Disaster-Tolerant Storage with SDN[⋆]

Vincent Gramoli [1] Guillaume Jourjon [2] Olivier Mehani [3]

[1] NICTA and University of Sydney, Australia
vincent.gramoli@sydney.edu.au
[2] NICTA, Australia
guillaume.jourjon@nicta.com.au
[3] NICTA, Australia
olivier.mehani@nicta.com.au

Abstract. Cloud services are becoming centralized at several geo-replicated datacentres. These services replicate data within a single datacentre to tolerate isolated failures. Unfortunately, the effects of a disaster cannot be avoided, as existing approaches migrate a copy of data to backup datacentres *only after* data have been stored at a primary datacentre. Upon disaster, all data not yet migrated can be lost.
In this paper, we propose and implement *SDN-KVS*, a disaster-tolerant key-value store, which provides strong disaster resilience by replicating data *before* storing. To this end, SDN-KVS features a novel communication primitive, SDN-cast, that leverages Software Defined Network (SDN) in two ways: it offers an SDN-multicast primitive to replicate critical update request flows and an SDN-anycast primitive to redirect request flows to the closest available datacentre. Our performance evaluation indicates that SDN-KVS ensures no data loss and that traffic gets redirected across long distance key-value store replicas within 30 seconds after a datacentre outage.

1 Introduction

With the advent of cloud services, the computation needed by individuals is progressively becoming geo-centralised in datacentres. While effective in terms of management and costs, this centralisation puts services at risk in the face of disasters, such as earthquakes or nuclear power plant explosions, which can affect large geographical regions. Few years ago, these risks motivated Wall Street financial institutions to build datacentres outside the blast radius of a nuclear attack on New York City, creating a ring of land in New Jersey called the "Doughnut",[4] illustrated in Figure 1. Located within a range of 30 to 70 km from the city centre, these backup datacentres aim to maintain rapid data transfer with the CBD to mitigate disasters.

[⋆] NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

[4] http://www.datacentreknowledge.com/archives/2008/03/10/new-york-donut-boosts-nj-data-centers/

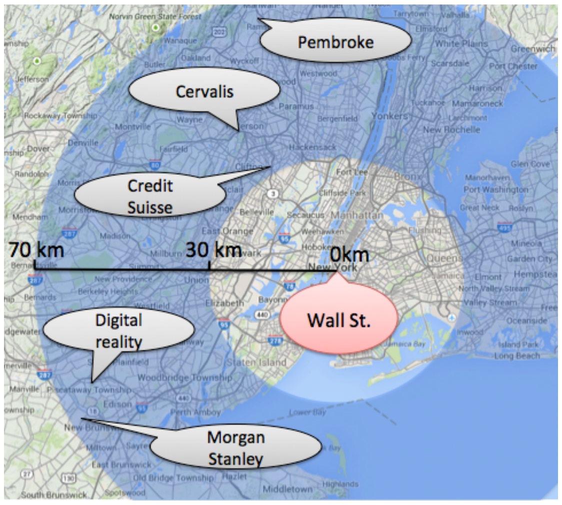
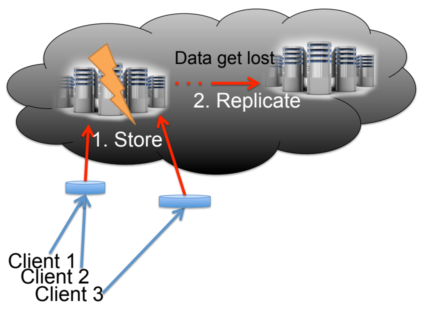


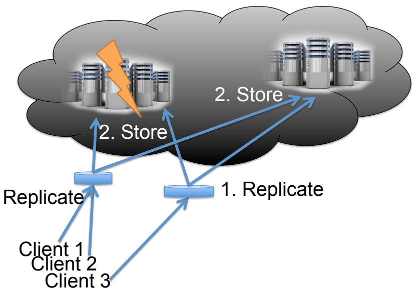
Fig. 1: The New York doughnut represents the locations where financial companies replicate their datacentres

Making services tolerant to disasters can be particularly challenging. Consider a fault-tolerant key-value store, which serves `get` and `put` requests from clients; various possible implementations (*e.g.*, MongoDB, IBM's Spinnaker, Amazon's SimpleDB) are offered by cloud service providers. Despite their simplicity, these applications tolerate isolated failures but not disasters by instantly replicating the effect of an update, say a successful `put` request, to multiple servers in one (not multiple) datacentre. Existing alternatives can mitigate disaster effects with mirroring, replication and logging techniques across datacentres of distant geographical locations [15,16,19]. Typically, these solutions store the client data at one of these replicated datacentres before starting the migration to another datacentre as depicted in Figure 2a. The challenge is then to minimise the migration delay as this may translate into some amount of data that can be lost during a disaster, also known as non-nil *Recovery Point Objective (RPO)* [17]. In this scenario, data are vulnerable to disasters between the time they are stored at the first datacentre and the time they are fully copied or logged at remote places. If a disaster affects the first datacentre, data stored but not migrated become unavailable until recovery or can even be definitely lost.

In this paper, we propose *SDN-KVS*, a consistent SDN-based Key-Value Store that adopts the opposite approach of *replicating before storing.* To this end, we leverage Software Defined Networks (SDN), namely the decoupling of control functions from the data processing and forwarding functions remotely controllable. In particular, we replicate the network flows even before data are actually stored at any datacentre as illustrated in Fig. 2b (this is the multicast feature of SDN-cast). Once the client issues a request to a particular datacentre, an SDN-enabled switch located at the edge of the network (outside any datacentre), duplicates the critical request flow and forwards a copy to two identical versions of the Key-Value Store service running at the targeted datacentre and at the backup datacentre. By duplicating the traffic at the network level, the storage application guarantees that the data is already replicated before it is stored. This is in contrast with previous solutions that require to first receive data before the mirroring, the replication or the remote logging of data can start.



(a) Existing fault-tolerant solutions store data before replicating them through mirroring or logging: the non yet replicated data get lost upon disaster



(b) SDN-cast replicates data before storing them: the network duplicates the critical traffic to remote datacentres

Fig. 2: SDN-cast proactively ensures data persistence as opposed to approaches that use a best effort recovery of data after disasters

Another consequence of disasters is the network outage that prevents remote clients from accessing the running backup service during a period of time, referred to as the *Recovery Time Objective (RTO)* [17]. More specifically, if a backup server starts rapidly operating using a different IP address, it may not be instantaneously accessible as refreshing DNS caches at the edge of the network could take hours or even days. The main problem is that the network itself takes a long time to recover from a disaster, hence delaying the application recovery. Our solution detects effectively outage at the network level to minimize RTO. The key to rapidly redirect the traffic to a backup datacentre is to distribute the network control that is usually centralized in the network core—typically in the datacentre network—to the network edge (this redirection is handled by the anycast feature of SDN-cast).

We evaluate SDN-KVS on top of an emulated wide area network connecting a client to two geo-replicated datacentres in Australia and Ireland, each running a copy of our key-value store application. Although not guaranteed to share the same state, our key-value store instances are strongly consistent and tolerate isolated failures by exploiting intra-datacentre replication but relies on our SDN-cast solution to cope with disasters. In this evaluation we demonstrate how SDN-multicast duplicates the flows between multiple datacentres while SDN-anycast detects edge failures in Sydney to redirect the traffic to the backup datacentre in Dublin. Results show that SDN-cast can achieve a nil RPO and a 30 s RTO, meeting higher disaster recovery objectives than any technique we are aware of.

The remainder of the paper is organised as follows. In Section 2, we evaluate the effects of disasters on data and explain how existing solutions aim at mitigating these effects. In Section 3 we present SDN-KVS. In Section 4 we show empirically that it ensures 30 s RTO and nil RPO. In Section 5, we present the related work. Finally, Section 6 discusses our solution and concludes.

2 On the Impact of Disasters on the Cloud

In this section, we present the problem of making cloud storage services disaster-tolerant. We first present the impact of disasters on cloud storage services before explaining why existing approaches may suffer from disasters.

2.1 The Case of Amazon Datacentres

Even common natural disasters can have important consequences on cloud computing services. On June 14th 2012, Amazon's datacentre in West Virginia (also known as its US-east-1 region) experienced a power outage of only half an hour. While the cause may seem negligible (severe storms), the consequences of this outage were dramatic for all the companies that relied on the Amazon Web Services running in West Virginia datacentre at that time. More precisely, the outage affected companies, like Pinterest and Instagram during up to 15 hours[5], because the power outage induced a cascade of problems affecting the whole service infrastructure of these companies. Larger disasters, as the extreme heat that led to the 2012 India blackout whose power outage affected 9% of the world population, could potentially have more important consequences.

In fact, datacentre service outages represent a key challenge of cloud computing. A recent survey indicates that the cumulative datacentre outage in the US in 2010 was 134 minutes, translating into a cost of \$680,000. Three years later the datacentre service downtime reduced to 119 minutes, however, the related cost increased to \$901,500. This cost increase reflects that more critical services are outsourced to the cloud environment making the financial loss more important in case of failures.[6]

Read-only cloud services, like web services, are easy to make tolerant to disasters. As long as the service does not store client-generated content, the service can simply be copied across geo-replicated datacentres to achieve disaster tolerance. In the Amazon US-east-1 region outage scenario, Netflix, which offers video-on-demand services, minimized service outage by simply redirecting the

[5] `https://gigaom.com/2012/06/29/some-of-amazon-web-services-are-down-again/`.

[6] `http://www.datacentreknowledge.com/archives/2013/12/03/study-cost-data-center-downtime-rising/`

traffic towards a secondary Amazon datacentre.[7] The problem is more complex for services that store client-generated content: their customer may request updates at any time but require their requests to be safely stored in real-time. One popular example of such storage service is the key-value store service.

2.2 Recovering a Storage Service after a Disaster

Key-value stores are cloud services popularized by the NoSQL movement that favoured simplicity over expressivity of data access. They offer a simple interface to manipulate key-value pairs, which exports `get`, `put` and `update` functions that respectively retrieve, insert and modify a pair of key and value. A key-value store achieves fault-tolerance by making sure that the effects of an update request (`put` and `update`) on a specific key-value pair get replicated at multiple servers.

While appealing, the replication needed for fault-tolerance also raises problems related to the consistency of data, as communication must occur between servers to ensure that the new value updated by a client is propagated to multiple servers. There are various forms of consistency provided by key-value stores ranging essentially from eventual consistency to strong consistency. To ensure the uniqueness of the value of a data, two concurrent updates of the same key, say `put(k,v)` and `put(k,v')` requests, should be consistently ordered by all servers. This can be achieved using timestamps computed based on the unique IP address of some server and a local counter that together "tag" the version of a value, indicating for example that `v'` is more up-to-date than `v`. This is similar to the technique used by multi-writer atomic register implementations in the message passing model [12].

Another consistency requirement is that a second update, issued after a first one on the same key has completed, should always have a value tagged with a later version. This requires that each request, whether it be a read-only, like a `get`, or an update, like a `put`, starts by requesting the current highest version to a *quorum* (*i.e.*, a mutually intersecting set) of servers before proceeding with propagating the most up-to-date value with the largest version. One example, employed by Dynamo [4] is to propagate any write request on a key to a quorum of two replicas of this key to guarantee fault tolerance of the data propagated. Using a quorum system, the read/write requests are strongly consistent. Note that a quorum system within a datacentre can be reconfigured to tolerate an unbounded number of isolated failures [3] but cannot tolerate disaster.

3 SDN-KVS: Disaster-Tolerant Key-Value Store

To tolerate failures and disasters our approach is twofold. As for fault-tolerance, we replicate all data within a datacentre where communication cost is low. This guarantees that the data persist despite isolated failures. We also replicate critical traffic (*e.g.*, `put` requests) to a datacentre towards a second datacentre located in a different region, similar to the backup datacentre in New Jersey. It

[7] `http://www.nytimes.com/2011/04/23/technology/23cloud.html`

is the responsibility of the client to distinguish normal from critical traffic (*e.g.*, by establishing separate connections), as cross-regions critical traffic experiences significant delays compared to non-critical traffic within a local region.

3.1 Correctness and Resilience Across Regions

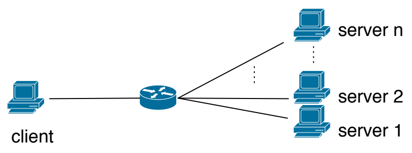
Our key-value store guarantees the strongest form of consistency we know of, called linearizability [7]. Within the same datacentre this is ensured using quorum systems and global timestamp as previously discussed: fetching a key-value pair (resp. the highest version of the storage) requires to contact a set of servers that includes at least one of the servers where the last update of the corresponding pair (resp. the highest version of the storage) was replicated.

To globally guarantee strong consistency, we need additional requirements. Provided that remote datacentres share a common initial state, say as given by some common virtual machine image, replicating critical traffic across datacentres guarantees that critical data is not lost upon geo-localized disaster, even if an entire datacentre goes down. It is also important to guarantee that the key-value store services respond identically to write requests despite possible reordering of requests at distant locations. Our implementation actually acknowledges identically a put and an update even though the corresponding request does not succeed in updating the store. This ensures that when two distant servers receive two update requests in different order, the corresponding response is identical (simply acknowledged). Note that these requests necessarily come from distinct clients as our new SDN-cast primitive uses exclusively TCP as we describe below.

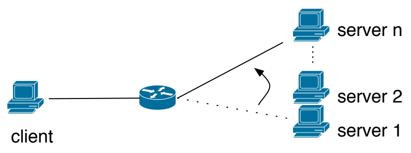
While linearizable, both datacentres may have distinct states because of distinct ordering of concurrent updates. As each client directly connects to the closest datacenter before or after a disaster, the only problem arises when a disaster occurs: the same client may read twice the same data item and observe a different results at the first datacentre right before the disaster and at the backup datacentre right after the disaster. In this case, we require that the client synchronises with the newly contacted datacentre before issuing requests (this is made possible as the application receives a RST packet in case of disaster as we describe later).

The two components of SDN-cast, namely SDN-multicast and SDN-anycast, are depicted in Figure 3. When the client sends a critical (`put`) request to a datacentre, SDN-multicast forks the TCP connection to multiple datacentres, thereby replicating the information before storing it (see Section 3.2). This is transparent to the client: when it receives an acknowledgement, it has already been safely replicated at different geographical locations. This guarantees that the sent data will persist despite a disaster. SDN-multicast hence achieves a nil RPO for all successful critical requests.

Upon detection of a network disaster, SDN-anycast redirects the traffic to backup datacentres, thereby guaranteeing that the data can still be accessed. This is used both for access to the critical data, which was geo-replicated by SDN-multicast, and to redirect all traffic to the surviving datacentres. As opposed to



(a) SDN-multicast transparently forks TCP flows to n servers in separate datacentres, allowing for data replication in real time



(b) SDN-anycast redirects the traffic from one server in the primary datacentre to the server of a remote available datacentre upon network outage detection

Fig. 3: SDN-cast (a) duplicates critical traffic to two datacentres before disasters and (b) redirects the traffic to the backup datacentre upon disaster

other approaches, SDN-anycast uses network-level failure detection to achieve a RTO of about 30 seconds (see Section 3.3).

The control plane is located at the edge to cope with region-wide disasters. (Note that the case of a disaster occurring at the network edge is of limited interest as the client would be affected by the disaster even if the datacentres were not.)

3.2 SDN-Based Multicast for a Nil RPO

The forking mechanism can be placed anywhere on the path between clients and servers. When traffic destined to any server under its jurisdiction is received, it replicates the packets to the whole set of live servers. From the client's perspective, the forking mechanism maintains the end-to-end reliability semantics of TCP: data is acknowledged only when all live servers have acknowledged it. This mechanism is illustrated in Fig. 4 in the case of two servers.

In order not to break the TCP session on any side (client or servers) of the connection, special care must be taken when replicating the packets. The client sends data from its own sequence space, and so does each server. Sequence and acknowledgement numbers therefore need to be adjusted depending on the server before packets are sent. To do so the TCP forking mechanism records the initial sequence number on the first (SYN) return packet it sees ($dc1\_seq$ in Fig. 4). It then computes and stores an offset from the initial sequence number of each of the other servers ($Offset_2 = dc2\_seq - dc1\_seq$ in the figure). This offset is added to the sequence number of return packets, and subtracted from the acknowledgement number on replicated forward packets. This allows to map the client's view of the server's sequence space to the actual range used by each server.

Once the connection is properly open at the client and servers, the former can send any data, such as `put` instructions for the key–value store. The switch transparently duplicates the TCP stream to all live servers, and an acknowledgement is sent back to the client only once successfully received by all servers. This

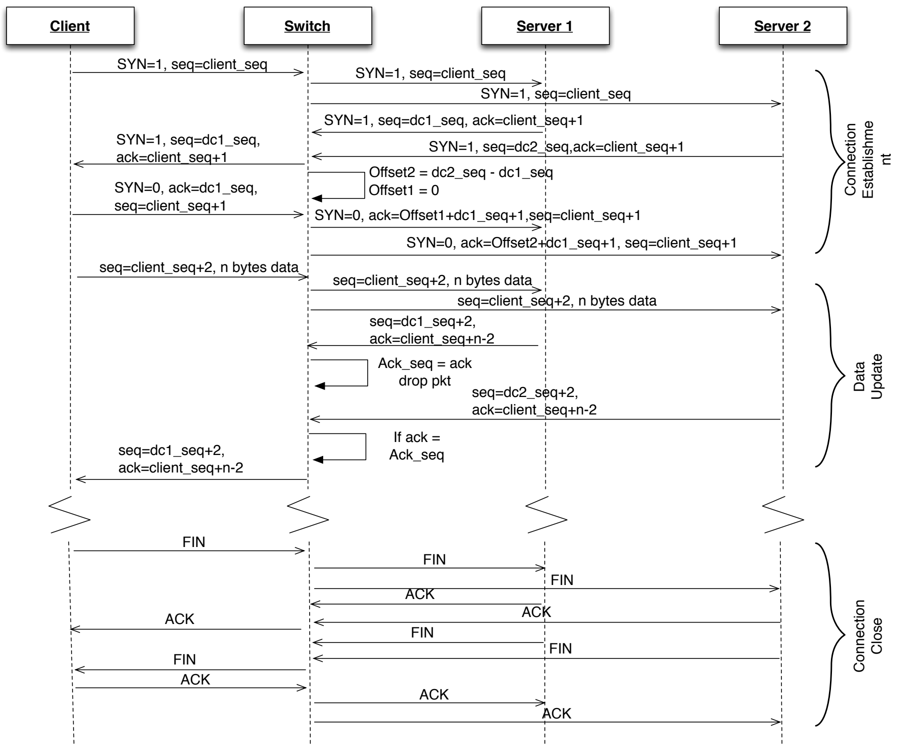

Client
Switch
Server 1
Server 2
SYN=1, seq=client_seq
SYN=1, seq=client_seq
SYN=1, seq=client_seq
SYN=1, seq=dc1_seq, ack=client_seq+1
SYN=1, seq=dc2_seq,ack=client_seq+1
SYN=1, seq=dc1_seq, ack=client_seq+1
Offset2 = dc2_seq - dc1_seq
Offset1 = 0
SYN=0, ack=dc1_seq, seq=client_seq+1
SYN=0, ack=Offset1+dc1_seq+1,seq=client_seq+1
SYN=0, ack=Offset2+dc1_seq+1, seq=client_seq+1
Connection Establishment
seq=client_seq+2, n bytes data
seq=client_seq+2, n bytes data
seq=client_seq+2, n bytes data
seq=dc1_seq+2, ack=client_seq+n-2
Ack_seq = ack drop pkt
seq=dc2_seq+2, ack=client_seq+n-2
If ack = Ack_seq
seq=dc1_seq+2, ack=client_seq+n-2
Data Update
FIN
ACK
Connection Close

Fig. 4: Mechanism for forking a TCP connection: the client establishes a normal connection to Server 1; the Switch replicates the traffic to Server 2; return traffic from the servers is only forwarded to the client when both servers have replied; sequence numbers and acknowledgements are adjusted by recording and applying offsets between servers's sequence numbers as needed

assumes that all servers to which the traffic is replicated reply with exactly the same message. This is not unreasonable, as all servers are identical, varying only in their location, and expected to handle the data they receive in the same way. The connection can finally be closed in a similar fashion, with FINs and ACKs being transparently replicated (and their sequence/ack numbers adjusted), until the slowest server closes its side, letting the client finalise its.

It is important to note that only a few fields are manipulated in the switch. Sequence, or acknowledgement numbers—depending on the direction of the traffic—have a static offset applied. SACK options are left untouched, but forcefully disabled during the initial handshake to force a fallback to cumulative ACKs. Beyond these changes, the TCP packet is left untouched. This allows to leverage the TCP stacks of the endpoints (client and servers) to take care of rate adaptation and loss recovery. As the end-to-end semantics and control of TCP is preserved, the client and servers will however exchange packets as allowed by the slowest link.

3.3 SDN-Based Anycast for Minimum RTO

SDN-anycast uses SDN-enabled switches connected to controllers located at the edge of the network to redirect flows upon network outage. In order to decide when to start the redirection, we propose a mechanism inside the edge controller. As SDN-anycast can detect network outage using lower level protocols (*e.g.*, LLDP, ICMP or transport timeouts, as used in Algorithm 1), it can responsively redirect the traffic with minimum delays. This detection algorithm takes information from the various edge switches while the decision is centrally taken by the SDN controller responsible for these switches. This centralised algorithm, presented in Algorithm 1, aims to minimise the RTO depending on the number of services currently deployed.

```
Data: Client flow to service
Result: Possible detection of failure
detection;
if duplicate packet from client then
    possibleFailure ++ ;
    if possibleFailure $\geq$ Threshold then
        add action to rewrite packets for Service to alternative server ;
        for every connections affected by failure do
            send RST packet ;
        end
    end
else
    if packet from Service then
        possibleFailure = 0;
    end
end
```

Algorithm 1: Detection of server failure through TCP timeouts, and mitigation by forcing a new connection

4 Experimental Evaluation

In this section, we show empirically that SDN-anycast recovers the throughput and latency of the key-value store service within RTO $\leq$ 30 s and that SDN-multicast increases the cumulated goodput by forking TCP connections between a single client and multiple geo-replicated datacentres to achieve nil RPO with no client or inter-datacentre overhead. We used Mininet [6] and the Python-based POX controller to evaluate the performance of our geo-replicated key-value store in the face of disaster.

4.1 Recovering Storage Service with SDN-Anycast

To evaluate the Recovery Time Objective (RTO) SDN-cast achieves, we deployed our solution over a topology comprising a local datacentre in Sydney, Australia, and a remote one of in Dublin, Ireland (see Fig. 5). These are the current respective locations of Amazon's existing AP-southeast-2 and EU-west-1 regions. Note that this represents one of the greatest distance between datacentres worldwide (~17,200 km) whereas most backup datacentres are located within only 30–70 km from the primary datacentre, as we illustrated in Fig. 1. Both datacentres run the same image of the service in the same initial empty state replicated on a distributed system of 9 machines with quorums of size 5.

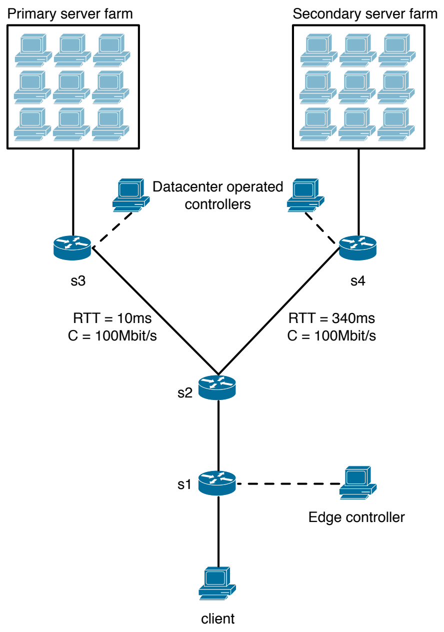
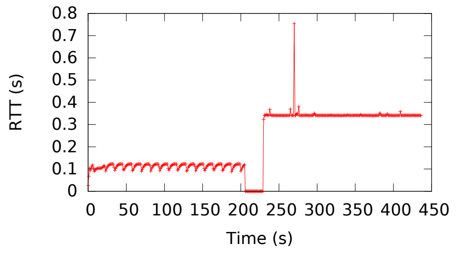
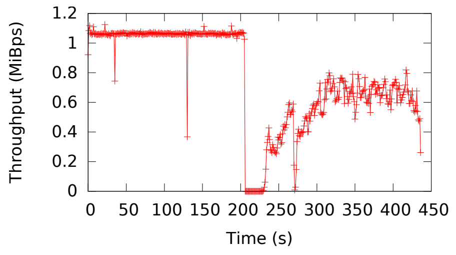


Fig. 5: Experimental topology deployed in Mininet to represent a client accessing a local datacentre in Sydney (left) and a remote datacentre in Dublin, Ireland (right)

The link from switches s_2 to s_3 has a one-way delay of 5 ms and a capacity of 100 Mbit/s whereas the link from s_2 to s_4 has a delay of 170 ms and a capacity of 100 Mbit/s (the RTT between the Sydney and Ireland Amazon EC2 datacentres is about 340 ms). Inside each datacentre, we have configured a key–value store over a quorum system of nine servers, meaning that a server receiving `put`, `update` or `get` requests exchanges messages with 8 or 9 other replicas before acknowledging the client of the success of its request, hence guaranteeing a high



(a) RTT



(b) Throughput

Fig. 6: SDN-anycast allows to achieve fast service recovery (*i.e.*, RTO $\leq 30\,s$) and no data loss (*i.e.*, nil RPO) across the globe

level of intra-datacentre fault tolerance. Both switches s_3 and s_4 act as load balancers inside each datacentre.

Our experiment scenario is as follows. At time $t = 0$, one of the client starts reading from the local datacentre as fast as possible, filling the pipe between s_2 and s_3. At time $t = 200$, we emulate a disaster by cutting the link between s_2 and s_3. Once the connection is cut, the TCP client starts retransmitting unacknowledged packets. The controller can therefore learn that a possible disaster has occurred in the network. As we fixed the detection threshold described in Algorithm 1 to 5 packets, the waiting period varies from 15 to 25 seconds. After this inactive period, the controller instructs the switch to redirect traffic to the secondary datacentre, and to terminate the connections with the now-defunct primary datacentre. The client then reconnects to the service with the same IP address but traffic is directed to the second datacentre.

In Fig. 6, we present RTT and throughput measurements extracted from the captured trace at the client side. Fig. 6a shows the RTT computed by the TCP sender. In particular, we can see that during the first 200 s the flow experienced an RTT of around 100 ms. During the second 200 s period, the flow experiences a minimum RTT of 340 ms necessary to reach the remote datacentre. Fig. 6b presents the throughput achieved by the TCP flow. As expected, when the delay is short, the TCP connection is able to fill the pipe when accessing the local datacentre. With a larger RTT it however becomes impossible for TCP to fully utilise the pipe. Nonetheless, the connection was successfully switched over to the Irish datacentre when the Sydney one failed. Using this technique, we empirically observed an RTO lower than 30 s.

4.2 Service Goodput of SDN-Multicast

The benefit of SDN-multicast in our scenario is to avoid the need of application-level replication, as it directly integrates the same service within the network. This reduces the number of communication between members of a group.

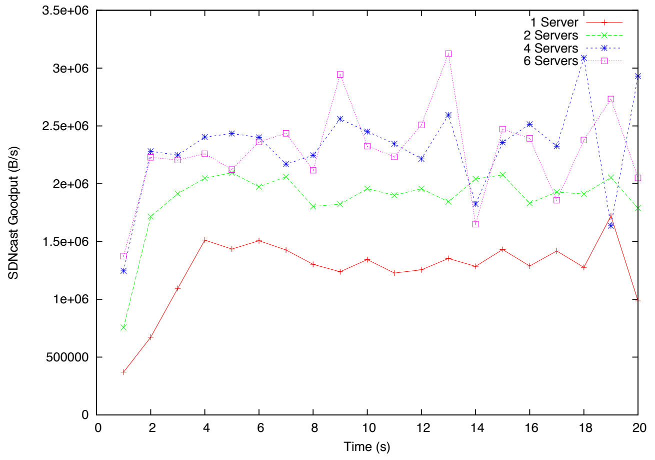


Fig. 7: Performance evaluation of SDN-multicast to up to 6 datacenters concurrently

More specifically, a client process sends a message to a group of server processes by initiating a single TCP connection through which it sends multiple messages. All messages are reliably delivered in-order to all servers. The TCP acknowledgements are simply collected at the switch level. Once all servers have acknowledged a specific sequence, the switch forwards the acknowledgement to the client.

In order to illustrate this use-case, we deployed our forking solution over a simple topology similar to the one presented in Fig. 3b where we varied the number of servers from 1 to 6. In this series of experiments we set identical RTTs between the set of servers and the client, and links capacity to 100 Mbit/s.

Fig. 7 presents the service goodput, defined as the number of bytes transmitted and acknowledged by all servers, as a function of time. As OpenFlow ≤ 1.3 does not currently offer actions to rewrite TCP sequence or acknowledgement fields, we had to let the switches forward all packets to the controller, hence creating a significant overhead and loss of performances: ideally, we would have expected a linear relation between the number of servers and the gain in goodput. Nevertheless, we can observe that our solution increases the service goodput. We envision this increase to become even more significant when OpenFlow switches support the necessary actions.

5 Related Work

Synchronous replication. Data persistence in case of disasters was explored in the context of databases [5] and gained momentum recently with the geo-centralization of data in cloud datacentres. While asynchronous replication [10, 16] was shown in practice to be effective to limit bandwidth consumption, synchronous replication is needed to prevent data losses [15,19]. Synchronous repli-

Table 1: Feature comparison of SDN-cast with related work

	NAT	TCP Splice [13]	Flow Aggregation [2]	MPTCP [1] SCTP	Any-cast	Dr. Multi-cast [18]	**SDN-Cast**
Multicast	×	×	×	×	×	✓	✓
Recovery	×	×	×	✓	✓	×	✓
Transparency	✓	×	×	×	✓	×	✓

cation requires first to store the data then to replicate the data somewhere else before acknowledging the client, a lengthy process generally suited for relatively short distances. Our approach is the opposite: first to replicate the traffic to store data at multiple locations concurrently before acknowledging the client.

Manipulation of network flows. Many solutions have been proposed, over the last fifteen years, to extend network functionalities. They generally vary in terms of the layer at which they are implemented and the location (*e.g.*, end-host, edge or core) of their deployment. Table 1 compares the features of various proposals, presented below, to SDN-cast, in terms of multicast capabilities, geo-localized disaster recovery and transparency to the client.

Similar to our proposal, solutions based on middle-boxes have been widely adopted in today's Internet. In particular, Network Address Translation (NAT) offers transparent static redirection services to end-user applications, and are generally deployed at the edge of the networks. TCPSplice [13] is a kernel-land TCP proxy allowing to intercept and redirect TCP sessions. Unlike application proxies, it offers near router speed performance. To mitigate the problems introduced by wireless communication, and use the links more efficiently, a flow aggregator inside a TCP proxy was proposed to allow to adapt the protocol to better use GSM links [2]. thoroughly studied [8, 14]. Although they provide many advantages, it is well-known that they introduce limitations for the evolution of end-to-end protocol such as TCP [8]. Nonetheless, as compared to our in-network layer-4 switch proposal, none of these solutions support both dynamic redirection of traffic and flow replication to multiple destinations.

Transport protocols supporting multiple paths such as MPTCP [1] or SCTP allows the use multiple paths within the same transport session. This is instrumental in supporting network fail-over by switching from one interface to another without connection disruption. However, switching conditions are hard-coded in the transport, and cannot be adjusted depending on arbitrary parameters. Unlike SDN-multicast, however, each end-to-end sessions can only be established to a single host, and cannot be failed-over to different, or multicasted to several, servers. Moreover, both protocols require support at both end-hosts to be used.

Dr. Multicast [18] provides multicast functionalities by coping with the limited scalability of IP multicast within one datacentre, however, it is not transparent as it requires the client to catch system calls and converts IP multicast addresses.

Software defined network (SDN). SDN offers inherent fault tolerance capabilities by allowing a controller to use lower layer signals such as LLDP to detect link failures in a timely fashion, and to adjust the behaviour of the switches it controls. This technique has therefore been suggested to reconfigure networks upon failures, especially in the context of datacentre networking [9]. SDN was recently used to mitigate the effects of disasters [20], similarly to our SDN-anycast feature. The goal was however to use alternative network paths in case of path failures, and this solution does not cope with the problem of data loss that SDN-multicast addresses: it only mitigates the effects of disasters rather than avoiding them. The use of multiple controllers was already shown effective in reducing the fault-tolerance of a particular SDN using NOX controllers and the Mininet virtualised environment [11]. In SDN-multicast, we leverage these capabilities to implement layer-4 switching mechanisms to improve RTOs and RPOs.

6 Discussion and Conclusion

We proposed SDN-KVS, a disaster-tolerant storage that exploits a novel SDN-cast communication paradigm to avoid losses of critical data. Even at extreme distances (Australia to Ireland), SDN-KVS achieves a 30 s RTO and guarantees that storage requests survive region-wide disasters.

As our solution is deployed at the end of the common path towards all servers (*e.g.*, leaf network edge or local ISP), the forking mechanism saves resources along that path by removing the need for multiple connection carrying the same traffic. We envision that as a side effect, it could potentially reduce bufferbloat issues by only using one TCP stream instead of one per server for the same data.

Our solution allows multicasting of TCP streams transparently to the client side. Due to its layer 4 orientation, it ignores any payload. While this is desirable for performance reasons in most cases, this is problematic if some application parameters need to be negotiated between client and server. This is particularly the case for SSL sessions, as each servers would try to negotiate a different session with the client. To provide similar levels of security, IPSec would be a better candidate for use with our forking mechanism.

The current version of OpenFlow (v.1.4) limits the capabilities of SDN-multicast. In particular, we were limited, in the forking experiment by the lack of support for manipulation of the acknowledgement and sequence number fields in the TCP header. It is also unclear whether arithmetic operations such as additions or subtractions of offsets are readily available. We believe that such actions should be considered for addition in future OpenFlow specifications.

For our solution to be widely-adopted, the SDN-anycast information should be made accessible to ISPs to reconfigure efficiently the network. By proactively redirecting traffic to live servers upon disaster, our SDN-anycast could simplify the task of edge ISP forensic departments as it would prevent a large number of connection establishments failures due to downstream disasters that would have triggered DDoS investigation procedures.

References

1. Sébastien Barré, Olivier Bonaventure, Costin Raiciu, and Mark Handley. Experimenting with multipath TCP. In *SIGCOMM*, pages 443–444, 2010.
2. Rajiv Chakravorty, Sachin Katti, Jon Crowcroft, and Ian Pratt. Flow aggregation for enhanced TCP over wide-area wireless. In *INFOCOM*, 2003.
3. Gregory Chockler, Seth Gilbert, Vincent Gramoli, Peter M. Musial, and Alexander A. Shvartsman. Reconfigurable distributed storage for dynamic networks. *J. of Parallel and Distributed Computing*, 69(1):100–116, jan 2009.
4. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.
5. Hector Garcia-Molina, Christos A. Polyzois, and Robert B. Hagmann. Two epoch algorithms for disaster recovery. In *VLDB*, pages 222–230, 1990.
6. Nikhil Handigol, Brandon Heller, Vimal Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *CoNEXT*, pages 253–264, 2012.
7. Maurice Herlihy and Jeannette Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.
8. Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. Is it still possible to extend TCP? In *IMC*, 2011.
9. Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, pages 3–14, 2013.
10. Minwen Ji, Alistair C. Veitch, and John Wilkes. Seneca: remote mirroring done write. In *ATC*, pages 253–268, 2003.
11. Hyojoon Kim, J.R. Santos, Y. Turner, M. Schlansker, J. Tourrilhes, and N. Feamster. CORONET: Fault tolerance for Software Defined Networks. In *ICNP*, 2012.
12. Nancy Lynch and Alexander Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *FTCS*, pages 272–281, 1997.
13. David Maltz and Pravin Bhagwat. TCP splicing for application layer proxy performance. RC 21139, IBM, Mar. 1998.
14. Alberto Medina, Mark Allman, and Sally Floyd. Measuring interaction between transport protocols and middleboxes. In *IMC*, pages 336–341, 2004.
15. Oracle. Oracle optimized solution for disaster recovery on oracle supercluster, 2013.
16. R. Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. SnapMirror: File-system-based asynchronous mirroring for disaster recovery. In *FAST*, pages 117–129, 2002.
17. Akshat Verma, Kaladhar Voruganti, Ramani Routray, and Rohit Jain. SWEEPER: an efficient disaster recovery point identification mechanism. In *FAST*, pages 297–312, 2008.
18. Ymir Vigfusson, Hussam Abu-Libdeh, Mahesh Balakrishnan, Ken Birman, Robert Burgess, Gregory Chockler, Haoyuan Li, and Yoav Tock. Dr. multicast: Rx for data center communication scalability. In *EuroSys*, pages 349–362, 2010.
19. Timothy Wood, H. Andrés Lagar-Cavilla, K. K. Ramakrishnan, Prashant Shenoy, and Jacobus Van der Merwe. PipeCloud: Using causality to overcome speed-of-light delays in cloud-based disaster recovery. In *SoCC*, pages 17:1–17:13, 2011.
20. An Xie, Xiaoliang Wang, Wei Wang, and Sanglu Lu. Designing a disaster-resilient network with software defined networking. In *IWQoS*, pages 135–140, May 2014.