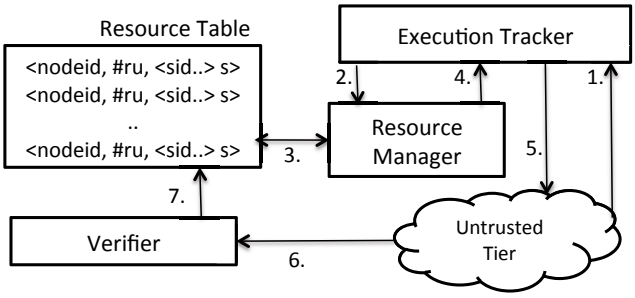


**Fig. 6.** Execution tracker & resource manager

1. A node in the untrusted domain with id *nid* sends a heartbeat message to the execution tracker.
2. The execution tracker checks with the *resource manager* to see if there is a task that can be scheduled on node *nid*.
3. The resource manager queries the resource allocation table to retrieve the *sid*s of tasks currently running on node *nid*. Using this, the resource manager looks at the list of running or submitted jobs to check if a there is a task from a job that does not already have a task running on node *nid*.
4. The resource manager provides a list of ready tasks corresponding to the number of free *ru*s in node *nid*.

5. The execution tracker replies to the heartbeat message with the task that needs to be executed.
6. During task execution, the verification function creates a message digest of data streaming through the verification point and sends the digest to the verifier. The verifier checks for $f+1$ matching message digests from different replicas. If the verifier times out without obtaining $f+1$ matching message digests, the job is initiated again with a higher value for $r$.
7. Based on the number of non-matching digests, the verifier updates the suspicion levels of node *nid.* in the resource table.

**Resource Manager.** We already outlined how the resource manager functions as part of the working of the execution tracker. There are two goals that we try to achieve by proper task selection: efficient execution and fast fault identification. Data local tasks enable faster execution. For fast fault identification job clusters can be overlapped in specific patterns. The scheduling strategy we use is to cause as many intersections as there are resource units in a node. That means if one node has three resource units, we try to pick tasks from three different jobs to execute. Other strategies can also be used to overlap clusters which we intend to explore in future work. The administrator can also configure a suspicion threshold such that if $s >$ threshold, then the resource manager will remove that node from its inclusion list and ignore further requests from that node. At this point administrators can intervene to re-initialize the node by taking the node off the grid, applying securing patches and reinserting the node.

```
1:  D ← A set of disjoint sets.
2:  O ← A set of overlapping sets.
3:  function FAULT ANALYZER(S)                ▷
    S, the set of nodes in a cluster that just
    returned a commission fault.
4:      if ∀X | X ∈ D, S ∩ X = ∅ then
5:          D ← {S} ∪ D
6:      else if ∃Y | Y ∈ D and S ⊂ Y then
7:          D ← D \ {Y}
8:          O ← O ∪ {Y}
9:          D ← D ∪ {S}
10:     else
11:         O ← O ∪ {S}
12:     if |D| = f then
13:         for each X ∈ D do
14:             A ← A ∪ X
15:         for each X ∈ O do
16:             X ← X ∩ A
17:         for each X ∈ D do
18:             for each Y ∈ O do
19:                 if X ∩ Y ≠ ∅ then
20:                     I ← I ∪ (X ∩ Y)
21:             if | I |= 1 then
22:                 D ← D \ {X}
23:                 D ← D ∪ {I}
24: end function
```

**Fig. 7.** Fault analyzer function

### 4.3 Fault Identification and Isolation

As described in Section 4.2, the output verifier collects output digests and asserts that at least $f+1$ digests are the same. If the verifier receives an incorrect digest or

does not receive a digest from nodes executing the data-flow, the suspicion level of all involved nodes is updated. This means if there is a faulty node that is part of multiple job clusters, that faulty node is likely to have a higher suspicion level. Once the verifier identifies a job cluster as returning incorrect result, the *fault analyzer* function in Figure 7 is used to further narrow down the list of suspicious nodes. The fault analyzer works in two stages. In the first stage disjoint subsets of suspicious nodes are isolated. This set of subsets is denoted by $D$ (line 1). This is done until the number of such subsets becomes equal to the highest value of $f$ the system has seen so far (line 12). This allows us to identify subsets of nodes such that there is exactly one fault per subset. The second stage (lines 13-23) reduces the number of nodes in these subsets by creating the intersection of a subset with other sets of faulty nodes. The intuition for the second stage is that if there are $f$ subsets in $D$ and a new set of faulty nodes intersects with only one of those $f$ subsets, then the nodes in the intersection must be faulty.

Byzantine behavior also means an infected node may be mostly producing correct output, and produce incorrect results occasionally. This means if nodes show malicious intent/fail frequently, fault isolation becomes faster.

## 5 Implementation

This section presents our prototype implementation of ClusterBFT. ClusterBFT is implemented in Java by modifying Hadoop 1.0.4 [19] and Pig 0.9.2. For instrumenting the Pig logical plan we modified the Penny [29] monitoring tool, distributed as part of Pig 0.9.2 source.

### 5.1 Hadoop

Hadoop uses a centralized *job tracker* and *task tracker*s on each computation node. The job tracker initiates a MapReduce job and task trackers spawn map or reduce tasks for the job, and send heartbeat messages and job status updates to the job tracker. It is relevant here to note that Hadoop allocates resources in a node as *task slots*. Each node may have multiple task slots depending on the number of CPU cores and physical memory available for processing. Typically 3-4 slots can be configured on a node with 4 CPU cores.

### 5.2 Request Handler

Penny consists of Penny agents that are inserted between Pig script states. These agents in turn are implemented as user defined functions that can exchange messages with other agents and a Penny coordinator. Our changes involve creating a verifying function as a Penny tool that creates a SHA-256 digest and sends the digest back to the coordinator in the trusted tier. We modified the Penny infrastructure to allow creation of multiple coordinators, so that different replicas can reply back to different coordinators.

### 5.3 Execution Handler

We implement the resource manager by creating a new task scheduler that extends the `TaskScheduler` class in Hadoop. Hadoop allows creation of multiple *job queues* to which jobs can be submitted. In ClusterBFT each replica of a job can be submitted to one queue. In order to tolerate faulty nodes, we also need to ensure that tasks from more than one replica of a job are not scheduled on a same node at any point of time. Such a collocation could result in one faulty node modifying the outcome of more than one replica and thus violating safety. Note that this does not prevent us from collocating tasks from different jobs on the same node. We added data structures to the `JobInProgress` class that will keep track of replica information to prevent this during task scheduling. We also added a new alphanumeric parameter `sub.graph.id` to the `JobConf` class. `sub.graph.id` corresponds to *sid* in Section 4.1 and is set during job initiation. All replicas of a single job must have the same `sub.graph.id`. `JobTracker` itself works without any modifications as our execution tracker.

### 5.4 Ensuring Determinism

The data parallelism leveraged by MapReduce may naturally lead to non-determinism, which can be observed through differing digest values across replicas even without faulty processes. For example in order to calculate an average, instead of finding the sum of all values of a key and dividing it by the number of values, users may decide to maintain a moving average, causing final outputs to differ (in the least few significant bits of precision). Our current prototype works around this issue by ensuring that the user programs deal with only integer values or truncate the last few decimal points before performing arithmetic operations. For a more general solution we intend to address this issue in future work by ordering the intermediate mapper output based on mapper ids.

## 6 Evaluation

To assess the benefits of our approach, we evaluate (a) overhead incurred by ClusterBFT, (b) gains of ClusterBFT under different replication degrees in the presence of failures (c) effectiveness of fault isolation algorithm and (d) system performance for higher approximation accuracy.

**Setup.** For evaluations in Section 6.1 and 6.2, we use planet-lab based Vicci [1] as our testbed. Machines are 12-core Intel Xeon servers with 48GB RAM virtualized using Linux containers. Our untrusted tier consists of 32 nodes and our trusted tier consists of 2 nodes. We use Amazon EC2 for the evaluation in Section 6.4 with 8 nodes in the untrusted tier and 4 nodes in the trusted tier.

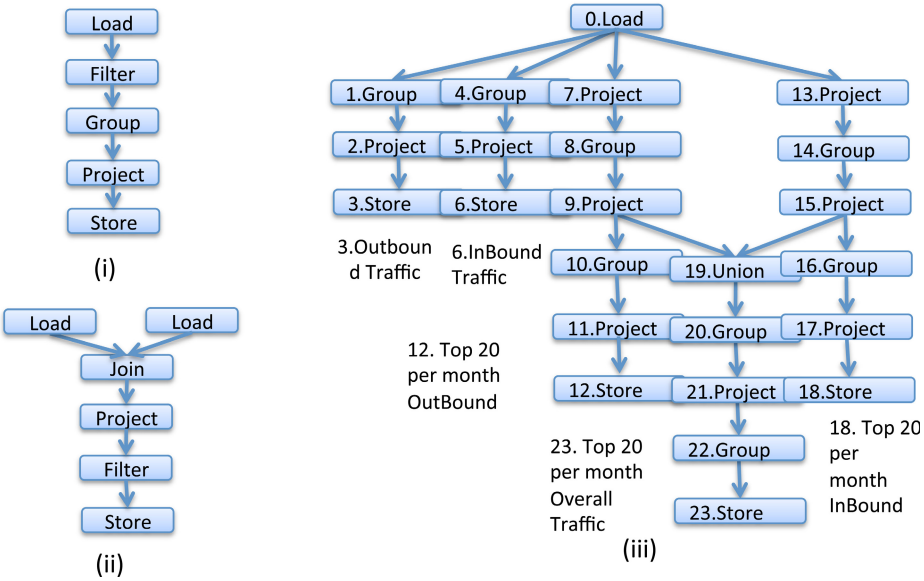


**Fig. 8.** Data-flow graph for (i) Twitter Follower Analysis (ii) Twitter Two Hop Analysis, (iii) Air Traffic Analysis

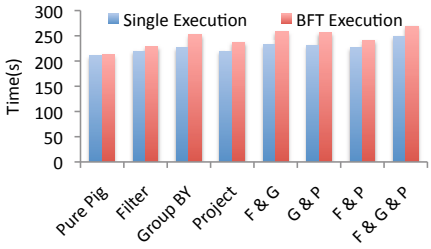


**Fig. 9.** Latency of running Twitter Follower Analysis


**Fig. 10.** Digest computation overhead for Twitter Two Hop Analysis

### 6.1 Verification Overhead: Twitter Data Analysis

First we measure the overhead involved in computing digests required for verification. For this we use the Twitter data-set from [22] and compute *SHA-256* digests at different points for two Pig scripts. The data-set consists of two columns, *user-id* and *follower-id* represented as numeric values. We run two Pig scripts outlined in [6]. The first script (Twitter Follower Analysis ) counts the number of *followers* for each user. It loads the data, filters out empty records, groups the record by *user-id*, calculates the counts and saves the *user-id* and respective counts. The second script (Twitter Two Hop Analysis) lists pairs of users that are are two hops away from one another. This job does a self-join that matches

one user with all its *follower's followers*. The data-flow graphs for these two scripts are presented in Figure 8 (i) and (ii) respectively. Figures 9 and 10 shows the total time taken for job completion when digests are computed at different points of the respective jobs. In both graphs, *Single Execution* shows the time taken by a single replica of the script and *BFT Execution* shows the time taken by 4 replicas of the script to execute. *BFT Execution* also includes the overhead of matching $f+1$ digests generated by the replicas. *Pure Pig* shows the baseline run with no verification points or replication. When digests are computed at multiple points in the data-flow graph, it is abbreviated using the first letter of the verification point. When digests are computed at multiple points in the data-flow graph, it is abbreviated using the first letter of the verification point. Figure 9 show a minimal overhead of 8% and worst case of 9%, 14% and 19% overhead with 1, 2 and 3 verification points respectively.

**Table 3.** ClusterBFT in the presence of Byzantine failures

| | $r$ =2 | | $r$ = 3, case 1 | | $r$ = 3, case 2 | | $r$ = 4 | |
|---|---|---|---|---|---|---|---|---|
| Measure | C | P | C | P | C | P | C | P |
| Latency (s) | 1.6× | 2.1× | 1.1× | 1.1× | 1.6× | 2.1× | 1.1× | 1.1× |
| CPU time spent (ms) | 3.5× | 4.1× | 3.1× | 3.1× | 4.5× | 6.2× | 4.2× | 4.2× |
| File read (Bytes) | 3.6× | 4× | 2.6× | 3× | 4.7× | 6× | 3.6× | 4× |
| File write(Bytes) | 3.4× | 4× | 2.4× | 3× | 4.7× | 6× | 3.4× | 4× |
| HDFS write (Bytes) | 2× | 4× | 2× | 3× | 2× | 6× | 3× | 4× |

### 6.2 Performance Under Failures: IRTA Airline Traffic Analysis

Next we look at ClusterBFT's performance in the presence of node failures. The input data-set for this evaluation is a 1.3GB subset of airline data-set provided by RITA [2]. We run a multi-store query outlined in [6] that finds the top 20 airports with respect to incoming flights, outgoing flights, and overall. The data-flow graph for this script is shown in Figure 8 (iii). The evaluation is set up for $f = 1$ and we show the benefits of ClusterBFT under various replication degrees with 2 verification points. We compare ClusterBFT (C in Table 3) with modified version of Pig which verifies digest of the final output only and not anywhere else in the data-flow graph (P in Table 3). The results are shown in terms of a multiplier over a single run of standard Pig without replication or digest computation. For both executions (C and P), one node was set up to always produce commission failures resulting in an incorrect digest. Also for $r = 3$, we took two measurements. The first measurement (case 1) shows results when all computations got done within the verifier timeout value. The second measurement (case 2) shows one correct replica not responding within the verifier timeout causing the script to be scheduled again with higher timeout value. Results show that latency decreases by 23% ($r = 2, r = 3$ case 2 ) for test runs that require rescheduling. For runs that do not require rescheduling, our latency is on par with running multiple replicas, and show up to 14% reduced overhead.

### 6.3 Effectiveness of Fault Isolation: Simulation

Next we evaluate the fault analyzer algorithm outlined in Figure 7. We wrote a Java-based simulator that mimics resource allocation in a 250 node Hadoop cluster. Each node is given 3 slots on which tasks can be scheduled. We consider jobs as falling under three categories: *large* (requiring 20 to 30 slots), *medium* (10 to 15 slots) and *small* (3 to 5 slots). The exact number of slots is determined uniformly at random. Each job is also associated with a unit of time as length. We studied the algorithm under various ratios of *small*, *medium* and *large* jobs as well as various length for jobs. We present a subset of our results here. Figure 11 shows the average number of jobs that got completed when the number of disjoint faulty sets ($D$) becomes equal to $f$ (Figure 7 line 12). This point is important because the number of suspicious nodes will not increase after this point. We show measurements for two ratios of job sizes and two values of $f$. Job size ratio $r1$ indicates $|large| : |medium| : |small| = 6 : 3 : 1$ and r2 indicates $2 : 2 : 1$. For $f = 1$, we used 4 replicas and $f = 2$, we used 7 replicas. The abscissa shows the probability with which a faulty node produces a commission failure. This result shows that if a node produces commission faults with very high probability, then by the time 10 jobs complete execution, we can isolate the fault to a much smaller subset. If a node produces commission faults with probability of .6 or more, less than 20 jobs are required to isolate the fault. The size of these subsets indicate the number of suspicious nodes, and this is explored in Figure 12 and Figure 13. In order to understand the number of nodes suspected by the algorithm and the suspicion level ($s$) of these nodes, we group suspicion level into four categories: no suspicion, *Low* (with $0 < s <= 0.33$), *Med* ($0.33 < s \leq 0.66$) and *High* ($0.66 < s \leq 1$). The goal of the algorithm is to narrow the suspicion down to fewer nodes, or in other words, we should have

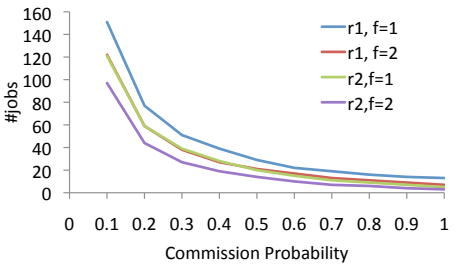


**Fig. 11.** Number of jobs required to identify disjoint set of faults

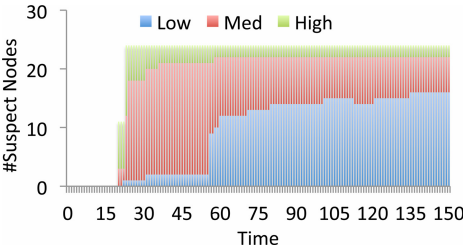


**Fig. 12.** Suspicion level changes over time

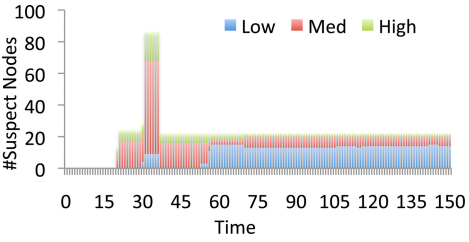


**Fig. 13.** Suspicion level spike as a result of multiple large clusters with faulty nodes

less nodes with high value for $s$. Figure 12 shows how $s$ changes with time. The initial values ($Time < 15$) indicates that no job has so far showed a commission fault. After this point we see that the number of nodes with $s > 0$ increases. It is also worthwhile to note that at around $Time = 25$, $|D|$ becomes equal to $f$ and the number of nodes with $s > 0$ does not increase further. The graph clearly shows that nodes start with *High* and *Med* suspicion levels, but over time, the suspicion levels of faulty nodes remain *High*, and of others are reduced. In fact, in these trials, by $Time = 50$, only the real faulty nodes were left in the *High* suspicion category. In Figure 13, we show occasional spikes in the number of suspicious nodes that we observed in some of the runs. This happens before $|D|$ becomes equal to $f$. This is because it may so happen that two replicas of *large* jobs show commission fault and all nodes in them gets a non zero value for $s$. But within a few more runs the algorithm prunes the suspicion list and increasingly suspects the real faulty nodes as can be seen when $Time > 35$.

### 6.4 Approximation Accuracy: Weather Average Temperature

Here we test how ClusterBFT performs if we increase the approximation accuracy from the default, one digest at one verification point, to multiple digests at each verification points. For this experiment we move away from the assumption of implicit trust within the trusted tier and instantiate $3f + 1$ replicas of the request handler. We use the BFT-SMaRT [5] library for achieving Byzantine fault tolerance within these request handler replicas. Input data for this experiment is a 640MB subset of the Daily Surface Summary of Day weather data [26]. The script involves finding average temperate over multiple years for each weather station followed by counting the number of stations with the same average. We take measurements for different values of $f$ and change the number of lines $d$ for which a digest is created. Figure 14 shows the results. In the figure, *Full* refers to script execution with digest computed and verified only for the output. *ClusterBFT* refers to using ClusterBFT with 2 verification points and *Individual* refers to digest computed for each vertex of the data-flow graph. Results show that latency overhead of ClusterBFT is within 10-18% of full replication even with increasing approximation accuracy.

## 7 Related Work

**BFT.** Works like PBFT [12], Q/U protocol [7] and HQ Replication [15] show how to make BFT s in general practical. Libraries like UpRight [13], BFT-SMaRt [5] and EBAWA [35] make it practical for anyone to efficiently and quickly implement some of these systems. Recent work like Zyzzyva [21] (based on Fast Paxos [24,18]) further improve the performance and efficiency of some of these solutions. All these solutions focus on replicating monolithic servers and do not provide parameterizable tradeoffs between overhead and fault tolerance. Yin et al. [39] separate request ordering from request execution in BFT server

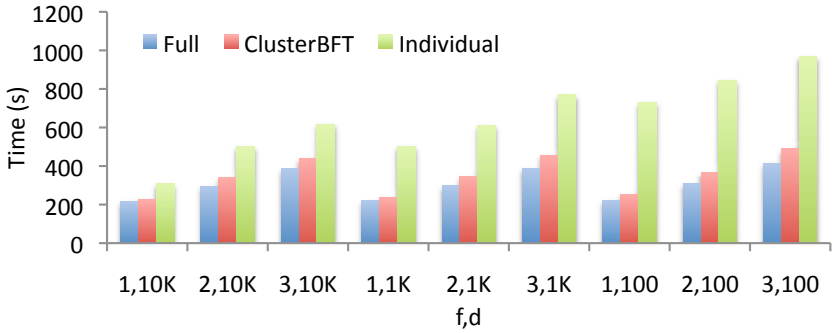


**Fig. 14.** Computing average weather temperatures

replication; we separate our architecture based on differences in the guarantees offered by nodes and not for consistency (no mutable shared state).

**BFT in Cloud.** With respect to cloud-based computations, Byzantine Fault Tolerant Mapreduce [14] explores executing Byzantine fault tolerant MapReduce jobs in the cloud and tries to reduce overhead by only starting $f + 1$ replicas of map and reduce tasks. Byzantine fault tolerance is achieved by restarting map and reduce tasks if $f + 1$ replicas do not agree on the output. This reduces the overhead when failures are not frequent but does not reduce the number of synchronization points required during job execution. BFTCloud [42] tries to secure generic computations run on voluntary resource clouds, but does not look at data-flow job specific optimizations. None of these solutions try to reduce the number of consensus instances required, or to actively identify faulty nodes by overlapping job clusters. ClusterBFT also provides parameterizeable tradeoffs between overhead and performance.

**Verifiability.** Works like Pepper [34] and Ginger [36] show that output verifiability is becoming more practical. These systems allows the computation initiator to encode the computation in such a way that it is possible to verify the result using the computation output and key. Pinocchio [30] further reduces the overhead involved and allows public verifiability. Even with these considerable improvements, these systems incur an overhead that is linearly proportional to the complexity of the computation. These systems are also limited with respected to computations involving dynamic looping constructs; requiring the programmer to inform the compiler how far the loop should be unrolled.

**Confidentiality.** We do not target to address confidentiality in this paper, but look at systems that preserve data privacy that can use ClusterBFT to secure computations. Airavat [33] adds operating system level mandatory access control to MapReduce to provide differential privacy. This allows untrusted mappers to work on sensitive data. It is possible to merge ClusterBFT with this system as they operate on mutually exclusive subsystems of Hadoop. sTile [10] distributes the input, output and intra-computation data across multiple nodes in the cloud, making it prohibitively costly for an attacker to piece together meaningful

information. CryptDB [32] preserves privacy by executing queries directly over encrypted data in a *centralized* database.

## 8 Conclusion

We presented the design and evaluation of ClusterBFT, a system for assured data processing and analysis. ClusterBFT achieves its objectives with practical overheads by using variable-degree clustering, approximated output comparison, and separation of duty. We are working towards providing confidentiality by using ClusterBFT for analyzing data encrypted using partially homomorphic cryptosystems.

**Acknowledgements.** We are very grateful to Larry Peterson for making it possible for us to evaluate ClusterBFT on Vicci.

## References

1. A programmable cloud-computing research testbed, `http://www.vicci.org`
2. Airline Data, `http://stat-computing.org/dataexpo/2009/the-data.html`
3. Apache Pig, `http://pig.apache.org`
4. Department of Defense Information Enterprise Strategic Plan (2011-2012), `http://dodcio.defense.gov/docs/DodIESP-r16.pdf`
5. High-performance Byzantine Fault-Tolerant State Machine Replication, `https://code.google.com/p/bft-smart/`
6. Pig Lab, `https://github.com/michiard/CLOUDS-LAB/wiki/Hadoop-Pig-Laboratory`
7. Abd-El-Malek, M., Ganger, G.R., Goodson, G.R., Reiter, M.K., Wylie, J.J.: Fault-scalable Byzantine Fault-tolerant Services. In: SIGOPS OSR, pp. 59–74 (2005)
8. Bessani, A., Correia, M., Quaresma, B., André, F., Sousa, P.: DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. In: EuroSys 2011 (2011)
9. Birman, K., Chockler, G., van Renesse, R.: Toward a Cloud Computing Research Agenda. SIGACT News, 68–80 (2009)
10. Brun, Y., Medvidovic, N.: Keeping Data Private while Computing in the Cloud. In: CLOUD 2012 (2012)
11. Burrows, M.: The Chubby Lock Service for Loosely-coupled Distributed Systems. In: OSDI 2006 (2006)
12. Castro, M., Liskov, B.: Practical Byzantine Fault Tolerance. In: OSDI 1999 (1999)
13. Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., Riche, T.: Upright Cluster Services. In: SOSP 2009 (2009)
14. Costa, P., Pasin, M., Bessani, A., Correia, M.: Byzantine Fault-Tolerant MapReduce: Faults are Not Just Crashes. In: CloudCom 2011 (2011)
15. Cowling, J., Myers, D., Liskov, B., Rodrigues, R., Shrira, L.: HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In: OSDI 2006 (2006)
16. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM, 107–113 (2008)
17. Denning, D.: A Lattice Model of Secure Information Flow. Commun. ACM 19(5) (1976)
18. Dutta, P., Guerraoui, R., Vukolic, M.: Best-Case Complexity of Asynchronous Byzantine Consensus. Tech. rep., EPFL (2005)
19. Hadoop: Hadoop, `http://hadoop.apache.org/`

20. Kihlstrom, K.P., Moser, L.E., Melliar-Smith, P.M.: Byzantine Fault Detectors for Solving Consensus. The Computer Journal, 16–35 (2003)
21. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: Speculative Byzantine Fault Tolerance. In: SOSP 2007 (2007)
22. Kwak, H., Lee, C., Park, H., Moon, S.: What is Twitter, a Social Network or a News Media? In: WWW 2010 (2010)
23. Lamport, L., Shostak, R., Pease, M.: The Byzantine Generals Problem. ACM Trans. Prog. Lang. and Sys., 382–401 (1982)
24. Lamport, L.: Lower bounds for asynchronous consensus. In: Schiper, A., Shvartsman, M.M.A.A., Weatherspoon, H., Zhao, B.Y. (eds.) Future Directions in Distributed Computing. LNCS, vol. 2584, pp. 22–23. Springer, Heidelberg (2003)
25. MRC: DARPA-BAA-11-55: I2O Mission-oriented Resilient Clouds (MRC), `https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-11-55/listing.html`
26. NCDC: weatherdata snapshot, `http://aws.amazon.com/datasets/2759`
27. Newell, A., Obenshain, D., Tantillo, T., Nita-Rotaru, C., Amir, Y.: Increasing Network Resiliency by Optimally Assigning Diverse Variants to Routing Nodes. In: DSN 2013 (2013)
28. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: PigLatin: A Not-so-foreign Language for Data Processing. In: SIGMOD 2008 (2008)
29. Olston, C., Reed, B.: Inspector Gadget: A Framework for Custom Monitoring and Debugging of Distributed Dataflows. In: SIGMOD 2011 (2011)
30. Parno, B., Gentry, C., Howell, J., Raykova, M.: Pinocchio: Nearly Practical Verifiable Computation. Cryptology ePrint Archive, Report 2013/279 (2013)
31. Pleisch, S., Kupsys, A., Schiper, A.: Preventing Orphan Requests in the Context of Replicated Invocation. In: SRDS 2003 (2003)
32. Popa, R.A., Redfield, C.M.S., Zeldovich, N., Balakrishnan, H.: CryptDB: Protecting Confidentiality with Encrypted Query Processing. In: SOSP 2011 (2011)
33. Roy, I., Setty, S., Kilzer, A., Shmatikov, V., Witchel, E.: Airavat: Security and Privacy for MapReduce. In: NSDI 2010 (2010)
34. Setty, S., McPherson, R., Walfish, A.J.B.: M.: Making Argument Systems for Outsourced Computation Practical (Sometimes). In: NDSS 2012 (2012)
35. Santos Veronese, G., Correia, M., Bessani, A., Lung, L.C.: Ebawa: Efficient byzantine agreement for wide-area networks. In: HASE 2010 (2010)
36. Setty, S., Vu, V., Panpalia, N., Braun, B., Blumberg, A.J., Walfish, M.: Taking Proof-based Verified Computation a Few Steps Closer to Practicality. In: Security 2012 (2010)
37. Shvachko, K., Hairong, K., Radia, S., Chansler, R.: The Hadoop Distributed File System. In: MSST 2010 (2010)
38. Verissimo, P., Bessani, A., Pasin, M.: The TClouds Architecture: Open and Resilient Cloud-of-Clouds Computing. In: DSN Workshops 2012 (2012)
39. Yin, J., Martin, J.P., Venkataramani, A., Alvisi, L., Dahlin, M.: Separating Agreement from Execution for Byzantine Fault Tolerant Services. SIGOPS OSR, 253–267 (2003)
40. Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U., Gunda, P., Currey, J.: DryadLINQ: a System for General-purpose Distributed Data-parallel Computing using a High-level Language. In: OSDI 2008 (2008)
41. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In: NSDI 2012 (2012)
42. Zhang, Y., Zheng, Z., Lyu, M.R.: BFTCloud: A Byzantine Fault Tolerance Framework for Voluntary-Resource Cloud Computing. In: CloudCom 2012 (2012)