Towards Practical Intrusion Tolerant Systems: A Blueprint*

[Extended Abstract]

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

ABSTRACT

In this paper, we present the blueprint of a novel middleware infrastructure that can be used to build mission-critical systems with increased resiliency against intrusion attacks. The infrastructure is designed to be practical and it imposes a well-defined structure on the application by adhering the principle of the separation of concerns: (1) the processing of each application request is carried out at a single execution node, and if the execution node becomes faulty, another node can take over immediately; (2) the state of the server is replicated transparently across a pool of state replicas, and a novel append-only strategy is used so that not only the state is protected against hardware failures, it is resilient to attacks aimed to cause state corruption and destruction; (3) the fault monitoring, execution and state integrity checking, and system configuration management are carried out by distinct components which by themselves are replicated.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Fault tolerance, reliability, availability, and serviceability; D.4.5 [**Operating Systems**]: Reliability—*Checkpoint/restart, fault-tolerance, verification*

General Terms

Reliability

1. INTRODUCTION

Byzantine fault tolerance is a fundamental technique to build mission critical systems that can continue correct operation under intrusion attacks [1]. However, we have yet to see its widespread adoption in practice due to a number of issues: (1) it incurs relatively high runtime overhead, (2) it requires the replicas to run deterministically, which seriously limits the type of applications that can be supported, (3) most of all, it assumes that the replicas fail independently under intrusion attacks, which may be the biggest roadblock for its practical use. Even though n-version programming has been touted as a potential effective means to achieve replica diversity [4], the extremely high development cost significantly impedes its practical use. Therefore, how to build practical intrusion tolerant systems is still largely an open issue.

In this paper, we examine the problem from a different angle and propose an intrusion tolerance infrastructure that is radically different from traditional Byzantine fault tolerance frameworks. We do not attempt to cope with all arbitrary faults, instead, we focus on those incurred by external adversaries exploiting the design and implementation deficiencies of the server software. We propose to use the following strategies to make the server software more resilient to such intrusion attacks:

Focusing on malformed requests. Even though the vulnerabilities that led to the exploit should eventually be patched, we anticipate that the fix will take considerable amount of time and for an immediate solution, the effort should be focused on the detection of malformed requests that materialized the attack and the prevention of similar requests from being accepted by the server.

Separation of execution and state replication. While execution can be easily aborted and restarted in case of abnormal situations, once the state is corrupted by an intrusion attack, it is a lot harder to recover. A big issue in the traditional Byzantine fault tolerance design is that the execution and the state are tied together. We propose to separate the execution and the state management for better intrusion resiliency. The novel approach enables us to run a single instance of execution process to handle each application request, which will lead to superb runtime performance while protecting the state by replication.

Multi-version state replication. A naive replication of the server state cannot ensure the integrity of the state if the execution process is compromised. Therefore, we propose to use an append-only strategy for state protection against intrusions. In the append-only state replication, all state

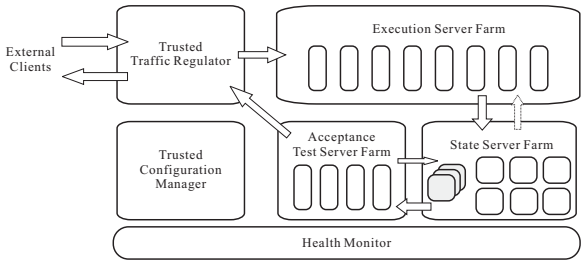

Figure 1: The architecture of the intrusion tolerance infrastructure.

changes are propagated to the state replicas from the execution process as usual, however, previous states stored in the replicas are *not* modified (that is, they are read-only once appended to the state log), instead, a new checkpoint is derived from the current state change and it is *appended* to the state log (this idea is inspired by the design of the Google file system [2]). In effect, we rely on multi-version state storage for intrusion resilience, which is easy to do, instead of the n-version programming, which is extremely costly.

Acceptance testing on execution integrity. To facilitate faster intrusion detection, we advocate the use of acceptance test [4] to validate the state changes due to the processing of each (or each batch of) application request. On failure of an acceptance test, the current execution node is forcefully shutdown and a new process is launched at a different node. Furthermore, the request that caused the problem is quarantined and analyzed. Such malformed requests and all future requests that bear similar signatures are rejected.

2. INTRUSION TOLERANCE INFRASTRUCTURE

The architecture of the proposed intrusion tolerance infrastructure is shown in Figure 1. As can be seen, the architecture consists of the following components: trusted traffic regulator, trusted configuration manager, execution server farm, acceptance test server farm, state server farm and the health monitor. During normal operation, an application request is first routed to the traffic regulator for preliminary screening, and then it is forwarded to an execution server for processing. At the end of request processing, the execution server propagates the state changes together with the reply to a set of state replicas located in the state server farm. The reply is then passed to an acceptance test server for verification. The acceptance test server verifies the integrity of the processing and the state changes by invoking an application-supplied acceptance test routine. The response is subsequently forwarded to the traffic regulator for sending back to the client. The details of each component is explained below.

2.1 Trusted Traffic Regulator

The trusted traffic regulator is the only entry point and exit point of application requests and responses, respectively. It performs the following duties:

It performs preliminary screening of incoming requests, possibly including the authentication of the clients. An important screening step is to compare the request with the known malformed request signatures accumulated so far. Non-qualified requests are logged and rejected without further processing. All accepted requests are logged and subsequently forwarded to an execution server process in the execution server farm, and verified responses are logged and sent back to the corresponding clients.

The traffic regulator acts as an intelligent load balancer. The decision on which target server to send to depends on the type of the request. If the request is read-only, it can be forwarded to any available server according to the load situation. However, if it can potentially lead to a state change, and/or it is within a previously established session, the request is sent to a specific server.

If a response has failed the acceptance test, the traffic regulator is informed. The traffic regulator removes the corresponding request from the verified request log and puts it to the quarantine area for analysis. The malformed request signature set might be updated due to the detection of this new malformed request. Furthermore, all future requests coming from the same host and port number are blocked.

The traffic regulator itself is protected by replication for high availability. During normal operation, only the primary actively performs the duty of traffic regulation and it transfers its state periodically to the backups in an asynchronous manner for high performance. If the primary fails, some information might get lost during the fail-over. We do not think this is going to be problem because most of the state can be rebuilt (such as malformed request signatures). There is a concern about the exactly-once execution of application requests – the task is handled by the state server farm.

2.2 Execution Server Farm

This server farm hosts the execution servers. Note that only those application requests that have passed the preliminary screening will be forwarded to the execution severs. Unlike traditional Byzantine fault tolerance where both execution and state are replicated across a set of nodes, a request is handled by only a single execution process in the farm unless the result of the execution fails the acceptance test, or the process does respond within a predefined time period, in which case, the request is rejected or rerouted to a different server, depending on the circumstances.

It is hard to tell immediately if a slow server is caused by a malformed request, a residue software defect, a hardware failure, or simply an overload situation. The slow server is forcefully shutdown and the request is rerouted to a lightly-loaded server. If the abnormal situation reoccurs a number of times consecutively, the request is regarded as malformed and the traffic regulator is informed. The request is subsequently quarantined, analyzed and rejected.

2.3 State Server Farm

The state server farm consists of a state master (replicated across several nodes) responsible for maintaining the meta of the state (such as the checksums of the state objects), and many more servers, referred to as the state replicas, that stores the actual replicated state objects. The replication degree can be dynamically configured depending on the

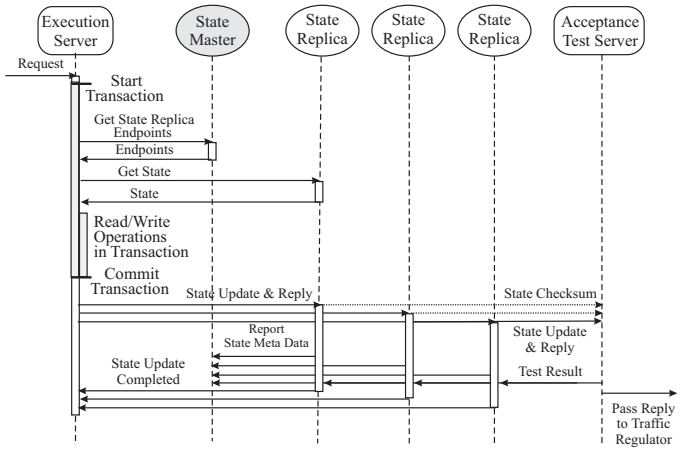

Figure 2: The interactions between the state server, the execution server and the acceptance test server during the processing of an application request.

hardware failure rate. The consistency of the state replicas is mediated by the state master so that expensive consensus algorithm is rarely run during the critical path of the application request processing. The replicas of the state master, however, are coordinated by the Paxos algorithm [3] because the state master maintains the core information for the state such as the checksum and the status (unverified, verified, and invalid) of each version of the state, and those related to ensure exactly-once execution of application requests. The interactions between the state server, the execution server and the acceptance test server during the processing of an application request is illustrated in Figure 2.

2.4 Acceptance Test Server Farm

When an acceptance test server receives a response from an execution server, it loads a pre-defined acceptance test routine and verifies the integrity of the request processing and the corresponding state transition. If the acceptance test succeeds, the state replicas are notified so that they can mark the corresponding state change as valid. As part of the acceptance test, the response is also examined for potential leak of confidential data to an unauthorized client.

If an acceptance test fails, the incident is reported to the traffic regulator to quarantine and analyze the corresponding request, to the configuration manager so that the configuration manager can shutdown the affected execution server, and to the state replicas to mark the corresponding state change as invalid. After the current execution server is shutdown, a new execution server is started from the most recent valid state.

2.5 Trusted Configuration Manager

This component is responsible for all system reconfigurations. It profiles all the nodes in the system (with the help of the health monitor), keeps track of failure statistics, and determines the replication degrees for each component in the infrastructure. It also provides an interface to interact with system administrators, *e.g.,* they can manually increase or reduce the replication degrees, or they may be notified of an urgent need for additional nodes through appropriate means such as email and instance messaging.

On detecting a compromised execution server, the configuration manager forcefully shutdown (or isolate, if it is more desirable for forensic analysis purposes) the faulty node for repair. Once this action is taken, the configuration manager notifies the traffic regulator so that it can forward future requests to an alternative execution server. In addition, the configuration manager periodically initiates proactive software rejuvenation on each of the nodes in the infrastructure to prevent software aging and eliminate undetected intrusions.

Like the trusted traffic regulator, the configuration manager is also replicated asynchronously for high availability. The state of the configuration manager is either predefined (such as the administrator contact information) or can be aggregated by profiling the existing nodes. Therefore, only the primary is active and no consensus algorithm is needed to coordinate the replicas.

2.6 Health Monitor

The health monitor is a logical component. Each node in the infrastructure is equipped with a host and a process failure detector, and a standard intrusion detection system for defense-in-depth. The failure detector periodically reports the health of the monitored node and process to the configuration manager.

3. CONCLUSIONS

In this paper, we presented a blueprint of a novel intrusion tolerance infrastructure. This infrastructure incorporated a number of unique design strategies. The separation of execution and state management enables the possibility of running a single execution server for each request, which will lead to the support of nondeterministic applications and greater runtime performance. The append-only state replication protects the state from being corrupted by a compromised execution server. The acceptance test limits the amount of damage caused by a successful intrusion attack. For future research, we plan to implement the proposed infrastructure in Java and incorporate the software transactional memory technique [5] to achieve highly concurrent execution and the ease of state replication.

Acknowledgement

This work was supported by Cleveland State University under a Faculty Research Development award.

4. REFERENCES

[1] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[3] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):18–25, December 2001.

[4] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[5] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, August 1995.