COSCAnet-FT: transparent network support for highly available cloud services

Steffen Kächele
Institute of Distributed Systems
Ulm University, Germany
Email: steffen.kaechele@uni-ulm.de

Franz J. Hauck
Institute of Distributed Systems
Ulm University, Germany
Email: franz.hauck@uni-ulm.de

***Abstract*—More and more applications move to the cloud, even critical systems that need high availability. In current clouds, faults can be handled for stateless HTTP applications. Other protocols and stateful applications cannot be supported. Highly-available stateful services could use active replication, but it typically needs client-side code for supporting complex totally-ordered multicasts. This paper first presents a transport-level router as a service in the network that transparently multicasts TCP traffic to actively replicated service instances. Second, it demonstrates the integration of this concept into a PaaS cloud as a value-added service to customers so that applications can be replicated on demand. Finally, an evaluation of our prototype shows reasonable throughput, latency and recovery time.**

I. Introduction

Today's cloud platforms provide scaling features that involve many physical nodes which makes the advent of a failure in one of them very likely. Although a lot of research has been done, current cloud platforms still lack comprehensive and transparent support for fault tolerance. Recent cloud outages, however, underpin that there is a need for fault-tolerance mechanisms in the cloud. For high availability, current clouds have support for stateless services that store state in a shared backend storage. The service itself can be deployed with multiple instances on different nodes. A load balancer dispatches incoming requests among them. In case an instance fails the load balancer can focus on the remaining instances and the service remains available. A client, however, may experience a failed connection, but can proceed after a retry [11].

If faults should be entirely transparent to clients, or applications are allowed to maintain state, state-of-the-art replication mechanisms can be used. With *passive* or *primary-backup replication* a client interacts with a primary instance, whereas additional instances stand by and get more or less state updates. This is not appropriate for highly available services because it needs a time-consuming fail-over to another primary in case of faults. With *active replication* clients communicate to all replicas at the same time [12]. For a stateful service, it is modelled as a set of state machines [15]. Each state machine gets the same input and executes the same deterministic state transitions. Thus all replicated state machines come to the same conclusions and answers to client requests. If one of the replicas fails others are immediately available to finish current and future client requests. Unfortunately, most replication mechanisms need client-side code that cooperates in recovering from faults, addressing the primary or implementing a totally ordered multicasts to all replicas. Thus, standard client software, e.g., mail clients and Web browsers, cannot be used. Besides, there is currently no support for this kind of replication in standard cloud systems.

978-1-4799-5804-7/15/$31.00

The first contribution of this paper is a concept that implements a transparent and totally ordered group communication between a client and an actively replicated service, by multicasting and aggregating an ordinary TCP connection. Client software and communication stacks do not need to be changed. Central is a network component, a *transport-level router* or router for short. At each replica all data arrives in the same order and same chunks, so that a deterministic run-time environment can support the state-machine replication approach. Such an environment is out of the scope of this paper, but previous work like the *Virtual Nodes Framework*, the deterministic Java suite *Dj* and deterministic scheduling could be used [5]. As this router can fail, special care is taken that its state is kept minimal and can be quickly restored by a backup component in case of failures.

Our second contribution is an implementation for TCP named COSCAnet-FT and integrated into our COSCA PaaS cloud [9], [10]. COSCA offers OSGi-like run-time environments and supports elasticity and migration of bundles offering OSGi services. COSCAnet-FT supports a value-added service as ordinary OSGi applications can be transparently replicated on demand providing high availability. COSCA already provided replication for stateless applications by its scalability and load balancing concepts. With COSCAnet-FT this is extended to actively replicated services that can be transparently accessed by arbitrary TCP-based protocols, e.g. IMAP, SMTP, LDAP, RTSP, SIP and HTTP. Also stateless applications benefit as there are no more connection failures, even in case of a failure of the router. Finally, this paper will present a couple of relevant measurements in order to evaluate our concepts. These show that COSCAnet-FT implies low overhead, good throughput and fast recovery.

The next section presents the related work. Section III discusses how to integrate fault-tolerance mechanisms into a cloud network layer. Before Section V concludes, we evaluate our approach in Section IV.

II. Related Work

To enable state machine replication a lot of group communication protocols have been published that dependably deliver messages to a group of replicas in a total order.

Defago et al. [4] states about 60 protocols each having its pros and cons, each implemented on application layer. Thus a client has to implement the specific protocol required by the service. This ties client and service together and does not allow generic clients, e.g. email clients for IMAP servers. Also an additional session layer [6] on top of TCP requires software on the client. COSCAnet-FT goes one step further by transparently integrating totally ordered group communication into a standard transport level protocol (TCP), an approach that perfectly matches today's cloud setups.

FT-TCP [1], ST-TCP [13] and ER-TCP [16] propose *primary-backup architectures*. For recovery they use a logger, but traffic logging may generate a lot of data. HydraNet-FT [17] provides a replicated primary-backup architecture, where only the primary responds to the client. To ensure atomicity and message ordering, they establish an acknowledgement channel from backups to the primary. Their implementation requires modifications on server side of the TCP/IP protocol and the process management. HydraNet-FT also requires that server applications be aware of replication (i.e. modified) and invoke special system calls. Furthermore, they use message encapsulation that can slow down the network. T2CP-AR [2] is designed for active replication but internaly uses a primary-backup architecture with a significant failover effort. To intercept traffic on the backup node, they use a broadcast mechanism on link layer extended by a traffic recorder installed on the gateway. Both mechanisms lead to an architecture that is not scalable in large setups and the gateway is a single point of failure. Although there have been proposed solutions for fault-tolerant TCP, they all show significant failover behaviour. COSCAnet-FT in contrast proposes fully active replication for significantly faster recovery that does not need any checkpointing, state update or logging.

III. Transport-layer group communication

In this section, we present how we transparently integrate group communication for active replication on transport layer. A transport-layer router multicasts and aggregates TCP connections. Eventually this router can be integrated into a scalable cloud system. We consider replicas that run self-contained services and do not store state on a shared medium. Our failure model is crash-stop. This means that in case of a failure a replica will no longer participate in client and inter-replica communication. Furthermore for the network layer it is no longer relevant what state the replica has; this may even have diverged from the authoritative state of the life replicas. Recovery however can take place on higher level, e.g. in cooperation with a recovery manager on application level.

Before we focus on our approach, we briefly summarise COSCAnet [11], the network and transport layer mechanisms of our PaaS cloud COSCA [9]. In COSCA each application gets an individual virtual IP address. COSCAnet's network- and transport-layer router forwards datagrams addressed to a virtual IP address to the right node. The mapping is maintained by COSCAnet so that applications are independent of the node's IP address, and thus can even migrate. Nodes and the network-layer component cooperate so that outbound datagrams carry the correct sender address. This mechanism is also used to implement scalability and elasticity by allowing multiple nodes to host instances of the same application, each having the same virtual IP address. The COSCAnet router then works as a transport-layer load balancer that correctly routes established TCP connections. Details can be found in [11].

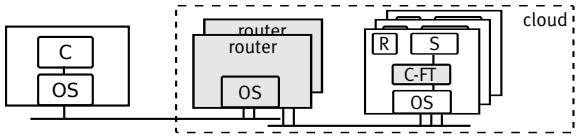

Fig. 1. Client-server interaction in COSCAnet-FT.

A. Architecture

In a non-replicated setup a client initiates a TCP connection to a single server, sends requests and receives responses. State-machine replication and active replication use multiple simultaneously working replicas to tolerate faults. Instead of a typical group communication protocol, COSCAnet-FT implements a transparent multicast TCP connection with multiple replicas. Server-side code is supposed to look like standard server code using TCP sockets. Socket access, however, is intercepted and managed by COSCAnet-FT, whereas the client side is completely unchanged. We currently consider only client-initiated TCP connections as they are used for most server applications. We further assume that behind the server-side sockets there is either an application with a deterministic behaviour or a run-time environment that takes care that in case of no failures all replicas will do exactly the same execution and invoke the same operations with the same data at the sockets in use. This part is outside of the scope of this paper, but is addressed for example in [5].

A totally-ordered multicast based on TCP as described above could be implemented either on client side, server side, in-between (i.e. in the network), or by a combination of these. As client-side should be unaware and server-side only is not dependable enough, COSCAnet-FT is based on mechanisms *inside the network* combined with some server-side help. Like COSCAnet, the COSCAnet-FT architecture introduces a transport- and network-layer router (in the following called *router* for short) that takes care of client-side transparency and the group-communication semantics (cf. Fig. 1).

TCP connections could end in the router and the router could have its own communication with the replicas. This approach is what is used in current TCP-level load balancers, e.g. *HAproxy.* Towards replicas we could even use a special protocol, as with server-side help the application endpoint could still look like an ordinary socket. Terminating and proxying of connections, however, introduces another single point of failure in the network. When the proxy crashes, the previous connection endpoint is no longer available and may sent confusing ICMP or transport layer packets during the crash. Instead, COSCAnet-FT uses standard TCP on client and server side, the router just forwards datagrams. On the replica side we use an almost standard operating system but intercept socket access to cooperate with the router. Beside replicating server instances, the router can also be replicated to achieve high availability. We will discuss this later in Sections III-C and III-D. With COSCAnet-FT, similar to COSCAnet, each replica has the same virtual IP address that serves as identifier for the entire group, making life easier for the router.

B. TCP-based totally-ordered multicast

Totally ordered multicast is typically defined for particular messages sent to a group. Communication has to ensure three properties (rephrased from [7], [3]): (i) If a message m is delivered to one replica, then all correct replicas eventually get m. (ii) For any message m, every replica gets m at most once, and only if m was previously sent by a client. (iii) If replica r_1 and r_2 both get message m and m', then r_1 gets m before m', if and only if r_2 gets m before m'.

Totally-ordered multicast with TCP could be implemented by multicasting each individual segment to the replicas. However, this has serious performance drawbacks, as data can only be delivered to application or sockets respectively when all other correct replicas will deliver this data too. That is why COSCAnet-FT heavily optimises multicasting by exploiting inherent properties of TCP. In principle COSCAnet-FT has to achieve the following four properties: (i) *TCP connection requests* from clients have to be delivered to all correct replicas in the same order, so that determinism at replicas can be achieved. If one correct replica cannot accept the connection none of the correct replicas shall accept it. (ii) *Incoming data chunks* have to be delivered to all correct replicas. As TCP is stream-oriented and reliable all correct replicas will get the same data in the same order. However, the stream data have to be delivered in the same chunks at all replicas, even though the network may deliver the data in different portions (fragmented, combined or scattered). (iii) *Outgoing data chunks* should be consistent even when coming from all replicas. Outgoing data is supposed to be deterministic by assumption. For our failure model it is even possible to immediately forward the first outbound data chunk to the client. This helps to maintain performance. (iv) *Closing the connection* is supposed to be deterministic. All replicas will close the connection at the same connection state. A closing request from the client is multicasted to all replicas in the same way as data.

On a technical level, COSCAnet-FT has to maintain these properties by considering all possible situations on the TCP-layer without compromising determinism and without introducing additional faults. Special care has to be taken for handling TCP ACK and RST messages and the case the client is entirely disconnected. The following sub-sections will address this issue and describe the details of COSCAnet-FT.

C. Multicast routing, inbound data

COSCAnet-FT extends a NAT-based routing, having a mapping for each replica. Network traffic, i.e. IP datagrams, to a particular service is forwarded to all of its replica processes p_i by the router. This packet replication is transparent to transport protocols (e.g. TCP). We first concentrate on data forwarding whereas connection establishment is handled in Section III-E.

In order to preserve determinism when using replication, a group communication has to ensure that input on all replicas is exactly the same. Mapped to TCP streams, all segments have to arrive in the same order, be delivered in the same chunks to the application and must be reliably forwarded to the replicas. TCP already takes care of *message ordering* with the use of sequence numbers which also works in a replicated setup. However, applications on different replicas may receive a different *amount of data* as result of a single `read`-call. This may happen when seg_1 arrives earlier than seg_0 on a replica. Due to ordering, a read call on this replica may return the content of both segments. For deterministic `read`-calls, we return at exactly a single segment at once for a single `read`. Replicas record the size of incoming segments for that purpose.

TCP uses acknowledgements and re-transmissions on transport level for a *reliable data transport*. Thus, best effort transmission on lower layer is sufficient for reliable transmission of streams. COSCAnet-FT basically exploits this property by just multicasting incoming segments to all replicas. With regard to transmission errors, the connection between a client and multiple replicas has two different sections. The link (c, r) directly influences all replicas and thus can behave as a usual best effort link that may drop packets. The links (r, p_i) in contrast have to provide guarantees for packet delivery. Both lost and reordered packets between the router and the replicas potentially lead to non-determinism, because input will be no longer the same for each process (p_i). This directly influences handling of TCP acknowledgement messages of replicas. In principle the router could forward the first ACK, but then has to make sure that all other replicas can certainly get all data without the help of the client. This would require to store segments in the router that are not yet acknowledged by all replicas. In order to keep router state minimal ACKs are collected from replicas and only forwarded when all replicas have acknowledged the same incoming data. This ensures that we can use the TCP re-transmission mechanism to ensure that all replicas get all segments. Naive usage can, however, lead to non-determinism, because TCP can repackage segments on re-transmission. In the unlikely case of a lost segment on the link (r, p_i), replicas can get segments in different length, leading to inconsistency. Our router thus stores the length of segments sent by c until they have been acknowledged by $p_i \forall i$ and can repackages them if there is a discrepancy on re-transmission.

A bigger issue for determinism is a client crash. Even in the case of an abnormal connection termination, COSCAnet-FT has to take care that each replica has read the same amount of data before the connection termination will be forwarded to application level. When not all replicas had received the full amount of data (e.g. in the unlikely case of lost segments on (p, r_i)) and the client crashes, the replicas will remain in different states. Via acknowledgements, the router can determine if all replicas have received the same amount of data. To come over the discrepancy, all replicas will save received segments for a certain time period. The router can recover data from one replica and transfer it to another. Thus, all replicas can be update to the same state before the connection terminates. To keep segment caching to a minimum, the router periodically sends cleanup messages.

Some of the mechanisms described in this section, cause state in the router which will be lost during a crash. This applies for the position in the stream to which replicas have acknowledged segments and the segmentation of the messages stream in the direction of the replicas. We run a daemon on the replicas where a new router instance can collect information about the current state during initialisation.

D. Traffic aggregation, outbound data

For outbound data the link (p_i, c) can be considered as a *many to one* mapping. A strategy to aggregate data is to wait for all

replicas before sending data to the client. A single replica, however, can significantly slow down the whole connection when it has performance issues. Even worse, when a replicas crashes data communication gets stopped until the crash has been detected and the replica has been removed from the replica group. Such a behaviour contradicts our intention to support highly available services. As we use a deterministic execution of replica processes we can forward data as soon as we receive it from one replica. Duplicate data of further replicas, will be detected and filtered by the client stack, but to avoid unnecessary data transfer the router takes care that each data is only transferred once. A side effect of this strategy is that the router has to deal with replicas that are in a previous state. In particular this applies for acknowledgements sent by the client. The router has to ensure that it forwards acknowledgements only to those replica that already sent the respective data. The router therefore saves the stream position of any replica and modifies acknowledgement numbers to match the current state of the replicas. As the information for the position in the message stream will get lost due to a crash on the router, the router can fetch the current stream position ($seq(p_i)$) after a reboot from the respective replica instance.

During a client crash its stack may either send FIN (application terminated), RST (unexpected reboot) or no message at all (system freeze). For client consistency it is important that all replicas have sent and received the same data before the connection terminates. The router takes care that all replicas will receive the stream until the same position, and broadcasts the position up to which each replica has to write to before it can signal a connection abort to its application.

To implement congestion control, TCP uses a sliding window mechanism. In a replicated setup advertised windows of clients can be simply replicated to the replicas, but the windows of replicas have to be aggregated. The router saves the window size last seen of each replica. When an acknowledgement is sent to the client, the window is set to the smallest value collected from the replicas. Similar to the information belonging to the position in the message stream, window size ($rcvwin(p_i)$) of the replicas are affected during a crash and can be recovered from the replicas by the new router instance.

TCP uses *timestamps* for congestion control and for protection against sequence number wrap around [8]. On each TCP segment, an endpoint adds its local timestamp and replies the timestamp last received from the remote side. TCP timestamps are required to be monotonically increasing, but in a scenario with multiple replicas, the mix of timestamps from different replicas may highly confuse the client. As timestamp synchronisation across replicas is very complex, we unify the timestamps on the router. Before sending segments to a client, we replace the original timestamp with a timestamp of the router giving the client a basis independent of the replicas. As the client will then use the timestamp of the router in its echo reply, we also adapt echo reply timestamps before forwarding them to the replicas. During router recovery of a crashed router, a new instance has to take care that forwarded segments include monotonically increasing timestamps. Otherwise, the client will drop them. The router can extract timestamps of a connection of incoming segment. If the router has to forward segments before it gets incoming segments, it can temporarily remove timestamps until it learns them from the client.

Due to consolidation introduced in this section, the router can use a *passive failure detection*. The router can remove non-responding replicas. Also FIN and RST segments from only one replica are an evidence for a crash.

E. Connection establishment and modification of FT groups

Security features of TCP such as random starting sequence numbers hinder a straightforward establishment of a TCP connection with multiple replicas. During connection setup, our router intercepts SYN segments being sent from the client and assigns additional information to the segments such as the initial sequence number. The replicas use this information to alter the TCP stack after connection setup. Before forwarding the SYN-ACK to the client, the router waits for all replicas sending a SYN-ACK. The following connection flow can be proceeded as usual. When a replica does not respond, the client will thus repeat the SYN which will be patched by the router with the same sequence number. After n re-transmissions without an answer the router will treat a replica as crashed and remove it from the replica group. During connection setup, the router has state, namely the sequence number determined for an established connection, but the router can recover this information from the replicas. An additional socket replication mechanisms allows adding replicas to an existing connection.

IV. Implementation and Evaluation

In our evaluation, we use four commodity Pentium E5300 machines ($m_1 ... m_4$) with 4 GB RAM all connected via a gigabit network. The replias m_2 to m_4 run the server application. m_1 runs as router with two network interface cards connecting the internal network with the outside. Similar to COSCAnet [11] it uses virtual addresses to address applications. A *netfilter* module implements our extended NAT routing technique. For replacement of timestamps and acknowledgement numbers, the module includes optimized routines for differential TCP checksum calculations. For a deterministic data delivery to server applications, a netfilter module on the replicas records the size of incoming TCP segments. A middleware intercepts `read`-calls from user space and returns exactly the amount of one IP packet on a single `read`. An additional kernel modul supports the manual set up of TCP connections. We use four different setups. *Native* represents a standard setup where the client directly connects to a single replica via a gigabit Ethernet switch. The *NAT/n* setups use our replicated server architecture with n replicas. *NAT/1* does not add fault tolerance. We use it to measures the overhead of introducing a NAT router. *NAT/2* and *NAT/3* measure the overhead of replication using two and three server replicas. We call these configurations *LAN* setups. We also use *WAN* setups where the client connects via the Internet. In these setups the client is connected via a DOCSIS 3 connection with 32 Mbps downlink and 1 Mbps uplink. Round trip time between client and the NAT router is about 10 ms.

In our **latency** experiment we use the *lat_tcp* test of the lmbench suite 3.0a [14]. We run its echo server on our replicas. After a warmup, we measure the round trip time of an 1 byte payload data using TCP. A single lat_tcp run uses 700 echo replies to determine the average round trip time. We use multiple iterations and use the average of 50 runs of lat_tcp (cf. Table I). The test does neither include connection establishment nor tear down. In the *native* setup, we measure

TABLE I. LATENCY FOR IN REPLICATED CONNECTION

experiment	setup	latency	overhead
latency (LAN)	native	0.0999 ms ± 0.0473	–
	NAT/1	0.1996 ms ± 0.0406	99.8%
	NAT/2	0.1997 ms ± 0.0806	99.9%
	NAT/3	0.2000 ms ± 0.0698	100.1%
latency (WAN)	native	10.428 ms ± 356.301	–
	NAT/1	10.532 ms ± 180.664	1.0%
	NAT/2	10.556 ms ± 306.161	1.2%
	NAT/3	10.638 ms ± 284.742	2.0%

a latency of 0.1 ms. We explored that just adding an additional switch significantly increases latency. The use of a NAT router (i.e. NAT/1) doubles latency (0.2 ms) due to an additional hop. Adding further replicas to the architecture only adds a marginal overhead (100-200 ns) since replicas can process requests in parallel. We observe that the overhead of adding a NAT infrastructure (99.8 %) in a LAN setup is quite high. In cloud computing settings, however, WAN setups are more likely, where we measure an overhead between 1 and 2%. Replication does not add significant overhead in those setups.

In our **throughput** experiment, we download a 10 MB file from a web server and measure the time the download takes including connection establishment and tear down (cf. Table II). In the fail free period, the native setup can saturate a 1 Gbit/s link (100 %). We calculate further experiments relative to it. In the NAT setup, throughput decreases to 97.5 %. Adding an additional replica to the architecture decreases throughput to 49.8 %. The main reason is a bottleneck at the internal network connecting the router with the replicas. The router has to receive and aggregate the output of n replicas until it can forward it to the client. Thus, the internal network has be n times faster than the connection to the client. As a consequence, throughput decreases to 32.6% in NAT/3. To overcome the bottleneck in the internal network a 10 Gbit/s network can be used. This situation however is mostly given in a WAN setup. Table II shows that we were able to almost saturate the client connection in all WAN settings. In the second part of the experiment, we simulate a crashed replica during transfer. We disconnect the network cable of the first replica instantly after starting the download. Thus, after about 5 MB transferred data, the first replica will no longer be available. In contrast to the experiments done before, we only use one test run per setup. We further skip the native setup and NAT/1 because they do not provide fault tolerance. For NAT/2 and NAT/3, we see that in a crashed situation throughput is higher than in a fail free period. The reason it the switch from NAT/2 to NAT/1 and NAT/3 to NAT/2 respectively that reduces the traffic on the internal network interface.

TABLE II. DATA TRANSFER OF 10 MB

experiment	setup	bandwidth	per cent
fail free (LAN)	native	109.7 MB/s ± 1.6	100 %
	NAT/1	107.0 MB/s ± 0.9	97.5 %
	NAT/2	54.61 MB/s ± 1.7	49.8 %
	NAT/3	35.76 MB/s ± 2.5	32.6 %
fail free (WAN)	native	4.00 MB/s ± 0.02	100 %
	NAT/1	3.97 MB/s ± 0.03	99.5 %
	NAT/2	3.98 MB/s ± 0.07	99.6 %
	NAT/3	3.98 MB/s ± 0.03	99.5 %
crashed (LAN)	NAT/2	74.4 MB/s	67.8 %
	NAT/3	45.9 MB/s	41.8 %

V. CONCLUSION

We presented an approach to transparently integrate group communication for active replication on transport layer. A special router maps addresses and multicasts TCP segments to a group of replicas. Replicas can be added and removed at any time. Our prototype is implemented as value-added service that may be integrated into future PaaS clouds so that applications can be replicated on demand. Our evaluation shows reasonable throughput and latency during fail-free periods. In order to saturate client connections, however, active replicas require a fast backend network. Currently, COSCAnet-FT only supports TCP but we work on extending it to other transport protocols. We also plan to compare content and thus extend our solution to cope with and detect Byzantine Faults. It may even be extended by support for scalability.

REFERENCES

[1] Alvisi et al. Wrapping server-side TCP to mask connection failures. In *Proc. of the 20th Ann. Joint Conf. of the IEEE Comp. and Comm. Soc.*, volume 1 of *INFOCOM*, pages 329–337. IEEE, 2001.

[2] N. Ayari, D. Barbaron, Laurent Lefevre, and P. Primet. T2cp-ar: A system for transparent tcp active replication. In *21st Int. Conf. on Adv. Inf. Net. and App.*, AINA, pages 648–655, 2007.

[3] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.

[4] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comp. Surv.*, 36(4):372–421, 2004.

[5] J. Domaschka. *A Comprehensive and Flexible Approach to Transparent Replication of Java Services and Applications*. Dissertation, Fak. für Ingenieurwiss. und Inf., Universität Ulm, August 2012.

[6] R. Ekwall, P. Urban, and A. Schiper. Robust TCP connections for fault-tolerant computing. In *Proc. of the 9th Int. Conf. on Par. and Distrib. Sys.*, pages 501–508, 2002.

[7] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Ithaca, USA, 1994.

[8] V. Jacobson, R. Braden, and D. Borman. RFC1323: TCP extensions for high performance, May 1992.

[9] S. Kächele, J. Domaschka, and F. J. Hauck. COSCA: an easy-to-use component-based PaaS cloud system for common applications. In *Proc. of the First Int. Workshop on Cloud Comp. Platf.*, CloudCP, pages 4:1–4:6. ACM, 2011.

[10] S. Kächele and F. J. Hauck. Component-based scalability for cloud applications. In *Proc. of the 3rd Int. Worksh. on Cloud Data and Platf.*, CloudDP, pages 19–24. ACM, 2013.

[11] S. Kächele and F. J. Hauck. COSCAnet: virtualized sockets for scalable and flexible PaaS applications. In *Proc. of the 6th IEEE/ACM Int. Conf. Utility and Cloud Computing*, UCC. IEEE, 2013.

[12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[13] M. Marwah, S. Mishra, and C. Fetzer. Tcp server fault tolerance using connection migration to a backup server. In *Proc. of the Int. Conf. on Dep. Sys. and Netw.*, DSN, pages 373–382. IEEE, 2003.

[14] L. W McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX Ann. Techn. Conf.*, pages 279–294. CA, 1996.

[15] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comp. Surv.*, 22(4):299–319, 1990.

[16] Z. Shao, H. Jin, B. Cheng, and W. Jiang. ER-TCP: An efficient fault-tolerance scheme for TCP connections. In *Parallel and Distr. Process. and Appl.*, volume 3758 of *LNCS*, pages 139–149. Springer, 2005.

[17] G. Shenoy, S. K. Satapati, and R. Bettati. Hydranet-ft: Network support for dependable services. In *Proc. of the The 20th Int. Conf. on Distr. Comp. Sys.*, ICDCS, pages 699–, DC, 2000. IEEE.