Byzantine Fault Tolerant Event Stream Processing for Autonomic Computing

Hua Chai and Wenbing Zhao
Department of Electrical and Computer Engineering
Cleveland State University, Cleveland, Ohio 44115
Email: wenbing@ieee.org

***Abstract*—Event stream processing has been used to construct many mission-critical event-driven applications, such as business intelligence applications and collaborative intrusion detection applications. In this paper, we argue that event stream processing is also a good fit for autonomic computing and describe how to design such a system that is resilient to both hardware failures and malicious attacks. Based on a comprehensive threat analysis of event stream processing, we propose a set of lightweight mechanisms that help achieve Byzantine fault tolerant event processing for autonomic computing. The mechanisms consist of voting at the event consumers and an on-demand state synchronization mechanism triggered when an event consumer fails to collect a quorum of matching decision messages. We also introduce an evidence-based safe-guarding mechanism that prevents a faulty event consumer from inducing unnecessary rounds of state synchronization.**

Keywords*—*Byzantine Fault Tolerance, Event Stream Processing, Autonomic Computing, Dependability, Integrity, Trust

I. Introduction

Event stream processing is a powerful way of constructing distributed systems that take input continuously from various sources (referred to as event producers), and generate alerts and derived events according to predefined rules for consumption by their clients (referred to as event consumers). Event stream processing can be regarded as a variation of message-oriented middleware technology, and it has been used in many mission-critical applications, such as business intelligence and collaborative intrusion detection. It is well-known that message-oriented middleware is a good fit to construct autonomic systems [1], [2]. Indeed, the event stream processing technology is readily used to build autonomic systems because:

- An autonomic system must continuously sense its environment (external operational context). The input from the environment can be modeled as events to be streamed to the autonomic system. The sensors would serve as the event producers.
- The event processing agents constitute as the autonomic components. The logic as dictated by the purpose and know-how of a specific autonomic system can be implemented by a set of event processing rules on top of an event processing engine.
- The alerts or derived events in an event stream processing system are equivalent to decisions made by an autonomic component as the result of changes in the environment. The system to be self-managed is equivalent to an event consumer.

Furthermore, the use of the event stream processing model for autonomic computing makes it possible to use event processing middleware, such as Esper [3], which simplifies the design and implementation of autonomic components, as shown in Figure 1. The use of the event processing model also decouples the sensors, the autonomic components, and the system to be self-managed, which makes it easier to upgrade and maintain individual components.

For autonomic systems, self-protection and self-healing are essential requirements. It is apparent that these requirements cannot be met without the use of redundancy and fault tolerance techniques. To be resilient against malicious faults, such as those caused by cyber attacks, the event processing agent must be replicated and Byzantine fault tolerance techniques must be employed.

In this paper, we study the threats on the event processing agent (serving as the autonomic component) and present how to achieve Byzantine fault tolerance with a set of lightweight mechanisms. Specifically, our mechanisms are designed to achieve the following objectives:

- The event processing agent must be made continuously available despite arbitrary faults. Streams of events generated by various sensors (*i.e.,* event producers) will continuously arrive at the agent, and they may trigger important decisions for self-configuration or self-optimization.
- The event processing agent must maintain its integrity under cyber attacks. An adversary might attempt to subvert the integrity of the event processing agent instead of crashing it, because it could lead to more severe damages, *e.g.,* a compromised replica might issue wrong reconfiguration decisions to the system

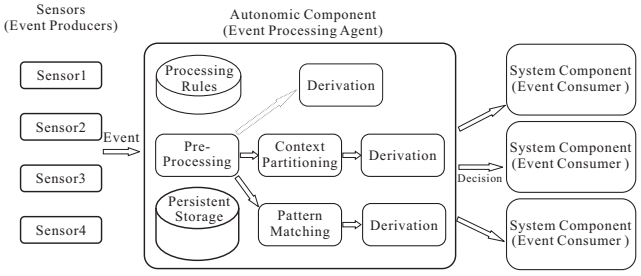

Fig. 1. Overview of a event stream processing system for autonomic computing.

978-1-4799-5079-9/14 $31.00 © 2014 IEEE
DOI 10.1109/DASC.2014.28

when no reconfiguration should take place.

This paper makes the following contributions:

- We present a comprehensive threat analysis on autonomic systems constructed using event stream processing. In particular, we highlight the threats on the validity and consistency of replicas of the event processing agent.
- We introduce a novel on-demand synchronization mechanism to ensure that all nonfaulty replicas of the event processing agent use the same context partition for decision making in the presence of malicious sensors (*i.e.,* event producers). The replicas of the event processing agent engage in a round of state synchronization only when inconsistency in decisions is detected at the system. Hence, during normal operation, our mechanism incurs minimum runtime overhead in response time for self-configuration and self-optimization.
- We introduce a voting mechanism to be used by the autonomic system so that decisions issued by compromised replicas are not accepted by an autonomic system.
- We use an evidence-based mechanism to validate each synchronization request to prevent a compromised component in an autonomic system from inducing unnecessary rounds of state synchronization.

II. Background

A. Byzantine Fault Tolerance

Byzantine fault tolerance (BFT) is a promising technology to increase the resiliency of distributed and networked systems against malicious faults [4]. BFT can be used to protect a replicated components against Byzantine faults, which denote arbitrary faults including hardware faults as well as malicious application-levels faults as the result of cyber attacks. The core mechanisms of Byzantine fault tolerance include:

- Replication. A mission-critical component is replicated so that even if a small portion of the replicas are compromised, the remaining replicas can continue providing correct services to other components in the system.
- Byzantine agreement. To ensure continuous availability as well as service integrity of the replicated component, the state of nonfaulty replicas must remain consistent. This requires the use of a Byzantine agreement algorithm to guarantee the following in the presence of Byzantine faults:
  - All nonfaulty replicas deliver and execute inputs to the component in the same total order.
  - All nondeterministic operations, if any, are rendered deterministic so that given the same input executed in the same order, all nonfaulty replicas go through the same state transitions and produce the same output (such as a decision or a reply to a client).

Well-known BFT algorithms such as PBFT [5] and Zyzzyva [6] are designed for general client-server applications with synchronous request-reply communication between the client and the server. Event stream processing (including autonomic systems), on the other hand, involves drastically different communication styles between the replicated event processing agent and other components (details will be given in Section II-B). Furthermore, Event stream processing systems are time sensitive and an extended delay in processing may render the resulting decision obsolete or harmful. For autonomic systems, the delay in decision making might render them reconfigure too late. BFT algorithms are designed to maintain the consistency of the replicas in an asynchronous environment, and they may not be able to make progress in some corner cases. To reduce the negative impact of these corner cases, the use of Byzantine agreement should be minimized.

B. Event Stream Processing

An event stream processing system consists of three major components, the event producers, one or more event processing agents (EPAs), and the event consumers, as shown in Figure 1. Event producers are the entities that produce events. In an autonomic system, sensors constitute as event producers. In an autonomic system, event consumers are typically components in the system that are instructed for changes or reconfigurations. The brain of an EPA is determined by a set of rules. The rules are typically predefined and they may be updated dynamically via some user interfaces to accommodate policy changes. In an autonomic system, the set of rules reflect the purpose and know-how of the system.

The EPA is usually powered by specialized event stream processing middleware frameworks such as Esper. Normally, incoming events are first pre-processed according to predefined criteria and non-qualified events are discarded. The middleware framework typically provide either user interfaces or application programming interfaces to revise the processing rules dynamically. The EPA processes incoming events according to the execution rules.

Event processing may be stateless, where each event is processed independently from other events. As shown in Figure 1, for stateless processing, the derivation step occurs right after the pre-processing step. During the derivation step, an event might be translated, enriched, or projected into another event for the event consumer to handle [7]. These stateless operations manipulate the attributes of the incoming events. In this paper, we do not consider stateless event processing because autonomic systems typically do not involve this type of processing.

More complex event processing are stateful in that multiple events are considered collectively. Stateful processing usually belong to one of the following two types:

- Aggregation oriented: Attributes of a sequence of events possibly belong to different event streams are computed to derive higher level information.
- Pattern matching oriented: Predefined event patterns are being matched against the incoming events.

In both types, the objective of the processing is to identify an important situation which may trigger a reconfiguration of the system. Stateful event processing is typically applied to a subset of the events in one or multiple streams at a time. The subset is determined by the context as defined in the processing rule. The events may be grouped into different context partitions based on a number of criteria:

- Time: The context is determined based on the timestamps of the events. Events that belong to one or more time intervals may belong to a temporal context. Different temporal contexts may overlap. In fact, sliding windows are one of the most common temporal contexts used in complex event processing.
- Location: Events in the context share the same location where events occurred.
- Identifier: Events in the context share the same identifier (such as customer id) as part of the event attributes.
- State: The events included in each context are determined by the external state. The external state may in turn be temporal, spatial, or identifier based.

The communication pattern between different components of an event processing system is drastically different from that of client-server systems. In a client-server system, the client typically sends requests to the server for processing, and blocks waiting for the corresponding replies (*i.e.,* for each request, there is one reply generated by the server as the result) before it issues the next one. In event processing systems, on the other hand, different components communicate via one-way messages:

- An event producer continuously streams events to an EPA and does not expect any reply.
- An EPA sends a decision message to the designated event consumer (*e.g.,* a reconfiguration decision to a particular system component) whenever one is generated at the end of the derivation step.

III. Threat Analysis

In this section, we analyze potential threats that could compromise the dependability and integrity of event processing. In particular, we discuss new threats that arise due to the replication of EPA. We do not consider general threats that could target any online services, such as distributed denial of service attacks. We divide the threats under consideration into three categories based on the originator of the threat: (1) threats imposed by a faulty event producer; (2) threats imposed by a faulty EPA; and (3) threats imposed by a faulty event consumer.

A. Threats Imposed by a Faulty Event Producer

A compromised event producer could attack the system in the following ways:

- Omit important events.
- Emit wrong events.
- Send conflicting events to different EPA replicas.

All these attacks could lead to wrong decisions or the absence of needed decisions. The third attack is a specific attack when an EPA is replicated and it aims to cause the replicas to generate inconsistent decisions, which could confuse the event consumers.

The first two attacks cannot be directly addressed by replicating an EPA because if all nonfaulty replicas make their decisions based on a set of events containing faulty events or the absence of important events, all of them would make the same wrong decisions. This is a validity problem rather than a consistency problem. These attacks must be addressed via two mechanisms:

- Incorporate redundant sensors (for event production) such that events can be cross-examined. Omission of events or faulty events sent by faulty event producers can be masked or excluded.
- Use robust pattern matching and/or derivation algorithms so that the presence of a single faulty event or the omission of an event would not adversely impact the matching or derivation result. For example, taking the medium value of the attributes of a collection of events is more robust than taking the average value.

The third attack can be controlled by ensuring the consistency of the set of events used to compute the decision at all nonfaulty EPA replicas.

B. Threats Imposed by a Faulty EPA

A compromised EPA could attack the system by sending wrong or conflicting decisions to other system components (*i.e.,* event consumers) for reconfiguration, or refusing to send such commands as it should. This type of threats can be mitigated by replicating the EPA, and by using voting on the decisions sent by different EPA replicas at each event consumer.

When an EPA is replicated, a compromised replica could disseminate conflicting events to other replicas during a round of state synchronization. This type of threats can be controlled by using Byzantine agreement to ensure the consistency of nonfaulty replicas.

C. Threats Imposed by a Faulty Event Consumer

Without replication, an event consumer cannot directly attack the EPA in the context of autonomic computing. When replication is used with our mechanisms, the system is vulnerable to a new attack, in that a faulty event consumer could attempt to trigger unnecessary rounds of state synchronization, which is an expensive step. This type of threats can be controlled by using a non-forgeable proof for each state synchronization request.

IV. Byzantine Fault Tolerance Mechanisms

In this section, we describe the Byzantine fault tolerance mechanisms for event stream processing. We first formulate the correctness properties that our mechanisms aim to ensure. We then present our mechanisms in detail. We conclude this section by proving that our mechanisms satisfy the correctness properties.

A. Correctness Properties

The correctness properties concern a decision accepted by a nonfaulty event consumer.

P1. The decision must have been generated by the majority of nonfaulty EPA replicas.

P2. If a faulty event producer issues conflicting events to different nonfaulty EPA replicas, at most one of the conflicting events is included in the computation for the decision.

Property P1 ensures that a nonfaulty event consumer never accepts a wrong decision issued by faulty EPA replicas. Property P2 ensures that a faulty event producer cannot cause nonfaulty EPA replicas to make different decisions without being detected.

B. Mechanisms Description

The validity of event processing in the presence of faulty event producers must be ensured by using redundant event producers and robust matching and derivation algorithms. The number of faulty event producers that can be tolerated by a system depends on the algorithms used. Detailed description of such algorithms are beyond the scope of this paper.

The consistency of event processing at different replicas can be achieved trivially by using a general-purpose BFT algorithm such as PBFT. However, doing so would entail the total ordering of all event messages, which would incur too much runtime overhead to satisfy the soft realtime requirement for autonomic systems. Hence, we choose to use an on-demand state synchronization scheme. In the absence of faulty event producers, no state synchronization is necessary, which significantly reduces the runtime overhead.

We assume that all messages exchanged between different components of the system are protected by a security token such as digital signature or message authentication code. This ensures that the sender is properly authenticated (*i.e.,* one sender could not impersonate as another sender) and the integrity of the message is protected (*i.e.,* a tempered message could be detected).

We assume the availability of $3f + 1$ EPA replicas to tolerate up to f faulty replicas. Each replica is assigned a unique id k, where k varies from 0 to $3f$. Each context partition P is assigned a monotonically increasing sequence number, s, which we refer to as the context id. For a context partition $P(s)$ with events e_1, e_2, ... e_n, an EPA replica maintains a history hash $h_s = hash(e_1) \oplus hash(e_2) \oplus ... \oplus hash(e_n)$, where $hash()$ is a secure hash function such as SHA-1, and $\oplus$ is the exclusive or operator. An important property for the history hash is that it is independent from the relative ordering of the events, *e.g.,* $h(e_1, e_2) = h(e_2, e_1)$. We assume that . A decision produced by a context partition $P(s)$ has the form $<\text{DECISION}, s, h_s, c>$, where c is the content of the decision. The history hash h_s is included in the decision as proof of the context partition formation.

1) Normal Operation: In the absence of faulty event producers, the system operates in the normal operation mode. The presence of faulty EPA replicas has no impact to the normal mode of operation provided that our assumption of up to f faulty EPA replicas is held. An event consumer waits to receive $2f + 1$ matching decision messages from different EPA replicas before it accepts and acts on the decision. Two decision messages are matching when they have the same context id s, the same history hash h_s, and the same decision content c. During normal operation, our mechanism does not introduce any additional runtime overhead, and it only incurs the overhead of piggybacking the history hash in each decision message and voting at the event consumer, the normal operation mechanism.

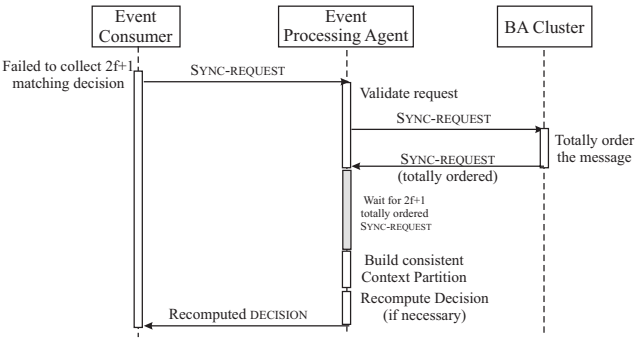

Fig. 2. Main steps in on-demand state synchronization.

2) On-Demand State Synchronization: In the presence of faulty event producers, an event consumer might not be able to collect $2f + 1$ matching decision messages. If the first $2f + 1$ decision messages received do not completely match, an event consumer sets a timer and attempts to collect more decision messages towards $2f + 1$ matching messages before the timer expires. When this attempt fails, the event consumer sends a synchronization request to all EPA replicas. The synchronization request takes the form $<\text{SYNC-REQUEST}, cid, s, DS>$, where cid is id of the event consumer, s is the decision id, DS is the set of mismatched, digitally signed decision messages as proof for the need of synchronization. The main steps of a round of on-demand state synchronization is illustrated in Figure 2.

On receiving a synchronization request $<\text{SYNC-REQUEST}, cid, s, DS>$, an EPA replica i initiates a round of state synchronization for the context partition $P(s)$ by broadcasting to all replicas a state-sync message via a Byzantine agreement service, provided that DS are valid (*i.e.,* each signed decision message in the set is verified). If the DS is invalid, the event consumer is apparently compromised and appropriate reconfiguration would take place.

The state-sync message sent by replica i has the form $<\text{STATE-SYNC}, i, s, CP_s^i>$, where s is the context partition id, CP_s^i is the list of event records in the partition at replica i. Each event record consists of an event id and the digest of the event. The replica also collects the state-sync messages sent by other replicas. Upon receiving the first $2f + 1$ totally ordered state-sync messages sent by different replicas, an EPA replica proceeds to building a consistent context partition CP_s according to the following rules:

- If an event e is included in all $2f + 1$ state-sync messages, it is included in CP_s.

- If an event e is included in at least $f+1$ state-sync messages, but no conflicting event e' is found in the $2f+1$ state-sync messages, it is included in CP_s. Two events are said to be conflicting if they have the same event id but different content.
- If conflicting events are detected, they must have been sent by a faulty event consumer. Such events are excluded from CP_s, and the corresponding event producer is blacklisted. No future events will be accepted from the blacklisted event producer.

Once CP_s is built, an EPA replica checks to see if its own context partition is identical to CP_s. If not, it replaces its own context partition with CP_s, and repeats the matching and derivation step. It is guaranteed that the decision messages generated using CP_s by different nonfaulty EPA replicas are consistent, which ensures the acceptance of the decision message at the event consumer.

C. Proof of Correctness

We now prove that our Byzantine faulty tolerance mechanisms satisfy the correctness properties described in section IV-A.

P1. The decision must have been generated by the majority of nonfaulty EPA replicas.

Proof: To accept a decision, an event consumer must collect $2f+1$ matching decision messages from different EPA replicas. Because there are at most f faulty EPA replicas, at least $f+1$ of the decision messages are sent by nonfaulty EPA replicas, which constitutes the majority of the nonfaulty EPA replicas. This proves that P1 is satisfied. ■

P2. If a faulty event producer issues conflicting events to different nonfaulty EPA replicas, at most one of the conflicting events is included in the computation for the decision.

Proof: It is easy to see that it is impossible for an event consumer to accept a decision that is computed directly using two conflicting events. When a decision is accepted at an event consumer, it means $2f+1$ EPA replicas have produced exactly the same decision, among them, $f+1$ must be nonfaulty replicas and they would only take at most one of the conflicting events by definition.

The only other scenario we must consider is when two or more event consumers are targets of the decision from the same context partition s. We prove that our mechanisms satisfy P2 in this case by contradiction. Assume that a faulty event producer sends two conflicting events $e1$ and $e2$ to two different subsets of EPA replicas. Further assume that the decision accepted at one event consumer $C1$ included $e1$ in the computation of the decision, and the decision accepted at another event consumer $C2$ included $e2$ in the computation of the decision. According to our mechanisms, for $C1$ to accept the decision, it must have collected from a quorum $R1$ of $2f+1$ EPA replicas. Similarly, for $C2$ to accept the decision, it must also have collected from a quorum $R2$ of $2f+1$ EPA replicas. Because there are $3f+1$ total EPA replicas, $R1$ and $R2$ must intersect in at least $f+1$ replicas, among which at least one must be a nonfaulty EPA replica. This is impossible because a nonfaulty EPA replica does not accept two conflicting events by definition. This completes our proof. ■

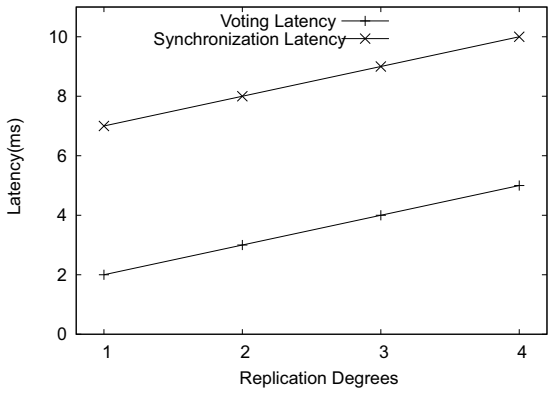

Fig. 3. Medium latency versus replication degree for voting and state synchronization.

V. Implementation and Performance Evaluation

We have implemented our Byzantine fault tolerance mechanisms with an open-source event stream processing framework called Esper in Java. The Byzantine agreement cluster is implemented in house in Java as well. The evaluation of the runtime performance of our mechanisms is carried out in a testbed consisting of 14 HP BL460c blade servers and 18 HP DL320G6 rack-mounted servers connected by a Cisco 3020 Gigabit switch. Each BL460c blade server is equipped with two Xeon E5405 processors and 5 GB RAM, and each DL320G6 server is equipped with a single Xeon E5620 processor and 8GB RAM. The 64-bit Ubuntu Linux server operating system is run at each node.

We carried out a micro-benchmarking of the performance of our mechanisms using two metrics:

- The delay caused by voting at each event consumer. This constitutes the additional latency incurred by our mechanism in the critical path of autonomic reconfiguration during normal operation.
- The latency for completing a round of state synchronization. This is the runtime overhead incurred by our mechanism in the presence of faulty event producers and/or EPA replicas.

During the experiments, we varied the replication degree for EPA as well as the Byzantine agreement cluster from f=1 (total 4 replicas) up to f=4 (total 13 replicas). The event producers and event consumers are deployed in different nodes in the same local area network. The benchmarking results are summarized in Figure 3. As can be seen in Figure 3, the medium latency for voting as well as the medium latency for state synchronization increase linearly with the replication degree. This trend indeed is expected because the number of messages to be collected in voting and in state synchronization increase linearly with the replication degree.

We choose to use medium instead of mean value of the latency because of unpredictable long tail in the distribution of the latency measurement, as shown in Figure 4. The measurement confirms that with high probability that voting and state synchronization can be completed within a few milliseconds, occasionally we do see relatively large delays. However, the

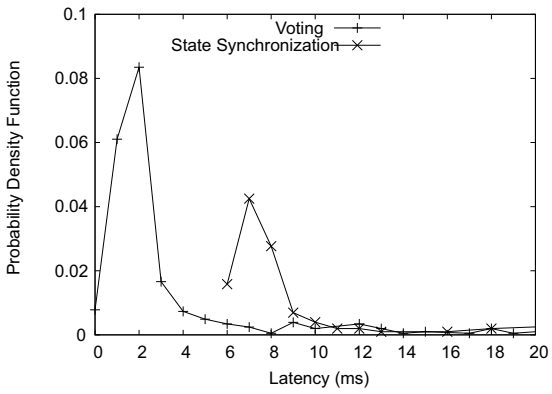

Fig. 4. Probability density function of the latency for voting and state synchronization for f=1.

occasional large delays may be attributed to the asynchrony in communication and processing in our testbed instead of our mechanisms. For systems that require tight realtime deadlines, special networking equipment and realtime operating systems must be used to avoid such unpredictable delays.

VI. Related Work

A crash-fault tolerance solution for event stream processing was proposed in [8] with active replication of EPA. The proposed solution is specifically designed for systems constructed using software-transactional memory [9]. The most interesting mechanism is to allow concurrent processing of events by exploiting the transaction processing model. Specifically, a validation step is introduced prior to the commit of each transaction to ensure that transactions are committed according to the total order of the events that triggered the transactions.

Previously, we extended the work in [8] to tolerate Byzantine faulty EPA replicas by using a BFT algorithm [5] to totally order all event messages [10]. Although similar in objective, the work described in this paper takes a completely different approach. First, we do not rely on the software-transactional memory processing model. Second, by exploiting the semantics of event stream processing for autonomic processing, we designed an on-demand state synchronization mechanism to ensure the consistency of EPA replicas and the decisions sent to event consumers.

Fault tolerant autonomic computing has been studied by several researchers [11], [1]. However, the solutions proposed are restricted by the crash-fault model used and they cannot survive malicious attacks, unlike that proposed in this paper.

Application-aware Byzantine fault tolerance is a promising research direction because it gives guidelines on how to develop Byzantine fault tolerance solutions for practical systems by using basic Byzantine agreement constructs in a way similar to designing secure systems by using cryptography primitives [12], [13], [14], [15]. This paper belongs to this line of work.

VII. Conclusion

In this paper, we discussed how to render an autonomic system built with the event stream processing technology resilient against malicious faults. We first argued that event stream processing is a good fit to building autonomic systems, and that Byzantine fault tolerance is essential for mission-critical autonomic systems. We then analyzed various threats against event processing in the context of autonomic computing, and outlined the principles on controlling these threats.

The main contribution of our work is a set of lightweight mechanisms that help achieve Byzantine fault tolerant event processing for autonomic computing. The mechanisms consists of voting at the event consumers and an on-demand state synchronization mechanism triggered when an event consumer fails to collect a quorum of matching decision messages. We also introduced an evidence-based safe-guarding mechanism to prevent a faulty event consumer from inducing unnecessary rounds of state synchronization. Finally, we implemented the mechanisms and assessed their runtime overhead.

References

[1] A. Almeida, J.-P. Briot, S. Aknine, Z. Guessoum, and O. Marin, "Towards autonomic fault-tolerant multi-agent systems," in *Proceedings of the 2nd Latin American Autonomic Computing Symposium*, Petropolis, RJ, Brésil, 2007.

[2] A. Padovitz, A. Zaslavsky, and S. W. Loke, "Awareness and agility for autonomic distributed systems: platform-independent and publish-subscribe event-based communication for mobile agents," in *Proceedings of the 14th International Workshop on Database and Expert Systems Applications*. IEEE, 2003, pp. 669–673.

[3] http://esper.codehause.org.

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

[5] 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.

[6] 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.

[7] O. Etzion and P. Niblett, *Event Processing in Action*. Manning Publications, 2010.

[8] A. Brito, C. Fetzer, and P. Felber, "Multithreading-enabled active replication for event stream processing operators," in *Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems*. IEEE, 2009, pp. 22–31.

[9] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, 1995, pp. 204–213.

[10] H. Zhang and W. Zhao, "Concurrent byzantine fault tolerance for software-transactional-memory based applications," *International Journal of Future Computer and Communication*, vol. 1, no. 1, pp. 47–50, 2012.

[11] Y. Tohma, "Incorporating fault tolerance into an autonomic-computing environment," *IEEE Distributed Systems Online*, vol. 5, no. 2, p. 0003, 2004.

[12] 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.

[13] 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.

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

[15] 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.