Tolerating Faults in Disaggregated Datacenters

Amanda Carbonari
University of British Columbia
acarb95@cs.ubc.ca

Ivan Beschasnikh
University of British Columbia
bestchai@cs.ubc.ca

ABSTRACT

Recent research shows that disaggregated datacenters (DDCs) are practical and that DDC resource modularity will benefit both users and operators. This paper explores the implications of disaggregation on application fault tolerance. We expect that resource failures in a DDC will be fine-grained because resources will no longer fate-share. In this context, we look at how DDCs can provide legacy applications with familiar failure semantics and discuss fate sharing granularities that are not available in existing datacenters. We argue that fate sharing and failure mitigation should be *programmable*, specified by the application, and primarily implemented *in the SDN-based network*.

1 INTRODUCTION

Today's datacenters (DCs) are server-centric: users rent servers with specific hardware capabilities tailored to their needs (e.g., compute-intensive EC2 instances from Amazon). A *disaggregated* datacenter (DDC) disaggregates, or separates, the resources in a traditional DC into resource *blades*, with each resource connected directly to an interconnect (Figure 1). We use the term *blade* to describe a 1U server containing one resource type, and *resource* for an individual CPU, DIMM, SSD, etc. in a blade.

The modularity of DDCs benefits both operators and users [20]. By separating resources into blades, a DDC provides efficiency: the operator can upgrade specific hardware blades without impacting other resources types. Users also experience an efficiency win: in a DDC, a user can provision the exact amount of resources they require and dynamically expand/shrink this set. This flexibility also increases DDC resource utilization.

We expect DDC designs to make it possible for resource failures to no longer *fate-share* [14]. For example, a failure of a memory resource will not cause the failure of the CPU using that memory resource. In a traditional DC, applications take on the responsibility of dealing with server failures, making

HotNets-XVI, November 30–December 1, 2017, Palo Alto, CA, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5569-8/17/11...$15.00
https://doi.org/10.1145/3152434.3152447

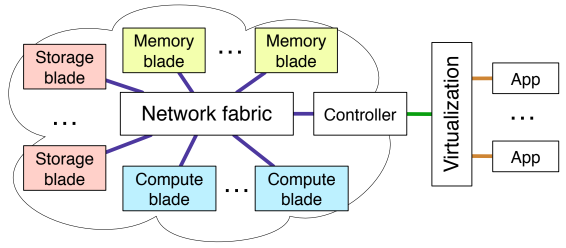

Figure 1: Simplified representation of a DDC.

them more complex. But, such legacy applications are not disaggregation-aware, and should not be expected to handle individual resource failures in a DDC. In a DDC with no fate sharing, what failures should and should not be exposed to applications? And, should failures that are visible to applications be presented as VM failures, resource failures, or something else? We believe that a key design choice for DDCs is the *granularity, or domain, of resource fate sharing*. Figure 2 illustrates two fate sharing granularities that are appropriate for legacy applications: VM-level (complete fate sharing) and process-level (partial fate sharing).

A DDC should not, however, merely emulate legacy fate sharing models; DDCs offer a unique opportunity to design new fault tolerance approaches, including new fate sharing models (Figure 2). For example, a file system that prioritizes strong consistency over availability may want to fate share a CPU resource that serializes operations with all but one SSD resource. This system will retain strong consistency guarantees even if the serializing mechanism fails by retaining a single (serializing) SSD resource, and failing the other SSD resources.

Considering a broad diversity of fate sharing models, we argue that fate sharing granularity and failure mitigation should be *programmable*, allowing applications to choose the most appropriate fate sharing arrangement and recovery model.

This failure programmability can be implemented in the application or the (software defined) network. We argue that the network is the right place for this programmability. First, failure detection and mitigation can be more easily and efficiently implemented by the network, especially in a DDC, where the network observes all inter-resource communication. Second, the network is a natural place to enforce DDC resource fate sharing.

Moving failure programmability into the network appears to contradict the end-to-end principle [40]. We are not, however, proposing to remove application-level failure handling entirely; for example, our envisioned mechanisms will not

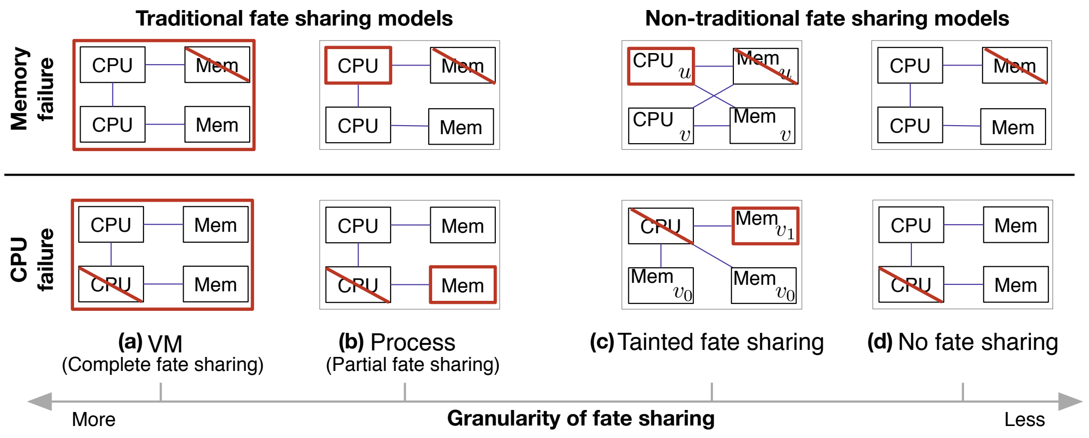

Figure 2: The memory (top row) and CPU (bottom row) failure semantics on a fate-sharing spectrum in a DDC context. A red slash denotes a failed resource. Red bolded boxes are fate-sharing failure(s) induced by the failed resource.

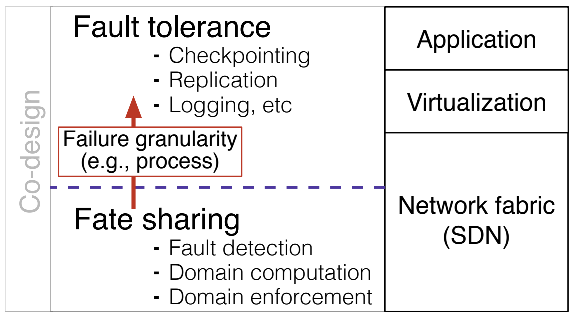

Figure 3: Fate sharing and fault tolerance relationship.

handle a broad range of faults such as byzantine failures (these must still be handled by the application). But we do argue that, in line with the end-to-end principle, certain resource faults can be more *efficiently* dealt with by the network.

2 BACKGROUND AND ASSUMPTIONS

Disaggregation can be partial or full. In partial disaggregation compute blades have a small amount of local memory, memory and storage blades have CPUs, and a NIC is attached to each resource. In full disaggregation each resource is independent and directly connected to the rack interconnect. Current research focuses on the practicality of partial disaggregation [1, 20, 32, 33].

We consider partial disaggregation at rack-scale: an application is restricted to resources in one rack (e.g., a CPU in rack1 cannot connect to a DIMM in rack2), and layer 1 provides connectivity. The disaggregated rack appears to the client as a single machine through a virtualization layer. Applications can request a number of VMs to allocate among the racks. Distributed applications might then run among several disaggregated racks, while a single-machine application will reside in one rack.

We only consider CPU and memory crash failures as these are the easiest to reason about. We do not consider storage (e.g., SSDs, NVMes) as it has different requirements. We assume that individual resources can fail within a *blade*. For example, a memory *blade* might consist of four 8GB DIMMs, if one of the DIMMs fails, the rest will continue to operate.

A software defined networking (SDN) controller has a global view of the network and can dynamically reconfigure routing [19, 25, 41]. Both features are useful in managing DDC resources. We assume that the controller and the switch can be implemented such that the controller is notified of new resources, the health of each resource, etc. Therefore, the controller can collect information about each resource in the rack. The dynamic flow rule changes will allow the network to react to events, such as failures or resource additions.

To recover from failure the DC needs to detect the failure before the application. On some fabrics failure detection can be automated [23]. If the fabric does not support automatic detection, heartbeats can be used as in e.g., [8, 13]. This scheme has been used in other SDN failure recovery work [31].

3 DDC FATE SHARING MODELS

We consider two types of fate sharing models: *traditional* models expose failures present in traditional DCs, while *non-traditional* models only exist in DDCs. In this section we detail two models from each category (Figure 2).

In our discussion we consider fate sharing separately from fault tolerance, although in practice these are co-designed. We believe it is instructive to consider this co-design from the bottom up (Figure 3), with fate sharing (implemented in the network) exposing a certain granularity, or failure domain, to a fault tolerance mechanism that can reside at different, higher, levels in the stack.

3.1 Traditional fate sharing models

Popular DC applications like Hadoop and Spark can recover from server and process failures [17, 46]. DDCs can support

Fate sharing		**Complete fate sharing**	**Partial fate sharing**	**Tainted fate sharing**	**No fate sharing**
Fault tolerance		**None**	**Mem repl.**	**None**	**Mem repl., CPU checkp.**
Modularity		Low	Medium	High	High
App.	Cost ($)	Traditional DC	Increased	Traditional DC	Increased
	Complexity	Traditional DC	Traditional DC	Low	Low
	Availability	Traditional DC	Depends on app	Depends on app	High
	Performance	Traditional DC	Improved	Slightly worse	Worse

Table 1: Trade-offs for four fate sharing granularities (Figure 2) instantiated with a particular fault tolerance scheme relative to traditional DC baseline (mem repl. is memory replication and CPU checkp. is CPU checkpointing).

these and other legacy application with traditional fate sharing models that emulate the failures that can occur in today's DC: VM failure and process failure. Table 1 summarizes the trade-off comparison between these models. We will overview these models and how SDN can be used to enforce them in a DDC.

3.1.1 *Complete fate sharing.*

Today's applications expect fate sharing between resources. Complete fate sharing (Figure 2(a)) emulates this in a DDC as a VM failure: when a resource becomes unreachable, it and the resources connected to it are isolated.

Enforcing fate sharing. Using SDN to isolate failures has been explored in previous work [31]. A complication for this model is when other CPU blades in the VM instance are communicating with the failed CPU or an external device. These other blades must fail along with the failed CPU and do so before a different failure granularity is externally observable. We believe that a key challenge for this model is providing such *atomic* fate sharing.

Fault tolerance techniques. Previous work considers several strategies for high-availability (HA) VMs [11, 16], usually implemented without support from the network. Other strategies from distributed systems fault tolerance work are also applicable [9, 21, 42, 45]. If this model is instantiated without fault tolerance, then the application observes failures as VM failures.

Summary. Table 1 compares complete fate sharing with no fault tolerance against a traditional DC along several dimensions. We think that single machine applications, such as GraphLab [35], which do not require high availability benefit the most from a complete fate sharing model. This model provides failure semantics that match current application failure assumptions: full server failure. However, complete fate sharing does not allow applications to take advantage of DDC modularity, which reduces DDC benefits.

3.1.2 *Partial fate sharing.*

Partial fate sharing (Figure 2(b)) exposes "process" failures to the application by reducing the failure domain to just the CPU and its attached memory. The main implementation decision in partial fate sharing is the handling of memory failures, which can either fate share with the CPU or be transparently recovered. Transparent recovery allows application to benefit from the decoupling of memory and CPU, but has higher overhead.

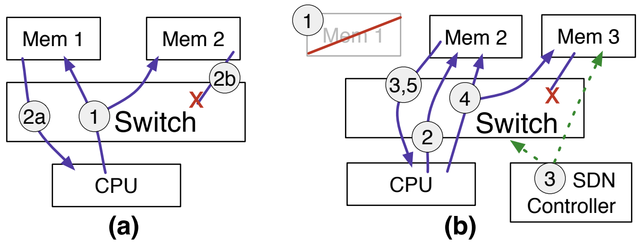

Figure 4: RAID-style memory replication. (a) Memory replication under normal operation. (b) Mem 1 fails, causing the controller to reroute requests to Mem 2 and initialize a replacement for Mem 1 (concurrent with CPU to Mem 2 traffic).

Both implementations require isolation of memory when the corresponding CPU fails.

Enforcing fate sharing. To expose a process failure, the system must atomically fail the memory corresponding to a failed CPU. This ensures that no stale state persists after the application observes the failure. As in complete fate sharing, the controller will detect the CPU failure, and install switch rules to block all traffic to and from the CPU and its corresponding memory (doing this atomically and consistently is, again, the key challenge).

Fault tolerance techniques. Tolerating memory faults has been considered in prior work on distributed shared memory (DSM) [2, 15], remote memory systems [18, 22, 37], and distributed file systems [21, 43].

These systems use logging and replication to tolerate remote memory failures. These prior approaches can be revisited and pushed into the network to provide transparent memory fault tolerance. For example, we can implement a RAID-style approach [38] by having the controller provision replica memory resources to receive identical traffic from the compute resource with port mirroring (previously used in network monitoring [39]). During normal operation (Figure 4(a)), the switch mirrors all in-bound memory traffic to a replica. Each memory resource acknowledges each operation and the switch drops acknowledgments from the replica. When a memory resource fails (Figure 4(b)), the switch reroutes to a replica. While the replica is servicing requests, the controller is notified of the failure and provisions a new replica and bootstraps it with the data from the first replica.

The challenges of fault tolerant remote memory in prior distributed systems work are directly relevant to DDCs. For instance, some systems forgo fault tolerance because of performance overheads [28, 36]. Prior work has also attempted to reduce overheads, e.g., with lightweight logging [15]. DDCs encounter the same issues but at a larger scale since all memory accesses[1] occur over the network. Another concern for DDCs, which has been considered previously, is replication cost. Log-based storage and replicas on secondary storage (i.e., SSDs) [18, 37], are options to improve crash recovery times and reduce this cost. Further investigation into how prior remote memory fault tolerance designs can be ported to DDCs and implemented in the network is necessary.

Summary. Table 1 compares partial fate sharing with memory replication to a traditional DC. The benefit of this model is that it presents the application with familiar process failure semantics and the option for in-network handling of memory failures.

By elevating resource failures to process failures, partial fate sharing reduces the number of failures other parts of the stack must handle and, consequently, improves performance. Many existing applications handle process failures. For example, MapReduce restarts completed map tasks after server failure because the results are on disk [17]. With partial fate sharing, only the running tasks will appear as failed, so MapReduce will only be exposed to task failures and not restart completed tasks (in contrast with complete fate sharing). Applications with existing and custom fault tolerance schemes benefit most from this model and will experience fewer failures.

3.2 Non-traditional fate sharing models

Disaggregation expands fate sharing options compared to a traditional DC. In particular, it makes new, dynamically computed, fate sharing domains possible.

3.2.1 Tainted Fate Sharing. Consider the case where CPU_1 writes all zeros to a memory resource (*Mem*). CPU_2 then does some computation and overwrites *Mem* with values. CPU_1 fails, what should happen to *Mem*? If the memory resource must fail with CPU_1, since CPU_2 uses that memory, CPU_2 must also fail. This raises research questions around how to determine the boundary at which the network enforces fate sharing in a dynamic context.

One formulation of the boundary that can help in the above scenario is fate sharing between tainted resources, or *tainted fate sharing* (Figure 2(c)). This model fails a memory resource only if it has not acknowledged the most recent write from the failed CPU. In the case above, since CPU_1 is not the most recent write to *Mem*, *Mem* will continue to operate and the failure domain will only include CPU_1. If CPU_2 dies before *Mem* acknowledges the write, then *Mem* will be included in the failure domain. At this point, there is a decision, should the system also fail CPU_1 because it has used *Mem* in the past? A yes answer can create a cascading failure. The application will need to define a clear point where the "taint" will stop persisting and recovery can happen.

Enforcing fate sharing. To enforce dynamic fate sharing, the network must determine the failure domain once a resource fails. When a resource fails, the controller is notified of the failure by the switch. The controller then computes the failure domain (discussed in Section 4) and creates rules to isolate all resources in that domain.

Fault tolerance techniques. Fate sharing models with dynamically computed domains require new fault tolerance techniques. In the above example, the scope of the fate sharing domain varies based on the workload of the system (i.e., if CPU_1 touches every memory resource then all will fate share). Therefore, the recovery scheme must also be dynamic, though it can still build on existing techniques, such as checkpointing (described in Section 3.2.2). In this case, the memory and CPUs can be checkpointed and recovered by the network. The tainted CPUs and memory will need to be rolled back to a previous checkpoint and restarted.

Summary. Non-traditional fate sharing models are enticing because of their dynamically-determined failure domains. The fault tolerance scheme could likewise be dynamic. This flexibility allows for more expressive designs. But, the implications of dynamically-determined failure domains and fault tolerance on performance, complexity, human error (i.e., poorly written failure domain code), are open questions that must be investigated. Table 1 captures our attempt at analyzing these trade-offs.

3.2.2 No fate sharing. In a no fate sharing design (Figure 2(d)), the DDC has two options: recover from both memory and CPU failures transparently or expose these directly to the application. We believe this choice should be made by the application.

Enforcing fate sharing. The DDC plays a minimal role in enforcing no fate sharing since this is the most natural fate sharing model for disaggregation. The network enforces isolation of the failed resource as in previous models, by dropping all packets to and from the failed resource. The network does, however, play a key role in recovery if the application elects to have network-based fault tolerance.

Fault tolerance techniques. For this discussion, we assume the same memory failure recovery scheme as partial fate sharing (Section 3.1.2). To recover from CPU failures, each CPU will asynchronously checkpoint its state, such as the program counter and registers, to a remote memory resource, as in prior work on transparent checkpointing libraries [5, 12], HA VMs [11, 16], and application checkpointing [42, 45].

In a partially disaggregated setting most remote memory operations must traverse the network[1]. In the case of local memory data modification, we assume that the change will eventually be paged out to remote memory as the local address space is a cache. Therefore, the network can record a sequence of operations that a CPU resource has performed by observing its traffic. Certain metadata must be included in the packets, such as the program counter and the running process.

Checkpointing state has been used in prior work. Transparent checkpointing libraries such as [5, 12] implement checkpointing protocols at the network layer (TCP) or message passing layer (MPI). These solutions naturally integrate into the network layer for disaggregation and can be improved with more information about process state that traverses the network in a DDC.

The combination of both memory and CPU failure recovery at a lower level is similar to work on HA VMs [11, 16, 34], which transparently handle failures for applications; however, in a DDC context, the virtual layer would also be unaware of the failure.

Summary. As compared to traditional DCs (Table 1) the no fate sharing model allows the application to take full advantage of DDC modularity. The cost to applications depends on how they handle resource failures (i.e., replication incurs more cost for extra memory). For comparison Table 1 considers no fate sharing with complete recovery.

More broadly, no fate sharing with recovery obviates the need to write and maintain application-specific fault tolerance and recovery code. For example, HERD has no fault tolerance scheme and cannot handle any type of crash failure [26]. If it were to run in a no fate sharing DDC with recovery, it will continue to operate in the case of failure because the network transparently handles all recovery. This makes applications less complex and more available. But it poses a challenge for legacy applications that come with fault tolerance built-in.

4 PROGRAMMABLE FATE SHARING

We previously noted several applications that benefit from a particular choice of a fate sharing model. For example, MapReduce would perform better with partial fate sharing, whereas HERD would perform better with no fate sharing and in-network recovery. We argue that a fate sharing model should not be chosen for the entire DDC. Instead, fate sharing should be programmable and selected by the application. We review a high-level workflow that describes one way this programmability can work in practice:

(1) When a user provisions machines for an application, they submit a partly compilable fate sharing specification (spec). This spec defines the kind of fate sharing and failure mitigation the application expects from the network.
(2) The spec is passed to the SDN controller, which uses it to provision resources for the VMs and the fault tolerance scheme, if any. The controller then compiles the spec and installs it on the switch. The controller also adds basic forwarding and failure detection rules that implement a base DDC policy.
(3) The switch monitors the running application, when it detects a failure it forwards the relevant information about the failure to the controller. The controller determines the correct fate sharing failure domain and installs proper rules based on the spec.

[1]In partial disaggregation, memory accesses only traverse the network if local memory is saturated. In full disaggregation, all accesses traverses network.

The spec should be checked for basic correctness. But, a more complex research question is to verify that the spec provides what the application expects (that the model in the spec satisfies certain safety/liveness properties). This may build on existing techniques for verifying network configurations [6] and distributed systems with pluggable failure models [44].

4.1 Failure specification requirements

The fate sharing *spec* provides an interface between the switch, controller, and application to describe the failure domain. We propose that this spec should be written in a high-level language which compiles down to a flexible protocol for the switch. Flexible protocols and high-level languages for switches exist [4, 10], so we do not discuss the details of the language here, just the requirements of the spec.

Applications must provide enough information to the DDC network for the network to enforce the chosen fate sharing granularity. A key way to characterize fate sharing is the failure *domain* for each resource: what other resources fail when a particular resource fails.

The spec must define the behavior of both the controller and the switch during three scenarios: monitoring, notification, and failure mitigation.

Passive application monitoring. Passive application monitoring defines what information must be collected during normal execution to inform failure recovery. This portion of the program must define a domain table, application protocol headers, and context information. The domain table maps a resource to context information. The program defines the format for this table as well as the logic to match entries when building a failure domain. The application headers expose application protocol information to the switch, which sends that information to the controller. The context information contains what is necessary to determine the fate sharing domain and, if required, carry out failure mitigation; this is highly dependent on the application.

For example, the tainted fate sharing model described in Section 3.2.1 will define the domain table to be a mapping between CPU resources and used memory, with the time the write started and the time the memory acknowledged a completed write:

```
domain: {cpu_ip, memory_ip, start, ack}
```

The application protocol headers will expose the type of request to the switch (read or write). Based on the headers, the switch will send the context information to the controller:

```
if (req.rtype == CPU && req.op == WRITE):
  ctl.notify(req.src, req.dst, req.tstamp, 0)
else if (req.rtype == MEMORY && req.request == ACK)
  ctl.notify(req.src, req.dst, 0, req.tstamp)
```

The switch will continue to forward information to the controller, allowing it to keep track of the context information for each resource. The controller will update each entry accordingly as it receives new information from the switch. In the case of tainted fate sharing, this allows the controller to keep a record of memory tainted by CPUs.

Application failure notification. The fate sharing spec can also define notification semantics, e.g. [30]. Since the network already performs failure detection on resources, the application can simply request to be notified of failure. This eliminates the need for the application to write failure detection code. For example, the distributed memory system using the tainting fate sharing spec may want to be notified about memory failures so it can perform its own failure recovery. In this case, it will add a notification action to the fate sharing spec:

```
def on_failure():
  if (failure_resource.rtype == MEMORY):
    app.notify("Memory " + i + " failure.")
```

The controller will provide an API for notification (i.e., `app.notify` call). The application will then listen for that notification on a predetermined port.

Active failure mitigation. When failure occurs, the active failure mitigation portion of the program initiates. It defines how to generate a failure domain and what action to take. The failure domain is created by the controller by comparing every `domain` entry to the failed resource. The comparator function determines if the `domain` entry should be considered a part of the domain or not.

In the tainted fate sharing example, the program will define the failure domain to be the failed CPU and every memory the CPU wrote to before failure that has not acknowledged the completed write:

```
def comparator(fResource, domain_entry):
  if (fResource.src == domain_entry.cpu_ip):
    if (fResource.tstamp < domain_entry.ack &&
        fResource.tstamp > domain_entry.start):
      fDomain.add(domain_entry.memory_ip)
```

The mitigation action can either isolate the failed resources (`fDomain`), for example by dropping all traffic to and from the resource, or can be a program-defined action. The tainted fate sharing example isolates the resources for CPU failure, therefore it will drop packets for the match entry. When a failure occurs, the controller adds all `domain` entries that match to a domain list using the comparator function. For each IP in the domain list, it creates a new rule which performs the `mitigation_action` on the exact match.

5 OPEN RESEARCH QUESTIONS

Programmable fate sharing in DDCs opens up several directions for research.

Other resource types. We do not address other server components in this paper such as GPUs and storage devices. As with CPU and memory we believe that in a DDC setting these devices can also benefit from a reevaluation of fate sharing and failure semantics.

What are the appropriate failure semantics for GPUs? The failure semantics for GPUs can conform to the same failure semantics as CPUs. But can it be more advantageous to have more GPU-specific failure semantics and options? GPUs have different performance and bottlenecks (such as memory copying) than CPUs.

What are the appropriate failure semantics for storage? Unlike volatile memory, data in storage persists after restart. This requires different failure recovery schemes and complicates fate sharing. For example, applications often layer their storage, and each layer might assume a different granularity of fate sharing and use a distinct fault tolerance scheme.

Other failures. Our focus so far has been on crash failures, but other failures must be considered and mitigated for DDCs to be viable, particularly at scale.

What happens if the switch or the controller fails? Switch or controller failure will render the system inoperable (Figure 1). There has been research on switch fault tolerance in datacenters [3, 24]. These techniques can be applied to DDCs. There has also been work on fault tolerant controllers [7, 27, 29]. Future research should consider how a controller can provide fault tolerance and itself remain fault tolerant.

How to handle correlated failures? Programmable fate sharing offers an opportunity for the DDC to expose correlated failures more gracefully to applications. However, this places a new burden on the application, which must now define a fate sharing domain and failure semantics under a variety of circumstances. This is a broad challenge to our proposal and requires further research in tools, languages, and libraries that lower the programmability complexity bar for applications.

6 CONCLUDING REMARKS

For DDCs to gain adoption they must provide usable notions of fault tolerance to applications. DDCs can do this by transparently recovering from resource failures or by exposing these to the application in a form that it can handle. We believe a key design criteria for DDCs to consider is the resource fate sharing granularity.

To provide the full benefit of disaggregation to a range of applications we argue that fate sharing granularity and failure mitigation should be (1) programmable, allowing applications to choose the most appropriate fate sharing arrangement and recovery model, and (2) primarily implemented in the network. We hope that this paper will inspire further discussion about failure abstractions in DDCs.

Acknowledgments. This research is supported by an NSERC discovery grant. We would like to thank Rachit Agarwal, Mihir Nanavati, and the anonymous reviewers whose feedback helped to refine our ideas and improved the paper.

REFERENCES

[1] Intel, Facebook Collaborate on Future Data Center Rack Techologies, 2013. https://newsroom.intel.com/news-releases/intel-facebook-collaborate-on-future-data-center-rack-technologies/.

[2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *SOSP*, 2007.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[4] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic Foundations for Networks. In *POPL*, 2014.

[5] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, 2009.

[6] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A General Approach to Network Configuration Verification. In *SIGCOMM*, 2017.

[7] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *HotSDN*, 2014.

[8] M. Bertier, O. Marin, and P. Sens. Implementation and Performance Evaluation of an Adaptable Failure Detector. In *DSN*, 2002.

[9] N. Bonvin, T. G. Papaioannou, and K. Aberer. A self-organized, fault-tolerant and scalable replication scheme for cloud storage. In *SoCC*, 2010.

[10] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *CCR*, 2014.

[11] T. C. Bressoud and F. B. Schneider. Hypervisor-based Fault Tolerance. In *SOSP*, 1995.

[12] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated Application-level Checkpointing of MPI Programs. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.

[13] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM*, 1996.

[14] D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *SIGCOMM*, 1988.

[15] M. Costa, P. Guedes, M. Sequeira, N. Neves, and M. Castro. Lightweight Logging for Lazy Release Consistent Distributed Shared Memory. In *OSDI*, 1996.

[16] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *NSDI*, 2008.

[17] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[18] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *NSDI*, 2014.

[19] N. Feamster, J. Rexford, and E. Zegura. The Road to SDN: An Intellectual History of Programmable Networks. *CCR*, 2014.

[20] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network Requirements for Resource Disaggregation. In *OSDI*, 2016.

[21] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *OSDI*, 2003.

[22] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient Memory Disaggregation with Infiniswap. In *NSDI*, 2017.

[23] W. L. Guay, S. A. Reinemo, O. Lysne, T. Skeie, B. D. Johnsen, and L. Holen. Host Side Dynamic Reconfiguration with InfiniBand. In *ICCC*, 2010.

[24] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers. In *SIGCOMM*, 2008.

[25] M. Jarschel, T. Zinner, T. Hoßfeld, P. Tran-Gia, and W. Kellerer. Interfaces, attributes, and use cases: A compass for SDN. *IEEE Communications Magazine*, 52(6):210–217, 2014.

[26] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-value Services. In *SIGCOMM*, 2014.

[27] N. Katta, H. Zhang, M. Freedman, and J. Rexford. Ravana: Controller Fault-tolerance in Software-defined Networking. In *SOSR*, 2015.

[28] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, 1994.

[29] H. Kim, M. Schlansker, J. R. Santos, J. Tourrilhes, Y. Turner, and N. Feamster. CORONET: Fault Tolerance for Software Defined Networks. In *ICNP*, 2012.

[30] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Improving Availability in Distributed Systems with Failure Informers. In *NSDI*, 2013.

[31] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Taming Uncertainty in Distributed Systems with Help from the Network. In *EuroSys*, 2015.

[32] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *ISCA*, 2009.

[33] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level Implications of Disaggregated Memory. In *HPCA*, 2012.

[34] J. R. Lorch, A. Baumann, L. Glendenning, D. T. Meyer, and A. Warfield. Tardigrade: Leveraging Lightweight Virtual Machines to Easily and Efficiently Construct Fault-tolerant Services. In *NSDI*, 2015.

[35] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Framework for Parallel Machine Learning. In *UAI*, 2010.

[36] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-Tolerant Software Distributed Shared Memory. In *USENIX ATC*, 2015.

[37] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP*, 2011.

[38] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD*, 1988.

[39] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: Millisecond-scale Monitoring and Control for Commodity Networks. In *SIGCOMM*, 2014.

[40] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 1984.

[41] S. Sezer, S. Scott-Hayward, P. Kaur Chouhan, B. Fraser, D. Lake, C. Systems Jim Finnegan, N. Viljoen, N. Marc Miller, N. Rao, and S.-h. Layout. Are We Ready for SDN? Implementation Challenges for Software-Defined Networks. *Future Carrier Networks*, 51(7), 2013.

[42] Y. Shen, G. Chen, H. V. Jagadish, W. Lu, B. C. Ooi, and B. M. Tudor. Fast Failure Recovery in Distributed Graph Processing Systems. *VLDB*, 2014.

[43] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *OSDI*, 2006.

[44] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *PLDI*, 2015.

[45] C. Xu, M. Holzemer, M. Kaul, and V. Markl. Efficient fault-tolerance for iterative graph processing on distributed dataflow systems. In *ICDE*, 2016.

[46] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *NSDI*, 2012.