SMAC: State Management for Geo-Distributed Containers

Jacob Eberhardt, Dominik Ernst, David Bermbach
Information Systems Engineering Research Group
Technische Universitaet Berlin
Berlin, Germany
Email: {je,de,db}@ise.tu-berlin.de

Abstract—**In this paper, we describe the design and architecture of SMAC, a system for state management in geo-distributed container deployments. It supports various common state management tasks related to distributed container deployments, e.g., cluster membership, service discovery, health monitoring and load monitoring. Furthermore, it exposes a shared data type abstraction which extensions can be based upon. The system gives up strong consistency guarantees in favor of availability, fault tolerance, scalability and performance. As proof of concept, we implemented SMAC based on a peer-to-peer Conflict-free Replicated Data Type (CRDT) storage system developed for that purpose.**

I. Introduction

Loosely coupled microservice architectures are trending and finding increased adoption. Hence, deployment models have to support frequent releases of independent software components. Continuous delivery is, thus, becoming the new normal.

The Docker [1] container engine is frequently used to realize this continuous delivery process, as (a) containers are lightweight and fast to provision and (b) the imaging feature can be used to build easily distributable entities without external dependencies.

Due to these properties, new software versions are often rolled out by starting new container instances instead of applying an update-in-place strategy. The placement of these new container instances is usually not known a priori when cluster management tools like Swarm [2] or Kubernetes [3] are used. A scheduler determines the physical hosts at deployment time, therefore state information on, e.g., service endpoints, health, or mastership, may change frequently. However, this information still has to be made available to all container instances, both existing and newly spawned ones. Obviously, keeping it in source code is not an option.

A simplistic approach to this problem is the use of configuration management tools, e.g., Chef, Puppet, or Ansible, which automatically retrieve and apply configuration information after an instance has been launched. This creates severe overhead, mitigating the advantages (a) and (b), which can be gained through container adoption. Thus, containerized applications started using coordination services, e.g., etcd [4] or Consul [5], which promise to consistently store information and provide it to a cluster while maintaining high availability.

These coordination services, however, are not suitable for geo-distributed deployments. To maintain a consistent view of data stored, they rely on consensus protocols, e.g., Paxos [6] or Raft [7], which (1) have poor performance properties in geo-distributed settings due to the necessity to exchange multiple rounds of messages with a quorum of replicas and (2) cause unavailabilities for a minority of cluster nodes in case of network partitions.

How can a geo-distributed service for cluster state information exchange be designed to support modern containerized applications? We made two observations enlarging the design space:

1) Some cluster state information is always stale or probabilistic in real networks. Health information, for example, can never be completely accurate at all times, since failure detectors are always imperfect in networks with unbounded communication latency [8].
2) The consensus-based coordination services introduced before do not provide strong consistency in the default read mode. Thus, stale data can be read [9].

These observations indicate that staleness can be tolerated in many cases. For these cases, we propose to relax consistency guarantees and apply the concept of Conflict-free Replicated Data Types (CRDTs) to address aforementioned issues.

In this paper, we make two main contributions:

- We describe the architecture of SMAC, a **S**tate **MA**nagement system for geo-distributed **C**ontainers. It supports asynchronous conflict-free background replication without requiring a quorum to be available.
- We provide the first implementation of an operation-based CRDT storage system.

This paper is structured as follows: Section II discusses existing approaches for propagating state information across containers and outlines related problems. In order to provide necessary background, section III introduces the concept of Conflict-free Replicated Data Types. Afterwards, in section IV, we describe the architecture of SMAC, a CRDT-based system for state management for geo-distributed container deployments. We introduce a proof-of-concept implementation in section V, before we arrive at a conclusion in section VI.

II. Related Work

In this section, we briefly describe existing approaches to providing state information in a distributed container deployment and discuss their suitability for the geo-distributed setting motivated in section I.

A. *Configuration Management*

One way to provide information to container instances is the use of configuration management systems. After an instance has been launched, a configuration management system obtains information by either pulling it from a central server or by having it pushed onto the instance by a coordinator. Often, scripts are executed to ensure an instance's convergence to a specified target state. Established configuration management tools are Ansible [10], Chef [11], and Puppet [12].

This approach requires additional logic to be executed after launching a container instance, which negates the advantages of faster provisioning times gained by using container images. Furthermore, configuration management systems have, if any, limited support for fault-tolerant geo-distributed operation. This can lead to heavily outdated information as, for example, with Chef's asynchronous pull-based master-slave replication [13].

B. *Coordination Services*

In contrast to the configuration management approach, coordination services do not apply scripts or exchange configuration files. They hold state in a durable way and provide an API which enables pull- or notification-based information retrieval. Such services are often used to implement higher level functionality like container health monitoring, dynamic service discovery and locking.

There are several existing coordination services which share closely resembling characteristics: Google developed Chubby [14], a distributed lock and storage service based on the Paxos [6] consensus algorithm. It provides a file system abstraction and aims to provide coordination mechanisms for loosely coupled distributed systems. Yahoo Zookeeper [15], based on the Zab [16] protocol, also aims to provide coordination services for distributed systems based on a file-system-like replicated data structure. The software is open source and provides means to maintain configuration information and implement distributed synchronization. Etcd [4] is a distributed, consistent key-value store for shared configuration and service discovery based on the Raft [7] consensus protocol. Consul [5] is another coordination service with a focus on service discovery and configuration. Based on the Raft [7] consensus protocol, it - like etcd - implements a distributed consistent key-value store, but provides additional service discovery and health-checking features on top of it.

All these services rely on a consensus algorithm, which requires a high amount of communication between nodes and leads to unavailabilities of minorities in case of network partitions. In a geo-distributed setting, this could lead to unavailabilities of the coordination service in multiple regions. The consensus-based services are thus optimized for clusters in a single data center as opposed to geo-distributed deployments. Unlike many users expect, strong consistency is generally not guaranteed by these systems in the default read mode. While writes always require a quorum to succeed, reads can be served by master nodes directly. Since masters are not guaranteed to always be unique, reads can in some cases return stale data.

C. *Others*

Netflix Eureka [17] is a service discovery and load balancing service designed specifically for the Amazon Web Services[1] cloud. It provides a REST interface and can be deployed in clusters for availability and fault tolerance. Eureka does not support geo-distribution, though, since clusters can not span more than one region and is limited to the service discovery feature.

Serf [18] is a group membership and failure detection service based on the SWIM [19] gossip protocol. It is designed as an eventually consistent peer-to-peer system and puts an emphasis on efficient and scalable messaging.

Both services are limited in scope and focus on a specific task related to cluster state management. The SMAC system introduced in section IV, in contrast, supports various tasks by building on conflict-free shared data types as data model. Based on this abstraction, additional cluster state management functionality can be realized.

III. Conflict-free Replicated Data Types

This section motivates and introduces the concept of Conflict-free Replicated Data Types, which are used as the data model and building blocks for higher level functionality in SMAC. Knowing these data types' properties and resulting implications is crucial for understanding the system capabilities and limitations.

Consensus-based coordination services maintain a strongly consistent view of their data by electing a master and serializing all operations through that single replica. These operations are then replayed on all other replicas (slaves). Some services allow reading from slaves directly to reduce load on the master as well as latency. Since, in that case, read operations are not serialized through the master, stale data can be read, i.e., data, for which a newer version exists on some replica. In many cases, this is a worthy trade-off, since staleness can be tolerated by many applications.

Since all writes have to be serialized through a single replica, coordination services fail to perform in geo-distributed settings, where communication latencies become large compared to processing times. Furthermore, consensus protocols require a majority of replicas to be available to successfully perform a write operation. Thus, network failures can lead to unavailabilities for partitioned minorities.

To avoid this, every replica should accept writes and distribute them asynchronously in the background. This introduces a new challenge, though: writes can be concurrent, which means they do not have a defined order. E.g., an integer counter could be set to value 3 on one replica, while it is concurrently set to 5 on another replica. Depending on the order in which these operations arrive at other replicas, these replicas can end up in conflicting states.

[1] http://aws.amazon.com/

There are various strategies to resolve such conflicts, e.g., timestamp-based resolution or pushing conflicting versions to the application layer. There is another approach, however: Conflict-free Replicated Data Types (CRDTs) avoid conflicts altogether. CRDTs, a theoretical concept by Shapiro et al. [20], are shared data types which define their operations in a way, that they are non-conflicting when concurrent, i.e., the order of execution does not matter for the result. Due to non-conflicting operations, all replicas are in equal state as soon as all information has been delivered to all replicas. The outcome of concurrent operations, referred to as concurrent semantics, is part of the data type specification as well. For example, a "contains" operation on an element in a set, which was concurrently added and removed before, could return true or false depending on the specification.

There is no algorithmic approach to arrive at a CRDT specification with desired concurrent semantics. Fortunately, the technical report [21] lists many specifications for various types with varying concurrent semantics, e.g., last-write-wins, add-wins, remove-wins. Amongst these data types are counters, registers, sets, (multi)maps, sequences and graphs.

Based on the replication strategy employed, there are two types of CRDTs: Commutative Replicated Data Types (CmRDTs) propagate operations, while Convergent Replicated Data Types (CvRDTs) propagate state. Shapiro et al. [20] provide a formal proof showing the equality in expressiveness of the two approaches. We now describe CmRDTs in more detail, as they are used by the prototype we introduce in section IV, while CvRDTs are only briefly discussed.

A. Commutative Replicated Data Types

A data type is called Commutative Replicated Data Type, if it uses operation-based replication and has the following properties:

1) Updates ordered by the happened-before relation are executed in that order. Updates not ordered by that relation are defined as concurrent and can be executed in any order.
2) A communication channel ensures that replicas receive all updates in that order.
3) Concurrent updates are commutative.

This definition ensures non-concurrent updates to be applied in the same order on all replicas. Only concurrent updates may be applied in different orders on different replicas. Due to the commutativity property of these updates, the result after the update operation is the same, independent of order. Hence, replica states converge given the properties defined above.

To ensure the delivery of all updates to all replicas, a reliable communication channel is required. Exactly once delivery has to be guaranteed, since operations are generally not idempotent. Furthermore, the delivery order needs to comply with the happened-before relation [22], which is also referred to as causal delivery.

A simple example of a CmRDT is an integer counter: The data structure itself is a simple integer. There are two operations: increase, which increases the integer's value by 1 and decrease, which decreases the value by 1. Obviously, the increase and decrease operations commute in any case.

We provide the - to our knowledge - first implementation of a CmRDT storage system in section IV.

B. Convergent Replicated Data Types

A data type is called Convergent Replicated Data Type, if it uses state-based replication and has a commutative, idempotent and associative merge function. This merge function combines local state and state received from a remote replica and computes a new state. A communication channel with eventual delivery is sufficient.

There are multiple systems implementing CvRDTs, e.g., Basho Riak [23] or akka [24].

IV. The SMAC System

Based on the the shared data types described in section III, we designed SMAC, a CRDT-based state management system for geo-distributed container deployments, which we now introduce. It can be used as a typical state management service in geo-distributed deployments, but can also store and distribute application meta data in a scalable and fault tolerant way. Not all functionality offered by consensus-based coordination services can be provided due to the relaxed consistency model. We argue, however, that such requirements need to be avoided during design phase in the first place, as they constitute an inherent performance bottleneck, limit scalability and impair availability.

In this section, we describe and discuss SMAC's deployment, interface, and functionality, before we introduce our proof of concept implementation in section V.

A. System Overview and Deployment

SMAC is a distributed peer-to-peer system consisting of a cluster of nodes running SMAC instances. Figure 1 depicts a typical deployment with one consuming application, which is co-deployed with a SMAC instance on the same host machine.

A single SMAC instance, however, can support multiple containerized applications on one host. This instance communicates with other instances of the system to replicate shared data types and with that make them available to remote application instances. Due to the system's architecture, unlike with consensus-based coordination services, no minimal cluster size is required. The system remains writable as long as the local SMAC instance is healthy. Due to the nature of CRDTs, the overall system is available as long as a single node can be accessed.

B. Interface and Data Model

An application, deployed in a container, accesses a SMAC instance via a REST interface. This interface provides CRUD[2] operations for the data types described in section III, as well as more advanced operations specific to state management scenarios, e.g., health monitoring and service discovery.

[2]Create, Read, Update and Delete

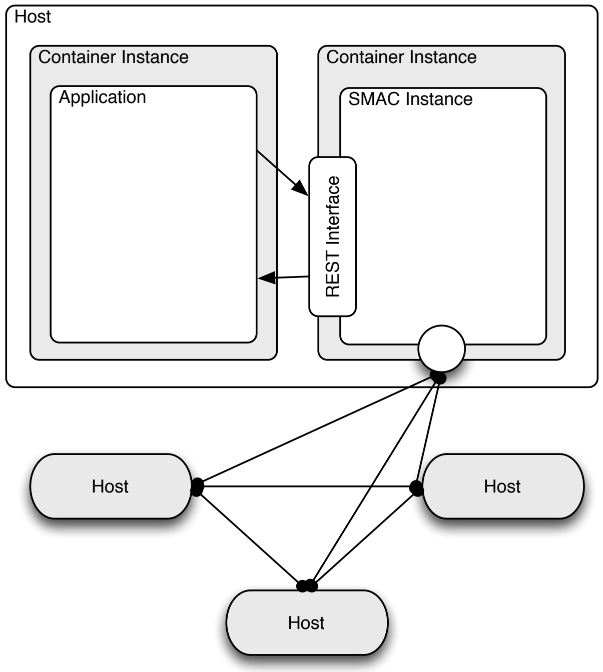

Fig. 1. SMAC Deployment.

Shared data types are the abstraction SMAC works with internally for information exchange. Thus, an appropriate data type has to be chosen manually for every new state management scenario. To assist with that, we will now show how we realized three common scenarios, which can serve as guiding examples when modeling further functionality on top of CRDTs.

Cluster Health Monitoring: A common state management scenario is to maintain and make available cluster health information to members. To realize this, an add-wins set with timer-based removal of entries can be used: Node IDs (e.g., IP addresses) are held in a set. Every time a node receives a message, it writes the sender's ID to the set. These messages can be application-specific or dedicated heartbeats. For each ID, a local timer is started. After it reaches a certain threshold, the item is removed, as the node is assumed to be unresponsive and hence unhealthy. In case of concurrent add and remove operations, add should take precedence.

Load Monitoring: For scheduling or load balancing purposes, load information can be valuable. The number of open connections, for example, can serve as a metric. To provide such values, a map data type can be used: Node IDs are mapped to integer counters. The counters are increased or decreased by nodes depending on their active connections.

Service Discovery: Due to dynamic container placements, service locations change and this information needs to be available to potential consumers. To realize service discovery, a last-write-wins register can be used: At startup, the service writes it's entry point to this register. Alternatively, a remove-wins set specification can be used as well. This allows to expose several endpoints in parallel, which can be useful when rolling out new versions.

C. Discussion

The SMAC system provides state management in a cluster while being highly scalable and partition tolerant. It can solve the availability and latency problem established consensus-based coordination services have and is highly suitable for various state management scenarios, as exemplified in IV-A.

Some tasks, however, can not be solved due to the relaxed consistency model: Locking and leader election, for example, require strong consistency and can thus not be supported. Such requirements, however, should be avoided in geo-replicated systems in the first place, since theoretical results imply that there is no solution with desirable liveness properties [25].

Consensus-based coordination services become unavailable for minorities in case of network partitions. While such outages are easily tolerable in single data center deployments, they can not be dealt with in geo-distributed settings. In contrast to that, SMAC is always available. Information provided can be stale, but that is certainly preferable compared to having an unavailable coordination service. Minorities can still make progress by working with stale data or updating it.

Compared to the key-value interfaces consensus-based coordination services typically provide, the data type abstraction used in our system requires more thought to be put in how to map a state management task to the available data structures. This process, however, is straight forward in most cases as can be seen in section IV-A.

Besides state management tasks on an infrastructure level, SMAC can also be used for the exchange of application metadata. Especially data related to geo-distributed deployments can be leveraged rather easily, e.g., trending videos per region on a video on demand platform.

Write operations in SMAC always return after being executed locally. They are then distributed asynchronously to other replicas in the background, which results in low response times. As opposed to this, consensus-based coordination services need to forward the write request to the master, which then has to contact a majority of replicas for the operation to be successful. Especially in a geo-distributed setting, this results in comparatively high response latencies.

V. Proof-Of-Concept

In this section, we introduce our proof-of-concept implementation of the SMAC system. Furthermore, we provide first experimental results obtained while testing the prototype.

A. Prototypical Implementation

Internally, the SMAC prototype is realized as an operation-based distributed peer-to-peer CmRDT storage system. Figure 2 depicts an overview of the prototype's architecture.

As a fault tolerant distributed system, SMAC is designed to be deployed in a cluster. Each node executes at least one instance of the software written in Java inside a Docker [1] container. The core entities are the *CmRDT Handlers*. They control the life cycle and manipulation of concrete CmRDT

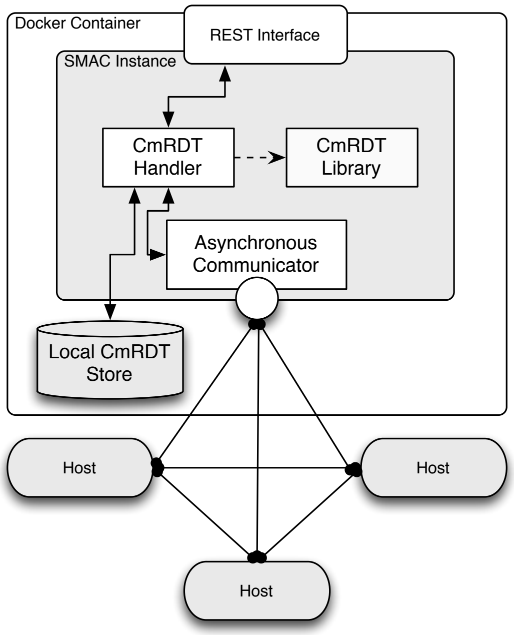

Fig. 2. SMAC Prototype Architecture.

instances, instantiated from a specific CmRDT implementation contained in the *CmRDT Library*. These instances are persisted in the *Local CmRDT Store*, which resides in memory in the current implementation. Handler-methods are invoked through the *REST Interface* in order to expose Conflict-free Replicated Data Types and higher level state management abstractions, e.g., service discovery operations. Currently, only request-based interaction is implemented, but notifications are supported by the architecture. Once applied locally, operations are asynchronously distributed to the other nodes in the cluster. All communication between processes is handled by the *Asynchronous Communicator*. This component ensures causal delivery of operations, which is required by CmRDTs as explained in section III. The current prototype employs direct messaging, but gossip-based communication is possible as well.

B. Experimental Evaluation

To test our SMAC prototype in a geo-distributed setting, we provisioned a cluster of three virtual machines in three regions (Singapore, Frankfurt, Northern California) of the Amazon Elastic Compute Cloud[3]. Based on an Ubuntu image with a Java Runtime Environment, we installed Docker as container engine and launched containerized SMAC instances on all three host machines.

This setup was used to perform various tests:

Convergence of CRDTs: To test the correctness of our CmRDT implementations, we tried to violate convergence properties by performing large numbers of requests (both, concurrent and causally dependent) on all hosts. We did not detect any violations with regards to convergence. The cURL[4] tool was used to perform automated requests to the REST interface.

Concurrent Semantics of CRDTs: After ensuring convergence, we deployed test client applications in another set of container instances accessing the SMAC instances through their REST interface on every host. We performed predefined sequences of requests and compared the returned values to the expected results to verify correct concurrent semantics of implemented CRDTs. No deviations from the expected results were detected.

Request Latency: As SMAC uses peer-to-peer asynchronous background replication and thus returns after committing a write operation locally, request latency is very low. For comparison, we deployed etcd in a similar cluster. First results indicate that request latency of etcd is significantly larger, which was expected due to the consensus algorithm employed.

VI. Conclusion

In this paper, we introduced and discussed the design of SMAC, a state management system for container deployments supporting geo-distribution. The system gives up strong consistency guarantees in favor of availability, fault tolerance and performance. We showed feasibility by sketching solutions for common state management use cases as well as application metadata exchange. Furthermore, we provided a prototypical implementation of SMAC based on a peer-to-peer CmRDT storage system developed for that purpose.

As future work, we plan to further evaluate the prototype by testing it in a large geo-distributed cluster with a realistic workload. We also aim to perform a thorough experimental performance comparison of SMAC and etcd. Moreover, we will add support for additional state management use cases and implement CRDT specifications not yet supported.

References

[1] Docker Inc. (2016) Docker. [accessed on Feb 8, 2016]. [Online]. Available: https://www.docker.com/

[2] ——. (2016) Docker swarm. [accessed on Feb 8, 2016]. [Online]. Available: https://docs.docker.com/swarm/

[3] Google Inc. (2016) Kubernetes cluster manager. [accessed on Feb 8, 2016]. [Online]. Available: http://kubernetes.io/

[4] CoreOS Inc. (2016) etcd - a highly-available key value store for shared configuration and service discovery. [accessed on Feb 8, 2016]. [Online]. Available: https://coreos.com/etcd/

[5] Hashicorp Inc. (2016) Consul. [accessed on Feb 8, 2016]. [Online]. Available: https://www.consul.io/

[6] L. Lamport, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[7] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX Annual Technical Conference*, 2014, pp. 305–320.

[8] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 225–267, 1996.

[9] K. Kingsbury. (2016) Jepsen: etcd and consul. [accessed on Feb 8, 2016]. [Online]. Available: https://aphyr.com/posts/316-jepsen-etcd-and-consul

[10] Ansible Inc. (2016) Ansible. [accessed on Feb 5, 2016]. [Online]. Available: http://www.ansible.com/

[3]https://aws.amazon.com/ec2/

[4]https://curl.haxx.se/

[11] Chef Software Inc. (2016) Chef. [accessed on Feb 5, 2016]. [Online]. Available: https://www.chef.io/

[12] Puppet Labs. (2016) Puppet. [accessed on Feb 5, 2016]. [Online]. Available: https://puppetlabs.com/

[13] Chef Software Inc. (2016) Chef server replication across failure domains. [accessed on Feb 3, 2016]. [Online]. Available: https://www.chef.io/chef/replication/

[14] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 335–350.

[15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX Annual Technical Conference*, vol. 8, 2010, p. 9.

[16] B. Reed and F. P. Junqueira, "A simple totally ordered broadcast protocol," in *proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*. ACM, 2008, p. 2.

[17] Netflix Inc. (2016) Eureka. [accessed on Feb 2, 2016]. [Online]. Available: https://github.com/Netflix/eureka/

[18] Hashicorp Inc. (2016) Serf. [accessed on Feb 6, 2016]. [Online]. Available: https://www.serfdom.io/

[19] A. Das, I. Gupta, and A. Motivala, "Swim: Scalable weakly-consistent infection-style process group membership protocol," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 303–312.

[20] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems*. Springer, 2011, pp. 386–400.

[21] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of Convergent and Commutative Replicated Data Types," Research Report RR-7506, 2011. [Online]. Available: https://hal.inria.fr/inria-00555588

[22] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[23] Basho Technologies, Inc., "Riak data types library," 2014, http://docs.basho.com/riak/2.0.2/dev/using/data-types/, accessed on Nov 14, 2014. [Online]. Available: http://docs.basho.com/riak/2.0.2/dev/using/data-types/

[24] Typesafe Inc. (2016) Akka distributed data. [accessed on Feb 1, 2016]. [Online]. Available: http://doc.akka.io/docs/akka/2.4.1/java/distributed-data.html

[25] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.