2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing

Application-Aware Byzantine Fault Tolerance

Wenbing Zhao
Department of Electrical and Computer Engineering
Cleveland State University, 2121 Euclid Ave, Cleveland, OH 44115
wenbing@ieee.org

***Abstract*—Byzantine fault tolerance has been intensively studied over the past decade as a way to enhance the intrusion resilience of computer systems. However, state-machine-based Byzantine fault tolerance algorithms require deterministic application processing and sequential execution of totally ordered requests. One way of increasing the practicality of Byzantine fault tolerance is to exploit the application semantics, which we refer to as application-aware Byzantine fault tolerance. Application-aware Byzantine fault tolerance makes it possible to facilitate concurrent processing of requests, to minimize the use of Byzantine agreement, and to identify and control replica nondeterminism. In this paper, we provide an overview of recent works on application-aware Byzantine fault tolerance techniques. We elaborate the need for exploiting application semantics for Byzantine fault tolerance and the benefits of doing so, provide a classification of various approaches to application-aware Byzantine fault tolerance, and outline the mechanisms used in achieving application- aware Byzantine fault tolerance according to our classification.**

Keywords*—*Application Semantics, Application-Aware Byzantine Fault Tolerance, Application Nondeterminism, Deferred Byzantine Agreement, Dependability, Intrusion Resilience

I. Introduction

Autonomous mission-critical computer systems must be made resilient against both hardware failures and cyber attacks. Byzantine fault tolerance has been intensively studied over the past decade as a way to help achieve this goal [1]. Highly efficient Byzantine fault tolerance algorithms have been developed [2], [3], [4]. Speculative execution of partially ordered requests helps further minimize the end-to-end latency [5]. However, standard state-machine-based Byzantine fault tolerance algorithms require deterministic application processing and sequential execution of totally ordered requests. We argue that these constraints would impede the adoption of the Byzantine fault tolerance techniques in practice, for example:

- Practical systems often involve nondeterministic operations when they execute clients' requests, and the states of the replicas would diverge if the application nondeterminism is not controlled.
- Sequential execution of all requests to the replicated server often results in unacceptable low system throughput and long end-to-end latency.

To overcome these issues, the application semantics must be tapped, as demonstrated by the first seminal work on modern Byzantine fault tolerance, PBFT [2]. In PBFT, a networked file system (NFS) is replicated for Byzantine fault tolerance. Some requests to the replicated NFS server would trigger the access of the local physical clock, which constitutes a nondeterministic operation. PBFT exploited the application semantics of NFS to identify this nondeterministic operation and designed a control mechanism accordingly. Furthermore, PBFT also identified the requests that do not modify the server's state based on the knowledge of the application semantics of NFS, and designed a mechanism for these read-only requests so that they are not totally ordered. Since PBFT, many other mechanisms and algorithms have been developed to facilitate the adoption of Byzantine fault tolerance techniques for practical distributed systems by exploiting application semantics at different levels. In this article, we provide an overview of these works and refer to this line of research work as application-aware Byzantine fault tolerance. Detailed discussions on individual works can be found in [1].

This paper makes the following contributions:

- We present a strong argument that it is not practical to treat the sever application as a black box when replicating it for Byzantine fault tolerance and, that it is essential to exploit the application semantics. In addition to performance overhead and replica consistency issues, we identify scenarios when the replicated system may deadlock if all requests must be executed sequentially according to a total order.
- We provide a classification of various application-aware Byzantine fault tolerance approaches based on the levels of application semantics that were exploited.
- We outline the mechanisms designed for achieving application-aware Byzantine fault tolerance according to our classification. These mechanisms could serve as a guideline when building dependability solutions for new systems.

II. The Need for Tapping Application Semantics

Byzantine fault tolerance algorithms designed for state-machine replication, such as PBFT [2] and Zyzzyva [4], concern only the total ordering of requests to be delivered to the replicated server replicas. Hence, such algorithms can be used by any application as long as the replicated component acts deterministically as a state machine, *i.e.,* given the same request delivered in the same total order, all replicas would go through the same state transitions (if any), and produce exactly the same reply. However, this does not mean that we should treat each application as a black box and employ a Byzantine fault tolerance algorithm as it is by totally ordering all requests and executing them according to the total order. In the following, we present three major motivations for taking an application-aware approach to Byzantine fault tolerance.

978-1-4799-5079-9/14 $31.00 © 2014 IEEE
DOI 10.1109/DASC.2014.17

A. Reduce Runtime Overhead

There are two main types of runtime overhead introduced by Byzantine fault tolerance algorithms:

1) Communication and processing delays for each remote invocation due to the need for total ordering of requests, which impacts the end-to-end latency as seen by a client.
2) The loss of concurrency degrees at the replicated server due to the sequential execution of requests, which impacts the system throughput (*i.e.,* how many requests can be handled by the replicated server per unit of time).

By exploiting application semantics, we can introduce the following optimizations:

- Reducing the end-to-end latency by minimizing the total ordering of requests. There is no need to totally order read-only requests (*i.e.,* requests that do not modify the server state). For some stateless session-oriented applications, source ordering (instead of total ordering) may be sufficient for all requests [6].
- Enabling concurrent processing at the server replicas for some requests. Independent requests can be delivered and processed concurrently. Similarly, commutative requests can be delivered and processed concurrently. Doing source ordering alone for requests also increases the system throughput.

B. Respect Causality and Avoid Deadlock

General purpose Byzantine fault tolerance algorithms are designed for simple client-server applications where the clients do not directly interact with each other and send requests to the replicated server independently. For multi-tiered applications with sophisticated interaction patterns [7], the basic assumption for these Byzantine fault tolerance algorithms may no longer hold and hence, if used in a straightforward manner, may lead to two problems: (1) causality violation, when the total order imposed on two or more requests is different from their causal order and, (2) deadlocks, when sequential execution of requests is imposed when concurrent processing of some requests is mandatory according to the application design [8]. In these cases, application semantics must be tapped to discover the causal ordering between requests and identify what requests must be delivered concurrently. Otherwise, the integrity and the availability of the system will be lost.

C. Ensure Strong Replica Consistency

State-machine replication requires that the replicas behave deterministically when processing requests. However, many applications involve some form of nondeterministic operations, such as taking a timestamp and accessing a pseudo random number generator. Without the knowledge of application semantics, it is usually impossible to know whether or not a request would trigger a nondeterministic operation [9]. If such nondeterministic operation is not controlled, the states of the replicas might diverge and the replicas might produce different replies to the client. Hence, exploiting application semantics not only helps in minimizing the runtime overhead, it is essential to ensure the safety and integrity of a Byzantine fault tolerant system.

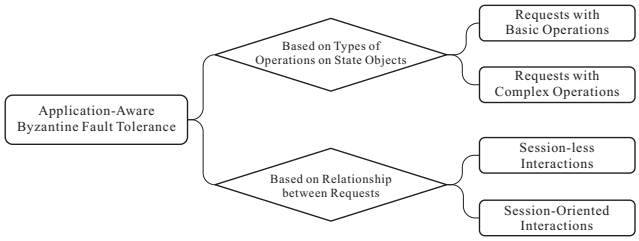

Fig. 1. Classification of application-aware Byzantine fault tolerance.

III. Classification of Application-Aware Byzantine Fault Tolerance

As shown in Figure 1, application-aware Byzantine fault tolerance (AA-BFT) can be classified based on the following two criteria.

The first criterion concerns the operations performed at each replica while executing a request sent by a client. We assume that the operations access one or more state objects. We classify the AA-BFT approaches based on whether or not the operations on state objects are limited to basic or complex types.

- Basic operation: An operation is basic provided that it is a read or write operation on a state object, or the creation or deletion of an object. Furthermore, if the operation is write or create, the value of the object is set deterministically, *i.e.,* it is either provided by the client that issues the request, or set by the system deterministically. Note that the create operation may effectively access two state objects (one of them is the object created as the result of the create operation). When an object factory is used, or the object is created within a specific hierarchy such as within a particular directory, the create operation will inevitably access both the object factory or the parent directory, and the object created.
- Complex operation: it refers to an operation beyond basic operations, such as increment and decrement for integer numbers, and append or truncate for text strings. We also consider an operation complex if it sets a new value to a state object nondeterministically, such as using the local clock value or using a value produced by a pseudo random generator.

The second criterion concerns whether or not the context to which the request belongs is considered. The context, if it is present, correlates a number of requests issued by the same or different clients within the same *session*. If a request is issued within such a session, the representation of the context is typically included in the message. If the context is not considered (or there is no such context), requests from different clients are assumed to be sent independently.

We first discuss the resulting two categories using the first criterion. We then elaborate the implications of the context as used in the second criteria. Furthermore, we assume that

the application is designed to execute concurrent requests with multiple threads, such as in a thread pool, for maximum throughput as well as for the capability of handling multiple concurrent clients.

A. Requests with Basic Operations

For this category, the relationship between different requests is determined by the target state objects and the corresponding operations. Two requests are considered dependent with each other if they access at least one common state object and the operation on that object from any of the requests is a write operation. Otherwise, requests are considered independent.

Note that for a write or a create operation, we assume that the new value is determined deterministically. Furthermore, create operations are independent from each other, and similarly delete operations are independent from each other. However, create and delete operations must be totally ordered with respect to each other (*i.e.,* they are considered dependent with each other).

For requests that are dependent with each other, they must be delivered and executed sequentially according to some total order at all nonfaulty replicas to ensure strong replica consistency. Independent requests, on the other hand, can be delivered and executed concurrently with respect to each other. Note that a request might be independent of one or more requests, however, it might be dependent on some other requests and such dependency must be preserved during execution.

B. Requests with Complex Operations

For this category, a request may be involved with basic as well as complex operations. The dependency among the requests can be analyzed in a similar manner as that for requests with basic operations by classifying complex operations into read-oriented and write-oriented types. However, such analysis is not adequate.

Even if two requests access the same object with at least one of them engaging in a write-oriented operation, they might not be conflict with each other depending on the complex operations involved. For example, if both requests are involved with the increment operation on the same state object, they are considered commutative and hence, can be delivered and processed concurrently.

The complexity of the operation could also require additional mechanisms to ensure strong replica consistency. For example, if either the value to be returned to the client, or the new value to be assigned to the state object is determined based on the reading of the local clock, or on the output from a pseudo random number generator, additional mechanisms are needed to ensure that all nonfaulty replicas decide on the same value [10].

C. Session-Oriented Interactions

For session-less interactions between the clients and the server replicas, the requests are assumed to be sent independently. Hence, the replicas may determine a total order of the requests sent by different clients arbitrarily (usually according to the order in which they are received at the primary). Furthermore, for concurrent requests, they may be batched together for total ordering. Of course, the requests sent by the same client are ordered according to the order in which they were sent to preserve causality.

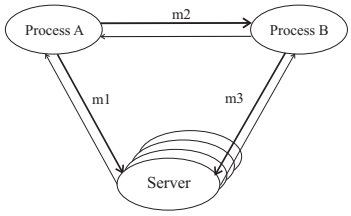

Fig. 2. A scenario that two requests might be causally related. In this example, m_3 might causally depends on m_1, hence, m_3 must be ordered and delivered after m_1 is processed.

For requests sent within a session, however, they may be correlated. Furthermore, session-oriented applications typically are multi-tiered. This correlation among the requests imposes the following constraints on the ordering and handling of requests:

- *Total ordering of requests must respect causality*: The most obvious constraint due to requests correlation is that the replicas can no longer impose a total order according to the order in which the requests are received because doing so might violate the causal ordering among the requests. Consider the example scenario shown in Figure 2. If process A issues a request m_1 to the server replicas and subsequently sends a request m_2 to process B (*i.e.,* A sends m_2 prior to receiving the reply for m_1). Subsequently, process B issues a request m_3 to the server replicas. It is apparent that m_1 must be ordered ahead of m_3 because m_3 might depend on m_1.
- *Forced concurrent processing*: In multi-tiered applications, nested remote invocations are common. If two or more replicated servers issue nested invocations to each other in response to requests sent from their clients, as shown in Figure 3, the nested requests must be delivered and executed prior to the initial requests are fully processed. Dictating a total order for the requests from the clients and those for nested invocations could lead to deadlocks.

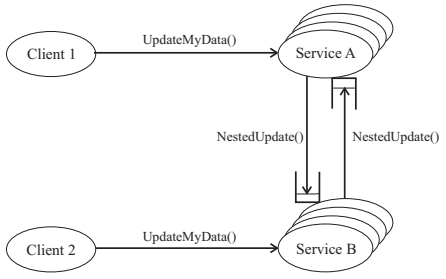

Fig. 3. A scenario that could lead to deadlock without concurrent processing of requests to the replicated servers. In this example, the remote invocation $NestedUpdate()$ must be processed concurrently with remote invocation $UpdateMyDate()$ to avoid deadlock at the replicated server.

On the other hand, the correlation among the requests within a session also offers additional opportunities for optimization, which we will introduce in the next section.

IV. Application-Aware Byzantine Fault Tolerance Mechanisms

In this section, we describe the mechanisms used in AA-BFT for various applications. We first outline the system model used in these mechanisms. Then we describe the mechanisms in three parts. In the first two parts, we elaborate the mechanisms designed for basic operations, and those to accommodate complex operations. These mechanisms are applicable to both session-less and session-oriented applications, respectively. In the third part, we present the mechanisms designed specifically for session-oriented application, regardless of the type of operations (*i.e.,* basic or complex).

A. System Model

All AA-BFT mechanisms described below are based on the system model introduced in PBFT [2], or with straightforward extension for session-oriented interactions. It is assumed that the mechanisms are operating in an asynchronous distributed system, and all messages exchanged are protected by unforgeable digital signatures or message authentication code.

For client-server applications, the server is replicated with sufficient replication degrees. For multi-tiered applications, one of the middle-tier servers or the backend server is replicated in the same way as the client-server applications except that a replica may issue nested requests to, and receive the corresponding replies from, other components in the system.

For simplicity, we assume that some Byzantine fault tolerance algorithm such as PBFT [2] are directly used at each replica to reach Byzantine agreement on the message total ordering and other values if necessary, although the task of Byzantine agreement can be separated out and provided as a service by a separate cluster [11].

B. Requests with Basic Operations

Research on requests with basic operations is heavily influenced by techniques developed for transaction processing [12], in particular, the concurrency control theories. This is not surprising because the problem layout for requests with basic operations is rather similar to that for transaction processing. It is intuitive to identify and handle read-only requests differently from requests that modify the server state so that the end-to-end latency for read-only requests is minimized. Requests that might modify one of more state objects are analyzed for potential dependency based on the type of operations (*e.g.,* read or write) and target state objects. In a conservative approach introduced in section IV-B3, a partial order is imposed on the requests that allows independent requests to be processed in parallel. A special case is requests partitioning, where requests that operate on disjoint state objects can be separated based on some simple measure and handled concurrently without further dependency analysis.

1) Read-only requests: The first mechanism to optimize the performance of Byzantine fault tolerant systems for this type operations is described in PBFT [2]. If it is known that a request will not change any of the state objects, *i.e.,* it is a read-only request, it can be delivered at a replica immediately without being totally ordered first. To ensure that the client reads from at least one nonfaulty replica, it collects $f + 1$ matching replies from different replicas before it delivers the reply. If tentative execution is employed for speculative execution at the replicas, the client would need to collect $2f+1$ matching replies before it delivers the reply, and in case when the client fails to collect $2f + 1$ matching replies, it resends the request as a non-read-only request.

2) Requests partitioning: Requests can be partitioned if they operate on disjoint state objects. Typically, requests can be separated based on some simple measure, such as an identifier contained in the request, without complicated dependency analysis (as shown in Section IV-B3). Requests that belong to different partitions can be executed concurrently and may be handled by different groups of server replicas for scalability. This approach is seen to be employed in farsite [13], as well as in Byzantine fault tolerant Web services coordination [14], [15].

3) Dependency analysis: The first systematic mechanism to handle this type of requests (beyond read-only requests) is introduced in [5]. The center piece of the mechanism is a predefined concurrency matrix that can be used to determine whether or not two requests are independent based on the operation and argument (representing the state object to be accessed) specified in the request. The message delivery is controlled by a parallelizer component that is sandwiched between the Byzantine agreement layer (for message total ordering) and the server application (referred to as executor in [5]), as shown in Figure 4.

The parallelizer dynamically tracks the dependency of totally ordered requests according to the concurrency matrix. The concurrency matrix is populated based on application-specific rules. To facilitate concurrent execution of requests, it is assumed that each replica runs a pool of worker threads

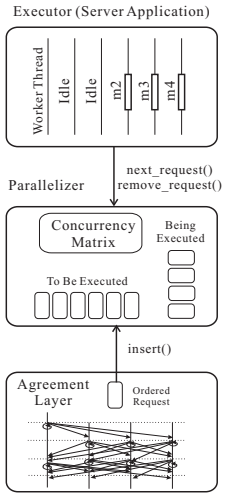

Fig. 4. The parallelizer and its interaction with other components.

that actively fetch ordered requests from the parallelizer.

When a Byzantine agreement on the total order of a request is reached, the method `insert()` of the parallelizer is invoked and the request is placed in the "to-be-executed" queue according to the total order. When a worker thread is launched, it invokes the `next_request()` method to block waiting for the next request to execute. If a worker thread finishes executing a request, it invokes the `remove_request()` method so that the parallelizer can remove the request from its "being-executed" queue. Subsequently, the worker thread invokes the `next_request()` method to fetch a new request to execute. When the `next_request()` method is called, the parallelizer first checks if the "to-be-executed" queue is empty. If true, the parallelizer blocks the call. Otherwise, the parallelizer decides on when to return the next request in the following way:

- If the "being-executed" queue is empty, the parallelizer retrieves the request at the head of the "to-be-executed" queue and returns it immediately.
- If the "being-executed" queue is not empty, the parallelizer retrieves the request at the head of the "to-be-executed" queue and consults with the concurrency matrix to see whether or not the request has a conflict with any of the requests in the "being-executed" requests. If the request is independent from all requests in the "being-executed" queue, the parallelizer returns it immediately. Otherwise, the parallelizer must wait until all conflicting requests in the "being-executed" queue have been removed before returning the request.

C. Requests with Complex Operations

In addition to the mechanisms designed for handling requests with basic operations, current research has focused on the following two aspects of complex operations: (1) commutative requests and (2) replica nondeterminism.

1) Commutative requests: If some requests are commutative while others are conflicting with each other, the mechanism described in Section IV-B3 can be easily modified to accommodate commutative requests. More specifically, the entries corresponding to the commutative requests will be labeled as none-conflicting. No additional changes are required.

The above approach would require the total ordering of all requests. In [16], an algorithm was designed such that commutative requests are not totally ordered and it may take as few as two communication steps to handle such requests. There are two tradeoffs, however. First, it requires the use of $5f + 1$ to tolerate up to f faulty replicas. Second, speculative execution of requests is involved and if the speculation on the ordering of a request is wrong, the execution would have to be rolled back and the recovery might be expensive.

One way to systematically ensure that all requests to a service are commutative is to implement the service using commutative (or convergent) replicated data types [17]. In this case, it is possible to synchronize the state of the replicas only when it is necessary or periodically, as described in [18].

2) Replica Nondeterminism: Some simple location-specific nondeterminism can be masked by using a wrapper function. The wrapper translates location-specific values such as process id to a group-wide id, which ensure strong replica consistency. This practice has been long used to build replication-based fault tolerance systems [19] and is used in [20] as an extension to PBFT. The use of a wrapper does not introduce any additional communication step.

In [9], a classification of replica nondeterminism beyond simple wrappable types is provided and a set of mechanisms for controlling replica nondeterminism is presented. The classification is based on two criteria: whether or not nondeterministic operations and their associated values can be determined prior to the execution of a request, or whether or not nondeterministic values decided by one replica can be verified by another replica. Some of the nondeterminism can be controlled by adding one more communication step to the Byzantine agreement step for the request to ensure that all nonfaulty replicas make the same nondeterministic decisions. Other nondeterministic decisions must be controlled via an extra Byzantine agreement round.

D. Session-Oriented Interactions

In this section, we describe the mechanisms used to address the two potential issues we identified in session III-C and the mechanisms to improve the performance by exploiting the correlation among requests with a session.

1) Preventing Causality Violation: The causality violation issue might arise only when clients communicate directly with each other without the knowledge of the replicated server. In multi-tired session-oriented applications, this may happen frequently. For example, in a Web services atomic transaction or business activity, participants often communicate with each other as well as with the coordination service [14], [15]. In these applications, the causality of requests is often reinforced at the application level, *e.g.,* a request is queued until the one that precedes it is received [14], [15]. Therefore, for applications that have provisions for causality detection and enforcement, the causality violation issue goes away if we allow concurrent delivery of requests.

2) Avoiding Deadlocks: The problem can be resolved with concurrent delivery of requests that have parent-children relationship using the mechanism described in [8].

3) Source ordering: For some multi-tiered applications with replicated middle-tier server and a backend server managing all persistent state, it is argued in [6] that maintaining source ordering of requests is sufficient due to the commutativity of the operations at the replicated middle-tier server, or due to the constraint imposed by the database server. Furthermore, only $2f + 1$ middle-tier server replicas are needed to tolerate up to f faulty replicas. A client, as well as the backend server, must collect $f + 1$ matching reply messages or nested request messages to deliver the reply or request.

4) Deferred agreement: A session by definition defines the beginning and the ending of an interaction between a group of participants. For some requests received at the replicated server, their total ordering can be determined right before the end of the session, thereby, reducing the runtime

overhead. Deferred agreement is different from the batching mechanism used in general purpose Byzantine fault tolerance algorithms [2]. Batching is done to the group of requests that arrive concurrently, while deferred agreement applies to requests received over time during a session.

As described in [15] in the context of Web services atomic transactions coordination, the registration requests sent by participants of a transaction are delivered right away without total ordering at the coordinator replicas. When the transaction initiator decides to commit a transaction, a registration update phase is started at the coordinator replicas to exchange the registration requests each replica has received. A replica initiates the first phase of the two phase commit when it has collected registration records from $2f + 1$ (out of a total of $3f + 1$) replicas (including itself). The replica then build a superset of the registration set, and initiates the first phase of the two phase commit. At the end of the first phase, a replica performs a round of Byzantine agree on the registration set as well as the transaction outcome. When a Byzantine agreement is reached, a replica proceeds to the second phase of two phase commit.

For long sessions, it is also possible to periodically synchronize the state of the replicas, as did for collaborative editing sessions [21].

5) Requests partitioning: Typically, requests that belong to different sessions are handled by different state objects at the middle-tier server in a multi-tiered application where the middle-tier is replicated for Byzantine fault tolerance. Hence, requests can often be simply partitioned based on sessions (more specifically, using the session id contained in the request messages) at the middle-tier server [6].

V. Conclusion

In this paper, we presented an overview of application-aware Byzantine fault tolerance techniques. We pointed out that it is essential to tap the application semantics to build practical solutions for Byzantine fault tolerance. In particular, we argued that application semantics not only can be used to improve the runtime performance, it is necessary to preserve the causality relationship of the requests and avoid potential deadlocks, as well as to control replica nondeterminism. Furthermore, we provided a classification of different approaches to application-aware Byzantine fault tolerance, and outlined existing and potential approaches in the context of the classification.

References

[1] W. Zhao, *Building Dependable Distributed Systems*. Wiley-Scrivener, 2014.

[2] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, 2002.

[3] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "Hq replication: A hybrid quorum protocol for byzantine fault tolerance," in *Proceedings of the Seventh Symposium on Operating Systems Design and Implementations*, Seattle, Washington, 2006.

[4] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," in *Proceedings of 21st ACM Symposium on Operating Systems Principles*, 2007.

[5] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *Proceedings of International Conference on Dependable Systems and Networks*, 2004.

[6] H. Chai and W. Zhao, "Byzantine fault tolerance for session-oriented multi-tiered applications," *Int. J. of Web Science*, vol. 2, no. 1/2, pp. 113–125, 2013.

[7] ——, "Interaction patterns for byzantine fault tolerance computing," in *Computer Applications for Web, Human Computer Interaction, Signal and Image Processing, and Pattern Recognition*, ser. Communications in Computer and Information Science, T.-h. Kim, S. Mohammed, C. Ramos, J. Abawajy, B.-H. Kang, and D. Slezak, Eds. Springer Berlin Heidelberg, 2012, vol. 342, pp. 180–188.

[8] W. Zhao, L. Moser, and P. M. Melliar-Smith, "Deterministic scheduling for multithreaded replicas," in *Proceedings of the IEEE International Workshop on Object-oriented Real-time Dependable Systems*, Sedona, AZ, 2005, pp. 74–81.

[9] H. Zhang, W. Zhao, P. M. Melliar-Smith, and L. E. Moser, "Design and implementation of a byzantine fault tolerance framework for non-deterministic applications," *IET Software*, vol. 5, pp. 342–356, 2011.

[10] W. Zhao, "Integrity-preserving replica coordination for byzantine fault tolerant systems," in *Proceedings of the IEEE International Conference on Parallel and Distributed Systems*, Melbourne, Victoria, Australia, December 2008, pp. 447–454.

[11] H. Chai and W. Zhao, "Byzantine fault tolerance as a service," in *Computer Applications for Web, Human Computer Interaction, Signal and Image Processing, and Pattern Recognition*, ser. Communications in Computer and Information Science, T.-h. Kim, S. Mohammed, C. Ramos, J. Abawajy, B.-H. Kang, and D. Slezak, Eds. Springer Berlin Heidelberg, 2012, vol. 342, pp. 173–179.

[12] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[13] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: Federated, available, and reliable storage for an incompletely trusted environment," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002, pp. 1–14.

[14] H. Chai, H. Zhang, W. Zhao, P. M. Melliar-Smith, and L. E. Moser, "Toward trustworthy coordination for web service business activities," *IEEE Transactions on Services Computing*, vol. 6, no. 2, pp. 276–288, 2013.

[15] H. Zhang, H. Chai, W. Zhao, P. M. Melliar-Smith, and L. E. Moser, "Trustworthy coordination for web service atomic transactions," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 8, pp. 1551–1565, 2012.

[16] P. Raykov, N. Schiper, and F. Pedone, "Byzantine fault-tolerance with commutative commands," in *Principles of Distributed Systems*, ser. Lecture Notes in Computer Science, A. Fernndez Anta, G. Lipari, and M. Roy, Eds. Springer Berlin Heidelberg, 2011, vol. 7109, pp. 329–342.

[17] M. Shapiro, N. Preguia, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems*, ser. Lecture Notes in Computer Science, X. Dfago, F. Petit, and V. Villain, Eds. Springer Berlin Heidelberg, 2011, vol. 6976, pp. 386–400.

[18] H. Chai and W. Zhao, "Byzantine fault tolerance for services with commutative operations," in *Proceedings of the IEEE International Conference on Services Computing*. Anchorage, Alaska, USA: IEEE, June 27 - July 2 2014.

[19] W. Zhao, P. M. Melliar-Smith, and L. E. Moser, "Low latency fault tolerance system," *The Computer Journal*, vol. 56, no. 6, pp. 716–740, 2013.

[20] M. Castro, R. Rodrigues, and B. Liskov, "Base: Using abstraction to improve fault tolerance," *ACM Transactions on Computer Systems*, vol. 21, no. 3, pp. 236–269, 2003.

[21] W. Zhao and M. Babi, "Byzantine fault tolerant collaborative editing," in *Proceedings of the IET International Conference on Information and Communications Technologies*. IET, 2013, pp. 233–240.